Contents

List of Figures

1 Abstract

ODH is a decentralized, peer-to-peer open documents sharing system that leverages the ethereum blockchain to enhance the digital self-sovereignty of its users and introduces a voting system for a global governance of the system by its users.

Documents are stored (partially or entirely) by its users thanks to the IPFS protocol, a transparent and reliable system of distributing content. That allows people to manage, synchronize the nodes through By leveraging the speed and redundancy of a decentralized and immutable file storage technology - IPFS (Interplanetary File System), ODH can scale and deliver thousands of documents in a very decentralized manner. This is achieved by storing the encrypted documents permanently on the IPFS swarm.

The set smart contracts deployed on the ODH local blockchain which collects hashes for synchronization of its peers an votes of users. The immutability of the blockchain reduces tremendously the attack surface this data are available for public and its immutability can be checked by comparing hashes.

When everything is transparently stored in the ODH smart contract, we also consider introducing additional governance mechanisms such as an ODH DAO (Decentralized Autonomous Organization) used for voting on new features, documents deletion and other.

Due to the fast-paced emergence of blockchain technology, and high fees within the Ethereum main chain, the ODH system uses a private ethereum network to deploy its smart contracts, the ODH source code itself is stored on IPFS and its hashes on the blockchain to facilitate the upgrade of the software and the way it is used to manage inner mechanics. The main reason to use off-chain data storage based is related to the speed of data loading and data storage. With the progressive shift toward free transactions and proof-of-stake-like consensus algorithms, the scalability of the ODH will certainly be improved and maybe the reliance to off-chain data storage will be irrelevant.

2 Resume

ODH est un système de partage de documents ouverts de facon décentralisé, peer-to-peer, qui tire parti de la blockchain ethereum afin renforcer l'auto-souveraineté numérique de ses utilisateurs et d'introduire un système de vote pour une gouvernance du système par ses utilisateurs.

Les documents sont stockés (partiellement ou entièrement) par ses utilisateurs grâce au protocole IPFS qui est un protocole transparent et fiable système de distribution de contenu. Cela permet aux gens de gérer, de synchroniser tout les nœuds.

En tirant parti de la vitesse et de la redondance d'une technologie de stockage de fichiers décentralisée et immuable - IPFS (Interplanetary File system) tout les documents sont chiffres et stocke en permanence sur IPFS. L' ensemble des contrats intelligents déployés sur la chaîne de blocs locale, permet a ODH de recueillir la liste des fichier disponibles grace a leur hach afin de pouvoir synchroniser les noeuds du reseau.

En raison de l'émergence rapide de la technologie blockchain et des frais élevés des transaction dans la chaîne principale d'Ethereum, le système ODH utilise un réseau privé ethereum pour déployer ses contrats intelligents, le code source ODH lui-même est stocké sur IPFS et ses hachages sur la blockchain pour faciliter la mise à niveau du logiciel et la façon dont il est utilisé pour gérer sa mécanique interne.

La principale raison d'utiliser le stockage de données hors chaîne est liée à la vitesse de chargement des données et au stockage des données. Avec le passage progressif vers les transactions libres et les algorithmes de consensus de type proof-of-stake, l'évolutivité de l'ODH sera certainement améliorée et peut-être que le recours au stockage de données hors chaîne sera hors de propos.

3 Introduction

These days peer-to-peer systems shines more than even in the history of distributed systems, the public attention raised by cryptocurrencies and smart contract platforms like Bitcoin[], Ethereum[], EOS[] or Tezos[] shows how the problems their trying to solve concern more directly humanity as a whole. These problems are obviously trust-related issues[?,?,?], the blind reliance of users to service providers (central banks, centralized cloud, etc..) and of course the awareness that social medias are harnessed by users private data mining coupled with cutting-edge psychological theories designed as huge propagandizing machines to hijack normal behavior to replace them with pernicious wishes of political actors of this global society. The hope of crafting such kind of decentralized technology is the vibrant need of control and freedom of personal data, that is digital self-sovereignty.

Designing an efficient peer-to-peer distributed system after the bitcoin energy issues and personal users data leaks issues after the Cambridge analytica case is, albeit a huge challenge, however it's the right path to follow for a better society. Therefore we do not entirely subscribe to the idea that scientific and engineering endeavor must be ideologically neutral, in our view it's the big flaw of scientific thinking to put social interest aside for a pointless descriptivism[] and pseudo-neutrality dictated ex cathedra by the free market system.

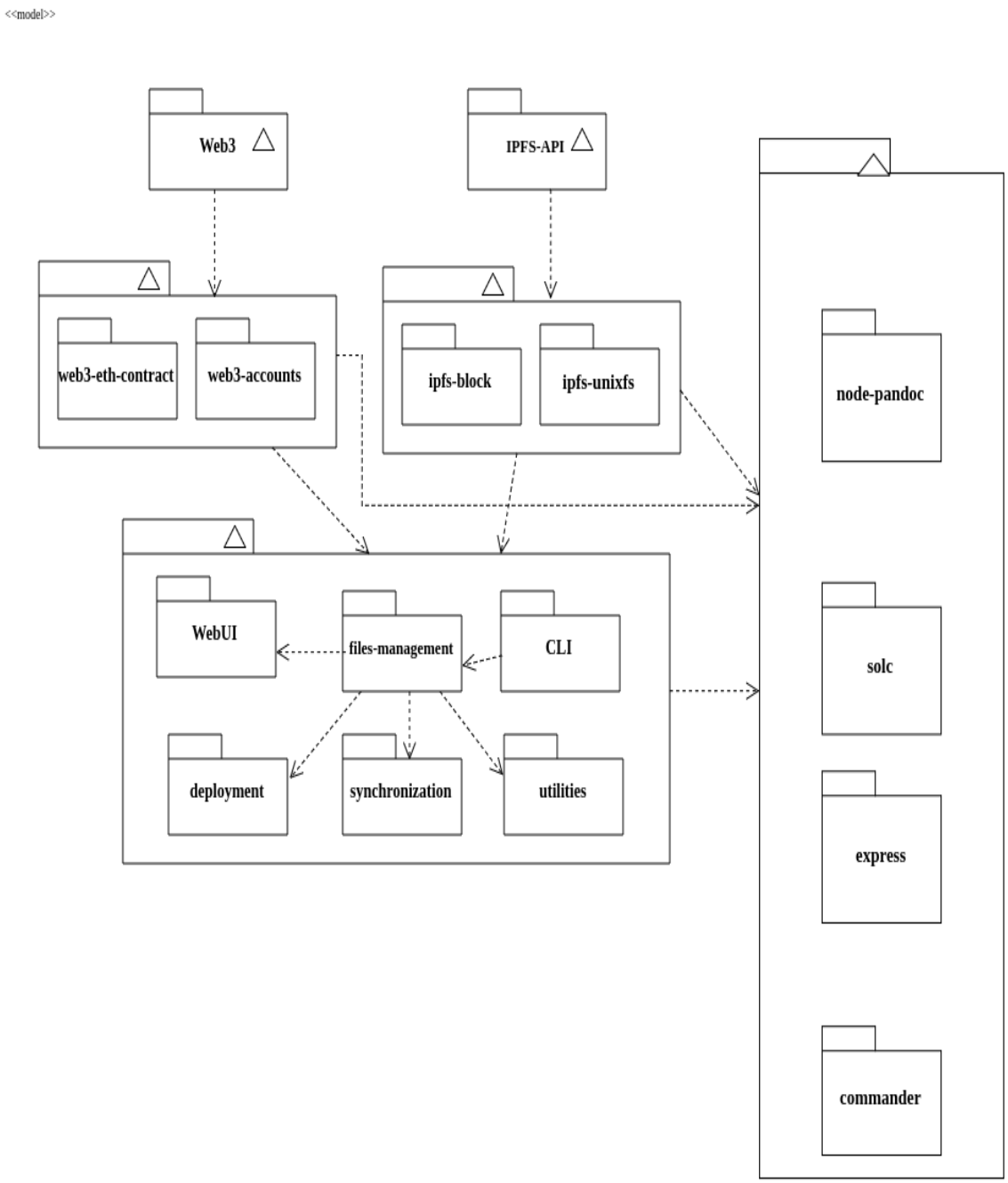
In this document we expose the design and the implementation of ODH, a peer-to-peer open documents hosting through the Golang implementation of IPFS protocol and the Ethereum blockchain and smart contract platform. At a high level standpoint, ODH is a nice coupling of ethereum and go-ipfs with a layer of synchronization, file management (conversion, o-checking, etc ...), and distributed governance through a voting system powered by a couple of smart contracts. ODH is mainly designed to run in an offline context, for a group a friends, researchers, for universities, libraries and so on. The choice to host only open documents is essentially due to legal restrictions as licenses or copyright (DMCA[]) etc... Furthermore open documents in general relies on open formats, which makes files conversion easy. *Respice finem*, in the following

chapters we shall describe the design of ODH: the concept of nodes, their synchronization, voting mechanism, and performance issues. The current implementation of ODH is in Node.js[], due to deadline restrictions, but will be soon released in Haskell and the voting and synchronization mechanism will be soon formalized and proved with Coq.

4 An Overview of ODH

As stated in the introduction, ODH leverages a private ethereum blockchain network and the IPFS protocol in order to achieve offline, peer-to-peer open documents exchange and decentralized administration of a common set data though voting and consensus. In this chapter, we will present an overview of the odh system, its main features and an outline of design of these features.

From a high level perspective, the ODH system structured as the following diagram:



The directory structure of the development project looks as this:

The next sections we will present the IPFS protocol and the Ethereum protocol which are main buildin blocks of ODH. IPFS is dedicated for data lookup and transportation and ethereum for ownership-related issues, voting, and synchronization. The ODH provides some glue code and few smart contracts to make peer-to-peer documents

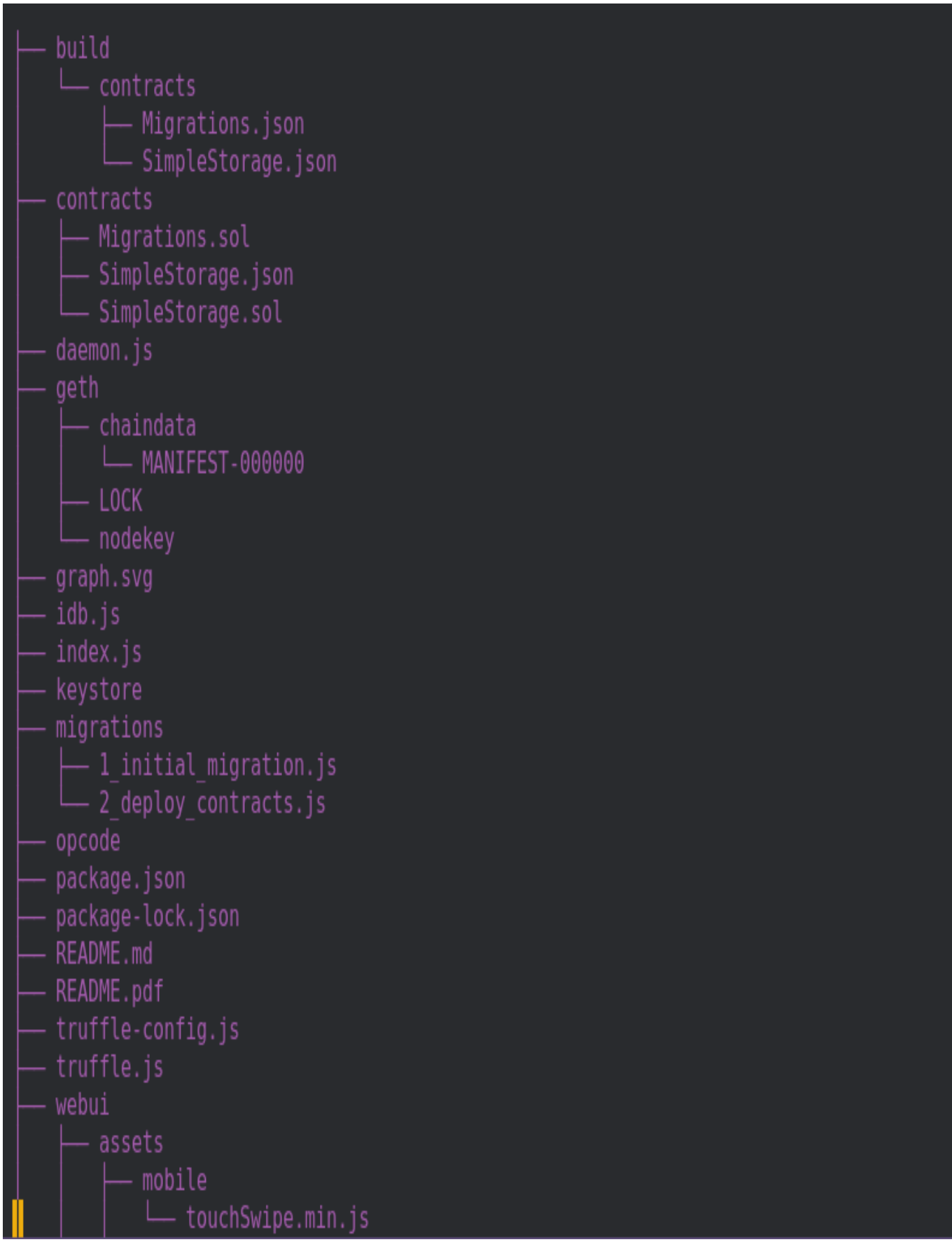


Figure 4.1: Project structure

exchanges administrated by its users.

4.1 1 - IPFS (Interplanetary File System)

Emerging from a project about scientific data sets management, named dat[], by Juan Benet, the IPFS protocol is, according to benet's paper[] , >*"a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of file"*[…] *"IPFS could be seen as a single BitTorrent swarm, exchanging objects within one Git repository."*.

There are multiple implementations of the IPFS protocol in Golang[], Javascript[] and Python[]. Furthermore with the emergence of cryptocurrencies and smart contracts platforms[], mainly influenced by the cryptoanarchy[] movement, the reliance on centralized systems of storage like Azure or Amazon S3, have been defeated, and IPFS was - and is still - the de facto decentralized system of distributing content.

Many[] Dapp (Distributed app) or blockchain-based apps use IPFS to store and retrieved content in a decentralized way, goodbye the single point of control, of failure vanished without hurting and exploiting people. Highly similar to Bittorrent as a peer-to-peer systems, files stored in the network will remain available as long as there is at least one node serving the content requested or - to use Bittorent terminology - seeding it through the network.

4.2 Features

According the IPFS white paper, the IPFS protocol has these features:

- *Multihashes and content addressing*: Also known as self-describing hashes, a multihash is a representation of address is a protocol for differentiating outputs from various well-established hash functions, addressing size + encoding considerations. It is useful to write applications that future-proof their use of hashes, and allow multiple hash functions to coexist.
- *Distributed Hash Tables* In distributed hash tables (DHT) the data is spread across a network of computers, and efficiently coordinated to enable efficient access and lookup between nodes. The main advantages of DHTs are in decentralization,

fault tolerance and scalability. Nodes do not require central coordination, the system can function reliably even when nodes fail or leave the network, and DHTs can scale to accommodate millions of nodes. Together these features result in a system that is generally more resilient than client-server structures.

- *Block Exchanges* A merkle DAG is a blend of a Merkle Tree and a Directed Acyclic Graph (DAG). Merkle trees ensure that data blocks exchanged on p2p networks are correct, undamaged and unaltered. This verification is done by organizing data blocks using cryptographic hash functions. This is simply a function that takes an input and calculates a unique alphanumeric string (hash) corresponding with that input. It is easy to check that an input will result in a given hash, but incredibly difficult to guess the input from a hash.
- *Merkle DAG*
- *Version Control Systems*
- *Self-certifying File System* It is a distributed file system that doesn't require special permissions for data exchange. It is "self-certifying" because data served to a client is authenticated by the file name (which is signed by the server). Anyone can securely access remote content with the transparency of local storage.

That is why the IPFS protocol is suitable of the purpose of ODH.

5 The Ethereum blockchain

The concept of blockchain and smart contracts, has good press right now, however to understand it, we may need represent the kind of conceptual shift it represent with the emergence of cryptocurrencies. First appearing with the bitcoin white paper[], its capabilities has been demonstrated without any doubt from the bitcoin experience since 2009. At current time, the Bitcoin implementation of blockchain have not been compromised. The emergence of the blockchain has seriously undermine the reliance to centralized servers, and the emergence of a new kind of applications called Dapps (Distributed apps).

Instead of having a network, a central server, and a database, the blockchain is a network and a database all in one. A blockchain is a peer-to-peer network of computers, called nodes, that share all the data and the code in the network. So, if you're a device connected to the blockchain, you are a node in the network, and you talk to all the other computer nodes in the network. You now have a copy of all the data and the code on the blockchain. There are no more central servers. Just a bunch of computers that talk to one another on the same network.

Instead of a centralized database, all the transaction data that is shared across the nodes in the blockchain is contained in bundles of records called blocks, which are chained together to create the public ledger. This public ledger represents all the data in the blockchain. All the data in the public ledger is secured by cryptographic hashing, and validated by a consensus algorithm. Nodes on the network participate to ensure that all copies of the data distributed across the network are the same. That's one very important reason why we're building our voting application on the blockchain, because we want to ensure that our vote was counted, and that it did not change. Therefore when a piece of data gets recorded, one of the computers on the network we can be confident that the data was recorded accurately forever.

On ODH, we use a private ethereum blockchain, which means that we reuse entirely the ethereum protocol and its rules, but we start our own blockchain, different from the ethereum *main chain*, on

which real world transactions happen every day.

Once they connect to the network, they cast their vote and pay a small transaction fee to write this transaction to the blockchain. This transaction fee is called “gas”. Whenever a transaction (sync, vote, deletion) is cast, some of the nodes on the network, called miners, compete to complete this transaction. The miner who completes this transaction is awarded the Ether that we paid to transact on the network. But this reward is nearly meaningless, because on ODH all nodes are also miners, and the Ethers of a private network has no concrete value.

However that notion of transaction fees and currency is relatively useful. Every ODH node must mine Ethers in order to perform any transaction. That means that to sync with others any nodes must *pay* few ethers to sync. Therefore as long as the node mine ethers, it can perform all transactions without interruptions. That notion of currency has also an interesting side effect in a private network, black sheep nodes refusing to dedicates few computation power to mine new blocks can not even sync with others is isolated to it’s fellows.

5.1 Smart Contracts

One of great innovation of the Ethereum blockchain, it that it allows us to execute code with the Ethereum Virtual Machine (EVM) on the blockchain with something called smart contracts. At a conceptual level, the idea is not new[], but first working implementation comes actually from Ethereum and after others (Tezos[], Cardano[], EOS, etc.).

Smart contracts are where a part the logic of our application lives. They sustain on ODH a voting mecanism for file deletion, nodes synchronization. This is where we’ll actually code the decentralized portion our app. Smart contracts are in charge of reading and writing data to the blockchain, as well as executing business logic. Smart contacts are written in a programming language called Solidity, which looks a lot like Javascript and then compiled to the EVM bytecode. It is a turing-complete programming language that can allow us to do thing as any other programming language like Javascript , but it behaves a bit differently because it runs on a blockchain

The function of smart contracts on the blockchain is very similar to a microservice on the web. If the public ledger represents the

database layer of the blockchain, then smart contracts are where all the business logic that transacts with that data lives.

Also, they’re called smart contracts because they represent a covenant or agreement and may have legal compliance in them, but that aspect goes beyond the scope of our subject.

In our project, there are four contracts, Voting, SyncContract, DeletionRequest, and Ownership. There are used for synchronization of nodes, voting about deletion management and ownership related activities.

5.2 Main features

- File exchange in the network
- Synchronization of nodes
- File conversion
- Ownership registering and claim
- Distributed file deletion

6 Proof-of-ownership and existence

If we remember debates between scientists, about the paternity of Einstein's relativity[], we have to acknowledge that if every idea, crafted by someone is recorded in a way to prove that he is the person behind the idea, and the time where the idea have been produced there should be no more debates about the paternity of someone concerning ideas and insights. Furthermore that ownership, paternity, is root of many lawsuit cases in the scientific research realm. It's even an issue for publishing and authorship-related activities like books publishers and so on, spectacular claims of plagiarism of intellectual property.

Widely known as proof Of Timestamp or proof of existence, signing the hash of a file into the Bitcoin blockchain has been well-known practice for a while now, and it cryptographically proves that a file has existed at certain time. The record is permanent, and can be checked at will, from the blockchain. Once the hash is reconded on the blockchain, it is nearly impossible to change or remove it.

But what if an idea with all of its mutations are tracked in a kinda zero-knowledge[] way. An individual will be able to proof under certain assumptions that he is the author of of sequences of bytes through this breakthrough way of managing distributed data base through timestamp-based and proof-of-work, stake consensus mechanism.

6.1 Generaties and the existing ways of doing it

6.1.1 The Bitcoin way!

There are two main ways of doing this. The first method involves broadcasting a deliberately erroneous transaction to the networ (known as "provably unspendable"). A provably unspendable transaction has space for a small amount of data to be included in th broadcast, which in this case is used to store the hash of the file. Documentation on this process can be found on the Bitcoin

wiki Using this method, you will only be required to send the transaction with a high enough fee for it to be confirmed. No Bitcoin outp required.

The alternate method involves generating a Bitcoin address based on the SHA-256 hash of the file and sending Bitcoin to said add With this method, you will never know the private key that is associated with the address. This is because a ECDSA key is never generated and hashed, instead you use the hash of the file that you want to sign into the blockchain. Take a look at the process of generating Bitcoin addresses on the Bitcoin wiki. Contrary to the first method, this does require a Bitcoin output. At the current time this isn't very significant, since you could send just 1 Satoshi (0.00000001 BTC), which is worth £0.0000073148 as of 19/Jan/2017. Many transactions created this way and used for the purpose of signing file hashes can be seen on the "Strange Transactions blockchain.info.

Bitcoin transaction fees are currently around £0.07 for a transaction of this sort. To summarise, this is how much it will "cost" to si the hash of your file onto the blockchain. The problem with this system is that it does not prove when the final version of the file was actually created. It is possible to sign a decade file into the blockchain and it would only prove that the file was created BEFORE that time. While this type of proof may be useful some situations, a two-way solution is better. The only real way to achieve this is by including some information in the file itself that proves that it was created AFTER a particula time. The first thing that you may think of is to include a current news article, since this information could not have been produced before the reported event actually happened. While this is a good solution that most people will understand, it is better to use a cryptographic proof. By including the most recent Bitcoin blockchain block hash in your file, you can cryptographically prove that t file was created AFTER the current time.

Now we have the full solution: by including the hash of the most recent Bitcoin block in the file, and then signing the hash of the fi onto the blockchain, you can prove that the final version of the file was created between two timestamps. These being the timesta of the most recent block hash at the time, and the timestamp of the Bitcoin transaction. In order to test the proofs, we can imagine different circumstances:

If the file were to be edited, the content of the file after its published it with proofs, the hash of the file would differ from the copy in th blockchain, therefore violating the proofs. If a

malicious user wanted to fake the timestamp of the file in order to make it look like it was created earlier than it actually was, the consensus mechanism behind of the blockchain, will reject it. But it would be impossible to sign the hash of the file into an older block. There is way to change the contents of older blocks on a blockchain since every future block directly depends on it. That's why blockchain technology is so revolutionary. If a user wanted to fake the timestamp of the file in order to make it look like it was created in the future, we would be unable to incl a future block hash or sign the hash of the file into a future block on the blockchain since neither of those exist yet. If we wanted to verify the timestamp integrity of the file, we would check the timestamp of both the included block hash and the asociated Bitcoin transaction, and make sure that they are within an acceptable time range of each other. This verifies the proofs. Signing the hash of a file onto the Bitcoin blockchain can be done manually as outlined above, however there are online services available designed to automate the process:

6.1.2 Thirty party services

- *Proof of Existence* (proofofexistence.com) is a website which automates the process described above. The user submit possibly the most well known service. You simply select a file, it is hashed and an add is presented to which you must pay 0.005 BTC (worth £3.67 as of 19/Jan/2017).
- *OriginStamp* (originstamp.org) is much more versatile than Proof of Existence. There are more options as to what you can submit the site, instead of just uploading a file, and it's also a completely free service!

Instead of having a separate Bitcoin transaction for each verified file, all of the hashes for every file submitted to the site each day are aggregated into one master file which is then used as the seed for generating a Bitcoin keypair. This means that there is only one Bitcoin transaction per day that verifies all of the files that were submitted that day. The downside of this is that you may have to wait up to 24 hours from submit your file for the actual Bitcoin transaction to take place. There is an optional 1 dollar fee to have your transaction be sent instantaneously.

OriginStamp provide an online interface to verify your file, however in the event that the site goes down it is still possible as long a you have the master file of hashes for the day that you

submitted your file. This can be downloaded from the site at the end of every day when the transaction takes place.

If you do not wish to sign the hash of your file onto the Bitcoin blockchain, you could use a third-party service. There are many services suited to this kind of use, including Archive.org and browser caching sites, such as Google Cache. You could also post the to any social media site that has timestamping, however the integrity of these timestamps may be questionable as there is no cryptographic proof behind them.

When it comes to including these proofs with your file, the most recent block hash is easy. However, including the hash of the file signed into the blockchain is much more difficult. While it is technically possible for a file to contain its own hash, it is not realistica possible and would require a large amount of time (probably millions of years) and computing power to achieve. The best solution this would be to predict the block number that your file hash will be signed into and include it in the file before signing. Ideally, thi would be the current block number + 1, however if your transaction fee is too low or the network is congested, you might not mak in. If you are using Proof of Existence or a similar service, they provide an interface to search for hashes that have been signed usi their service. Even if the service goes offine, you should still be able to find the transaction since the signature of the transaction i prefixed with some custom text in order to make it stand out.

6.2 On Ethereum with smart contracts

With the emergence of ethereum blockchain, writing data on the blockchain has become less tedious with the implementation of smart contracts. There is no need to perform complicated tasks to include the hash as a transaction on Bitcoin. A simple smart contract, written in few line can produce the expected outcome, namely the proof of ownership of the documents.

The set of assumptions used to achieve such a thing is pretty narrow, they are namely: - The data submited to the networks are inserted into a block and chained together with the hash of the previous block except of course the first block (genesis block); in that case the rule is that to be considered as the member of the network you *must* use the same genesis block with other nodes of the network. - When a block is chained with others it’s impossible to change it without changing the hash of that block

and the hash of the merkle root. - blocks are timestamped so if a block is chained at a certain time it is a proof that, the data inside had been inserted at that time prior to the argument that the blockchain is immutable.

An individual comes with a piece of file (an o-document in our case). The ow Therefore we simply need to insert the file itself but this solution is useless because the hash of the file is sufficient for our proof and also that a hash fits perfectly with our zero-knowledge context. ## Implementation level

At the implementation level, we have the implementation of the contract itself and the glue code used by the command line tool to use from a simple interface to assemble all the peaces to make the ownership registration an claim possible. The Ownership contract is structured like this:

```
contract Ownership {
  event Insertion(uint h, uint h_);
  event Reading(uint h1, uint h2);

  function setOwnership (uint h, uint h_) payable public { }
  function getOwnersList() public  payable returns (uint h,
     uint h_) { }
}
```

Inside, we found two methods and two events. The setOwnership method store the hash of the files containing the list of all ownership registration. That method is call each time there is an ownership registering. The hash submitted is first inserted or appended to a file dedicated to store all registrations. It is the hash of that register that is stored on the blockchain through the contract.

The list of owners is then used to check ownership claims. In that occasion, the getOwnersList method is used by any node to perform the ownership check. For that, we extract the hash form the blockchain, pull the file through IPFS and then check whether it is in that list or not.

Here is the full ownership contract.

```
contract Ownership {
  uint hashBlock1;
  uint hashBlock2;
  event Insertion(uint h, uint h_);
  event Reading(uint h1, uint h2);
```

```
  function setOwnership (uint h, uint h_) payable public {
    hashBlock1 = h;
    hashBlock2 = h_;
    emit Insertion(h, h_);
  }
  function getOwnersList() public  payable returns (uint h,
     uint h_) {
    h = hashBlock1;
    h_ = hashBlock2;
    emit Reading(hashBlock1, hashBlock2);
  }
}
```

7 ODH nodes and synchronization

7.1 Full nodes, light nodes and simple clients

At the early stages of the development of ODH the concept of full nodes weren't seen as relevant because of the fact that if one node is storing a couple of files as node of ipfs therefore, the ipfs protocol makes access to these files as if they were actually stored by the requested peers of that 'server'. The mechanics behind such a thing isn't relevant here, however it yield an issue that peer-to-peer networks are designed to solve. Indeed this prior design choice has the side effect to foster centralization instead of decentralization. If a collection of files is pinned by a single node its availability in the network hangs on that node, which leads inexorably to the so-called single point of failure from data serving perspective. The aim of peer-to-peer networks is exactly the opposite. Here comes the switch to the idea of using full nodes.

In itself, there is nothing new about them at all. That thing is as old as well-known peer-to-peer systems as BitTorrent, IPFS, Bitcoin, Ethereum and so on.

As mentioned in [1] there are two kind of node in the network. Full nodes and light nodes. A full node stores entirely the ODH blockchain and the files submitted by users of the network. It also participates in the governance of the network as vote for the deletion of a content, validating transactions in the blockchain by performing a cheap proof of work, and storing the entire submitted files and voting for the upgrade of the protocol designed to rule how nodes should operate in the network.

Light nodes stores only the header of blocks and however they do not validate blockchain transactions. Concerning documents, they can decide witch files they decide to store. Internally they are simply IPFS nodes serving a subset of our hashes data bases with a layer of files management procedures.

Simple clients to the ODH network can be seen like simple IPFS nodes they may use the go-ipfs client tool or using the odh-cli

tool to interact with ODH.

7.1.1 Full node architecture

A full node is actually simultaneously an full node of our private ethereum network and an ipfs node. From the ethereum perspective, the node stores entirely the odh blockchain, perform proof of work for blocks mining and validate transactions in the network. Fortunately, the network size could'nt be as big as the ethereum main chain, hence the proof-of-work will be cheap and pretty fast. Concordantly from the IPFS perspective, systematically all files handled by ipfs is pinned to its requested node, hence physically stored on that computer.

7.2 Synchronization of full nodes

Synchronization is an actively studied set of method in the distributed systems research field. To state it simply, let be two nodes damdam and moussa. Damdam is the server and moussa is client of damdam, to sync these nodes we need to pull the set of files stored by damdam and copy them to moussa. These files are downloaded from their multihashes stored also by damdam. How that thing is possible ?

The idea is to keep track of the set file hashes from damdam in a file called hashesbase. Then we add that file to the IPFS network and send its hash to moussa in order to get the file (hashesbase). Then moussa himself perform the ipfs get operation on each hash from the hashebase. Hence moussa is synchronized with damdam. But what if moussa add a new file to his data base? His hashebase will be different from the hashebase of damdam. We simply have to invert the roles here mousa takes place of damdam and viceversa. The figure below shows a sketch of the algorithm of synchronization.

Its implementation is pretty straightforward, it stands in few lines.

```
const syncWithPeers = () => {
    var config = loadConfig();
    bin = config.code;
    abi = config.abi;
    account = config.account;
    contract = new web3.eth.Contract(abi);
    contract.options.address = config.address;
```

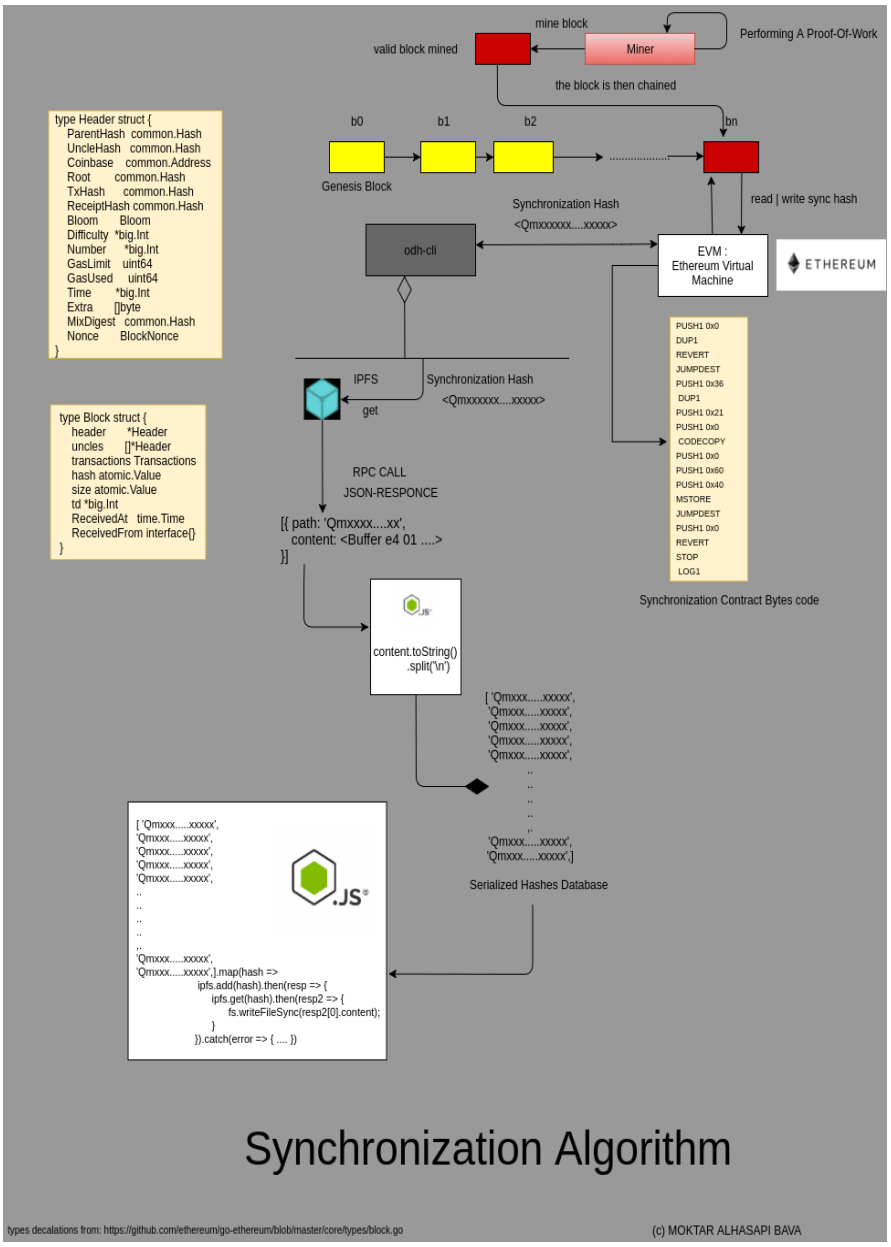


Figure 7.1: Sync Diagram

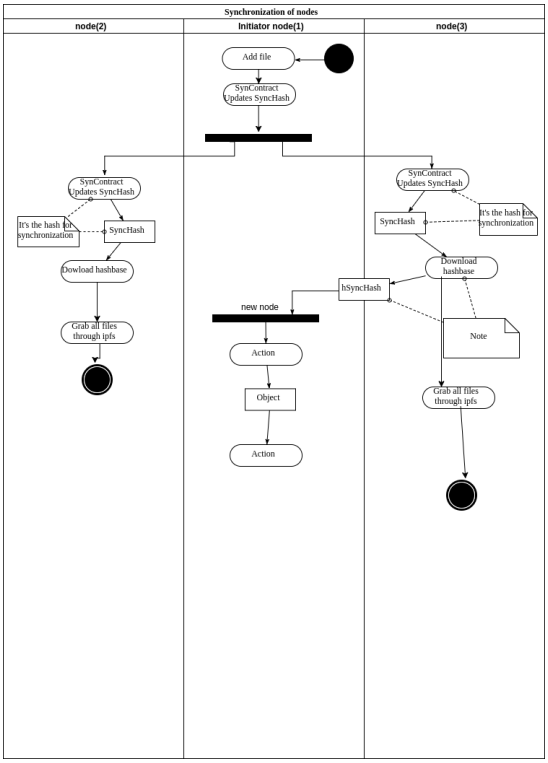
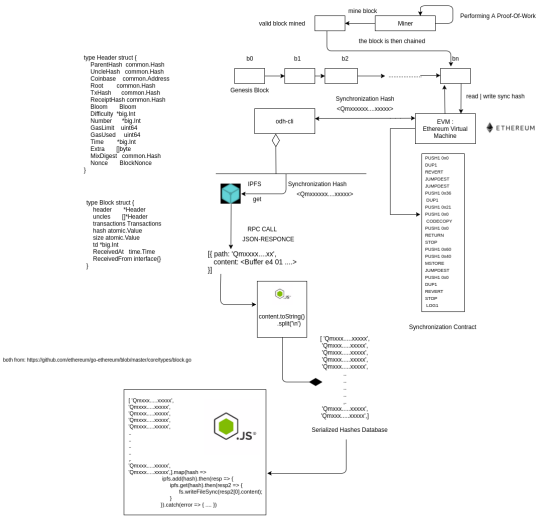


Figure 1: SYN_ACT

1

Figure 7.2: Synchronization activity diagram



Synchronization Algorithm

Figure 1: ODH_SYNC

1

Figure 7.3: Synchronization Overview

```
    console.log(”The synchronization with peers is starting:”)

    web3.eth.getAccounts().then(accounts => {
        contract.methods.get().call().then(t => {
            var fst = web3.utils.toBN(t[”0”]);
            var snd = web3.utils.toBN(t[”1”]);
            var hash = fromBNtoIpfs(fst) + fromBNtoIpfs(snd);
            console.log(hash)
            ipfs.get(hash).then((obj) => {
                hashes = obj[0].content.toString().split('\n');
                for(var j in hashes) {
                    ipfs.get(hashes[j]).then(z => {
                        var s = z[0].path;
                        console.log(chalk.bold.blue(”[ ! ]”),s,
                           ”downloaded successfully.”)
                    })
                }
            })
        });
    })
}
```

It basically extracts the synchronization hash from the blockchain through the contract.methods.get function from the synchronization contract.The synchronization hash is a hashlink to the hashbase which contains the hash of all indexed files in the network. The next step is to pin each file from its hash in order to store it locally. That is done by using the pinning methods from the IPFS API.

In the smart contract: We store only the hash of the hashbase which contains, as previously mentionned all indexed files in the network.

```
contract SyncContract {
    .
  function set(uint h, uint h2) public payable returns (uint,
     uint) {
      ipfsHash1 = h;
      ipfsHash2 = h2;
      return (ipfsHash1, ipfsHash2);
  }
  .
  .
}
```

The synchronization contract contains two functions set and get. The former function is used to insert the synchronization hash in the blockchain. Every node planning to write data on the blockchain must pay few ethers, that is why the set method is public and payable. And of course in order to able the pay the transaction fees, each node must mine block and the rewarded with ethers by doing so. Therefore if there is no miner in the network there will be neithere a way to add a document nor a way to synchronize with other nodes. Furtheremore, because of the size of ipfs multihashes (46 bytes), we have splited them in two parts (23 + 23 bytes) in order to store them in two uint Solidity integers, that is way the set and the get methods of the SyncContract takes and respectively returns a tuple of unsigned integer. For recall uint represents values in 32 bytes.

```
contract SyncContract {
  .
  .
  .
  function get() public view returns (uint, uint) {
    return (ipfsHash1, ipfsHash2);
  }
}
```

The latter method (get), reads values inserted in the blockchain through the set method and returns a pair of integers. Here is the code of the full synchronization contract:

```
contract SyncContract {
  uint ipfsHash1;
  uint ipfsHash2;
  function set(uint h, uint h2) public payable returns (uint,
     uint) {
      ipfsHash1 = h;
      ipfsHash2 = h2;
      return (ipfsHash1, ipfsHash2);
  }

  function get() public view returns (uint, uint) {
    return (ipfsHash1, ipfsHash2);
  }
}
```

Therefore as long as each instance of the blockchain are coherently synced with others, and according to the fact the hashbase on all nodes are - generally - append only, a new nodes cannot miss files.

8 Voting mecanics: Deletion requests and the voting process

One of the most attractive feature of smart contracts is the possibility to make create DAOs (Decentralized Autonomous Organizations)[]. It makes utterly straighforward the management of an entire organization (actioners, funds, decisions, administraction, etc.) on a a smart contract through a voting mechanism.

On ODH we've implemented a little DAO upon which some of decisions on the network are possible through voting of the members. The only supported decision making mechanism is deletion of documents. We choosed deletion of demonstraction purpose and because many online documents hosting websites like b-ook.org or gen.lib.rus.ec support a certain kind of deletion according to a valid legal claim. Thus, is a certain document indexed by ODH and hosted by all full nodes are subject to a legal concern, ODH provides a mecanism to vote for the deletion that documents in the case vote a reached a consensus or the proponents for the deletion reached more than 50% of the vote.

8.1 Deletion request

Whenever a member of the network witnessed for example that classified informations for the documents are exchanged by people and he wanted to stop its spread because of legal claims, that member *must* submit a deletion requested on network. That request is propagated to each node for vote. The structure of the deletion request contract is the following:

```
contract DeletionRequest {
  constructor () public {}
  function setRequest(uint h, uint h_) public payable { }

  function isPending () public view returns(bool res) {}
  function getHash () public view returns(uint, uint) {}
```

```
  function stopReq () public payable { }
}
```

When the contract is deployed, the blockchain register the fact that there is not pending request. if so no user can submit a deletion request till the end of the request. There are two way to do so, first members have voted the fate of the file so the request ended or the submited have stoped the request himself.

In the DeletionRequest contract, there are four methods. The setRequest that enables the insetion of the hash of the targeted file, it’s obviously payable for the reason that nodes refusing to participate in the maintenance of the network by mining blocks cannot submit the request. As other contracts on the network, it takes two uint integers in order to store a single IPFS’smultihash. Then it writes the hash on the blockchain broadcast it to the network.

The isPending method checks if there is a pending request and the getHash method extract the hash of the targeted file from the blockchain. And finally the stopReq designed to cancel a deletion request either performed by a submiter or voters members.

Here is full code of the deletion request:

```
contract DeletionRequest {
  uint h1;
  uint h2;

  address owner;
  bool pending;

  constructor () public {
    pending = false;
  }

  function setRequest(uint h, uint h_) public payable {
      owner = msg.sender;
      h1 = h;
      h2 = h_;
      pending = true;
  }

  function isPending () public view returns(bool res) {
    return pending;
  }
```

```
  function getHash () public view returns(uint, uint)  {
      return (h1, h2);
   }

  function stopReq () public payable {
    if (msg.sender == owner)
      pending = false;
  }
}
```

8.2 The vote for the deletion of a document

Like any voting system, there should be a way to identify voters without collision of idenitities. On Ethereum, Each member is identified by its unique ethereum address. On the Voting contract, voters are reperesented by:

```
  mapping (address => bool) voters;
```

Votes are represented by a boolean. The true value means the acceptance of the deletion and false its rejection. Each vote is counted from the previous mapping as such that there is no way to increase the number of votes, however a member can change its votes at the condition that there is a pending deletion request. The voting contract is structured as:

“‘javascript contract Voting { uint number; uint accepted; address owner;

mapping (address => bool) voters; constructor () public {} function vote (bool choice) public payable { } function positiveOutcome () public view returns(bool) { } function negativeOutcome () public view returns(bool) { } }

We have the voters and three methods. The vote method which any node can use to vote and two function to check whether there is a positive outcome of the voting process or a negative one. The vote is finished if all nodes participated.

Here is the full code of the Voting contract:

“‘javascript contract Voting { uint number; uint accepted; address owner; mapping (address => bool) voters;

constructor () public { number = 0; accepted = 0; owner = msg.sender; }

```
function vote (bool choice) public payable { voters[msg.sender] =
choice; if (voters[msg.sender]) { accepted += 1; } number += 1; }

function reachedConsensus () public view returns(bool) { return
(accepted > number/2); } }
```

9 Design and implementation difficulties

9.1 Hardware difficulties

As a network-based app, it was necessary to get access to few computers for testing each feature of the app. We have to also add that most of the members the development team weren't familiar enough with Linux distributions and their machine were, *PB hev* computers, on which, - after several attempts - have failed to host a proper linux distribution as expected. Thus for testing, we've relied essentially on - highly slow - Virtual Machines and upon an old desktop computer.

9.2 Ideological standpoint of the cryptoverse and The maturity of technologies involved

At high-level scale, the practice of designing Dapps (Distributed Apps) is, unlike the case of regular networking applications, at its early days, where there isn't a well-defined and standardized procedures of producing such kind of apps. Dapps unlike, regulars apps embed within theirs most basic architecture a deep connection with cryptocurrencies, peer-to-peers exchanges and a high will to rewrite the structure of the web itself.

Most of the developers from the cryptoverse are cryptoanarchists[], they are inclined to reject[] standards, and hence are likely to follow their own way of structuring apps, smart contracts, how to test the apps and so forth.

9.3 Design methods and implementations languages

At the early stages of the development of ODH, go (or Golang) have been selected as to the main implementation language for both

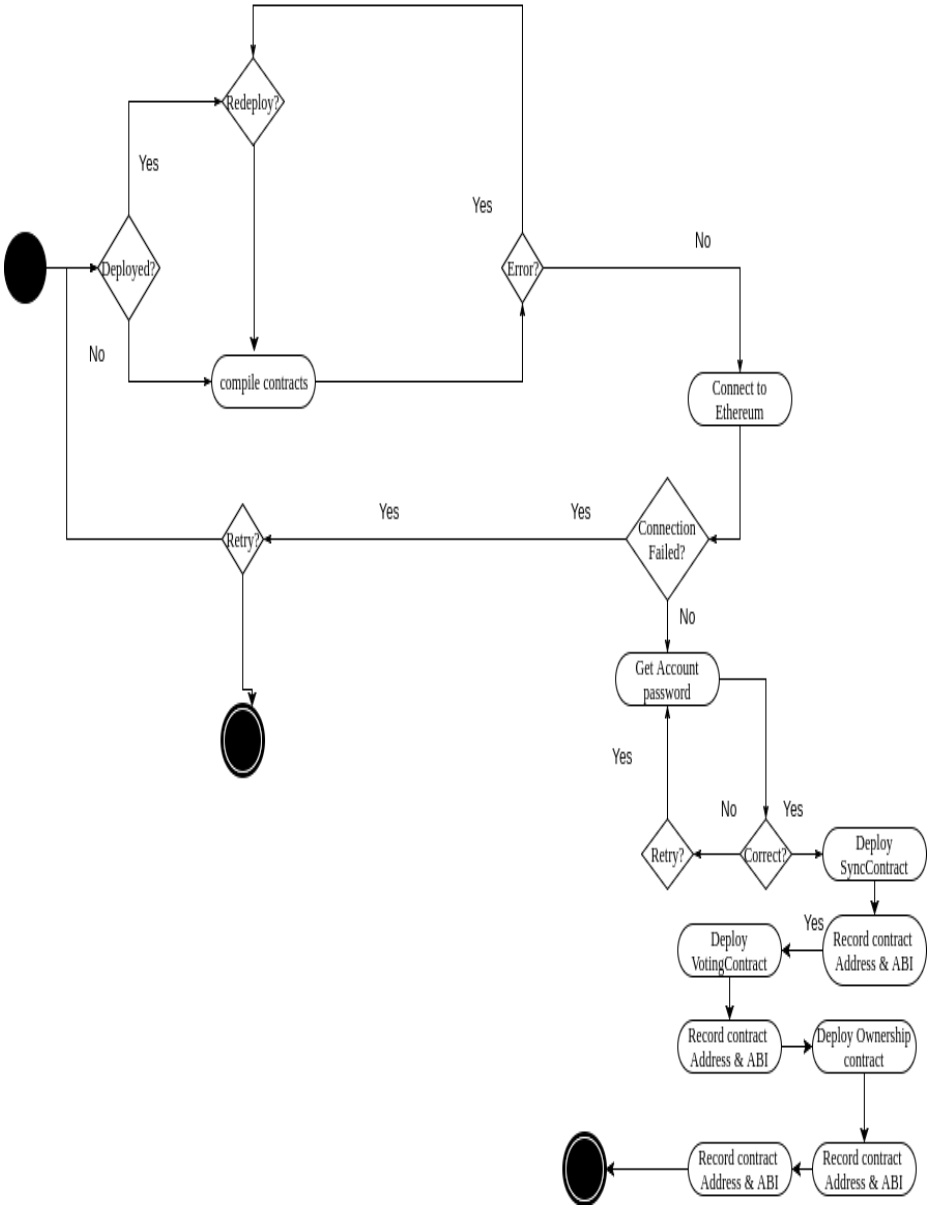
the command-line client and the web server itself. The reasons behind the choice was, at that time, related to the fact that, the reference implementation underlying technologies behind ODH were go-ethereum and go-ipfs. Our aim was to *tweak* the internals of these code bases as far as possible, so that they will be capable to support the set of requirement elaborated in the (**???**).

We have tried to rewrite the ethereum peers discovering protocol for an offline usage of the protocol and coupling it with ipfs swarm protocol unfortunately the time to test, an assess the code weren't sufficient to follow that path further, therefore we've left it to a more prosaic implementations on Nodejs[]. After few weeks of redesigning the app from a *Javascript on Node* perspective, we have also struggled with the extensive reliance of existing libraries (web3 and ipfs-api) on Javascript's Promise and asynchronous functions, which is a huge leap from our traditional ways of doing thing. Even at the current state of the project, these libraries do not match as expected to their docs. to interact with geth, the go-ethereum client implementation and testing,

9.3.1 The case of web3

Being the *de facto* library for interacting with the ethereum blockchain, the web3 library suffers mostly the problem of undocumented features, outdated documentations, and an unstable API. During the developement of ODH, we've struggled with the smart contract API of web3 and contradiction between the official documentations of the library and the current implementation.

10 The Architecture of ODH: Overview diagrams and algorithms



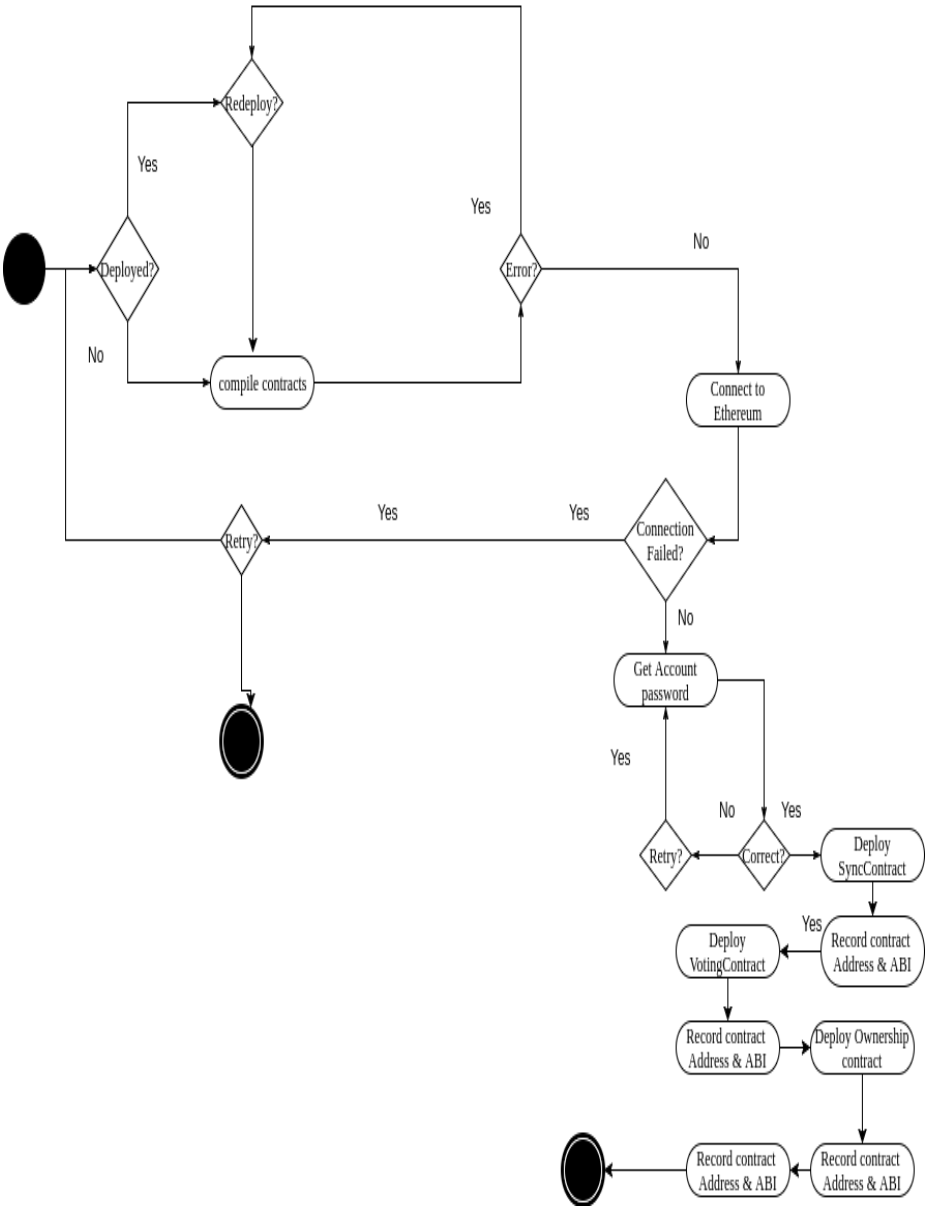


Figure 10.1: Smart contracts deployement

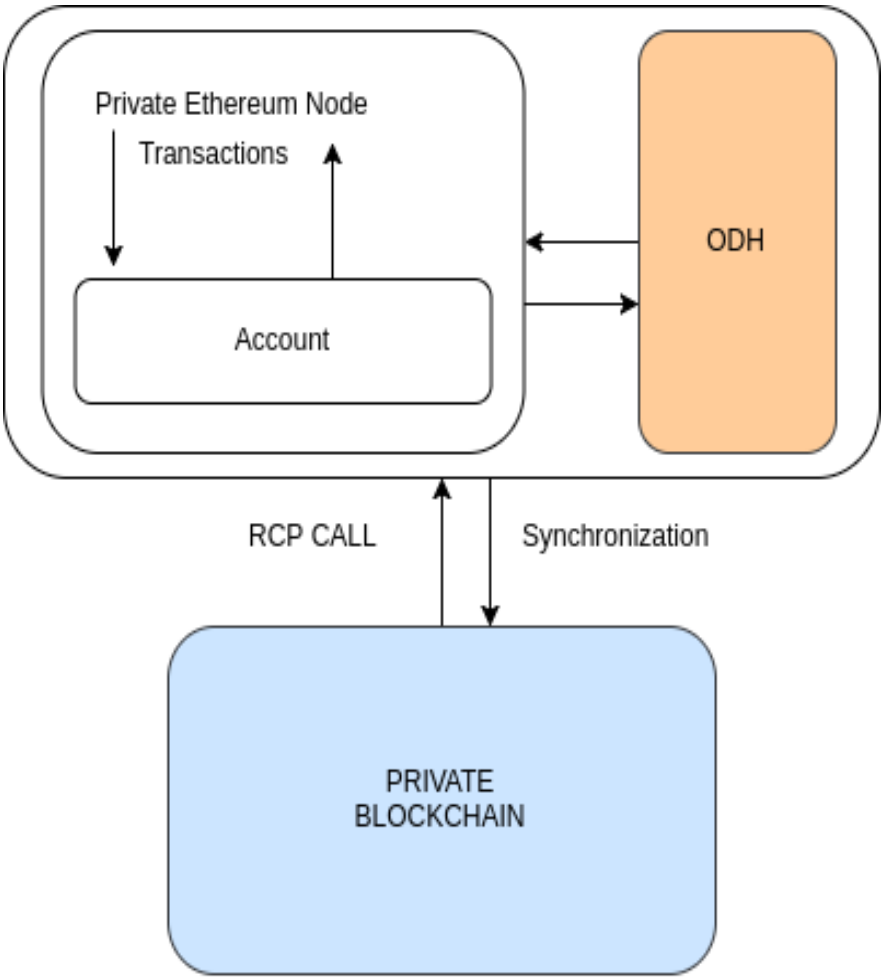


Figure 10.2: COMPONENTS INTERACTIONS

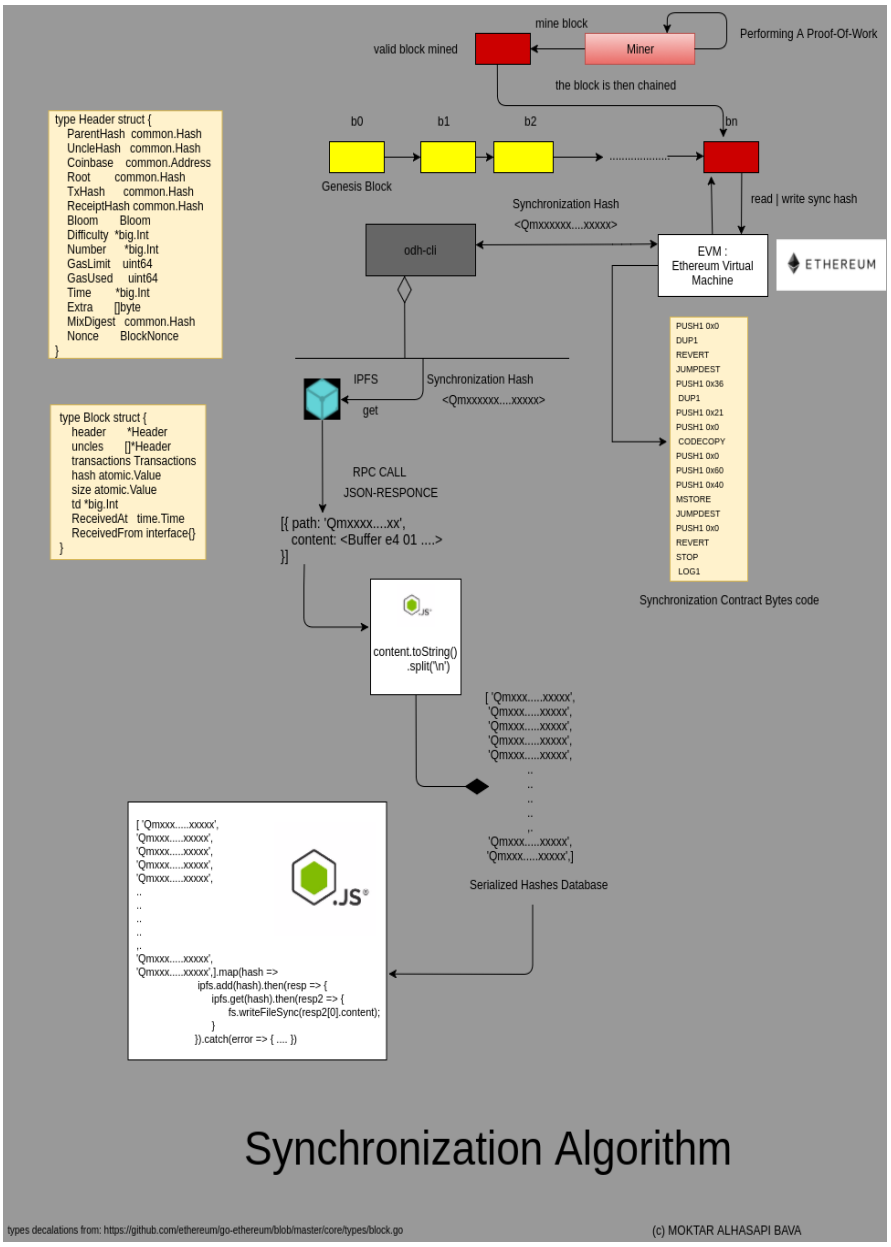


Figure 10.3: SYNCHRONIZATION

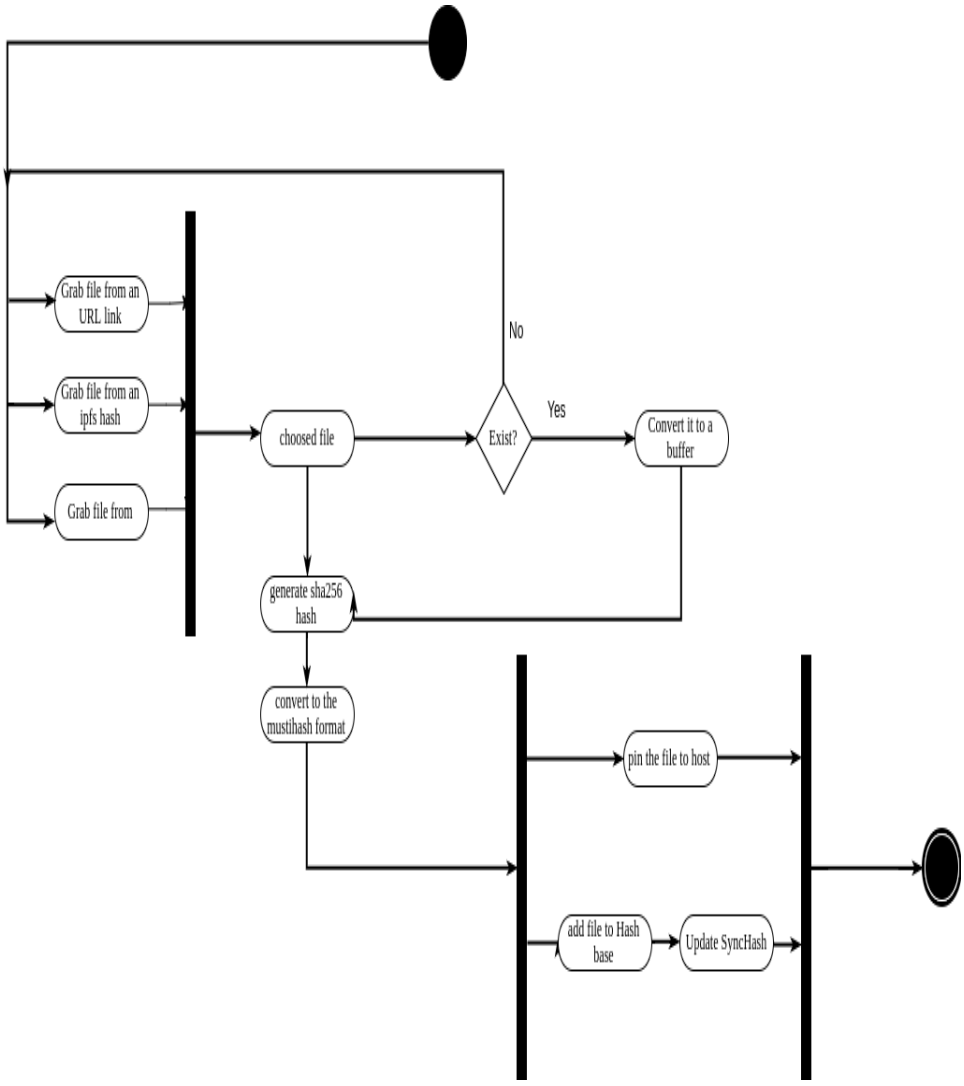


Figure 10.4: FILE ADDING

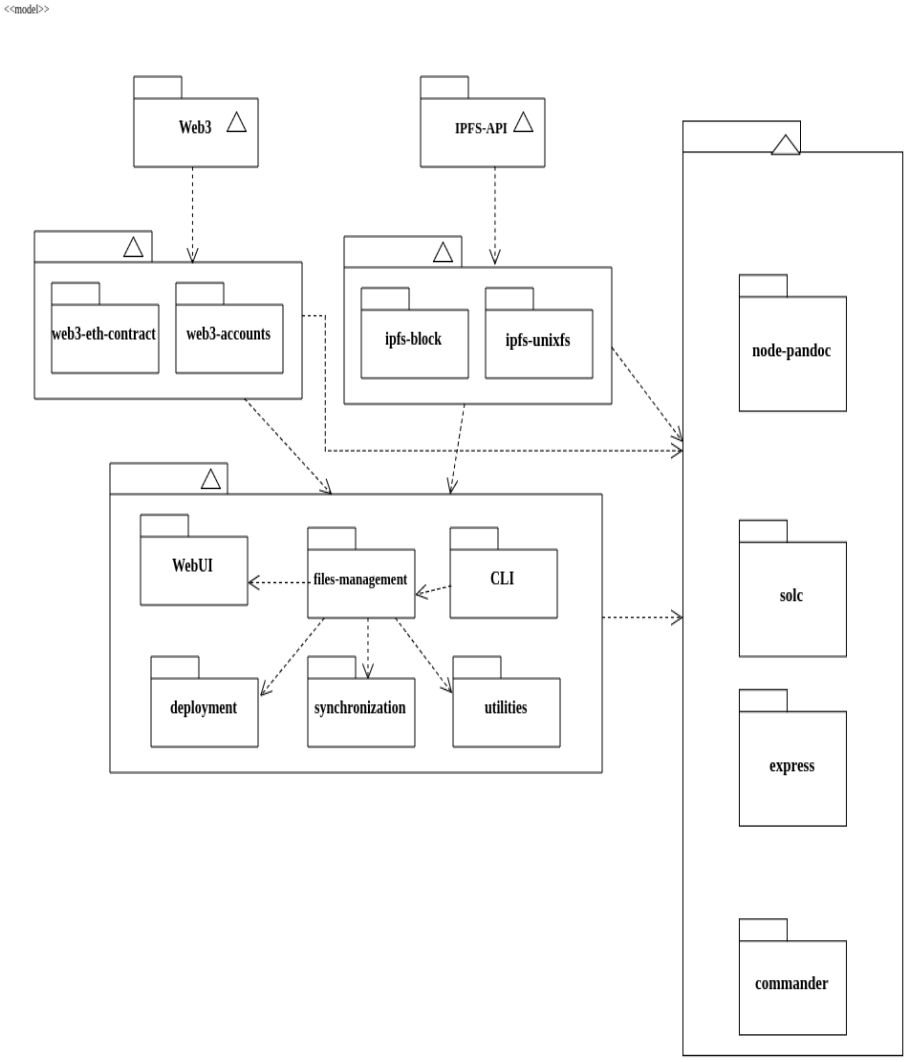


Figure 10.5: STRUCTURE AND COMPONENTS

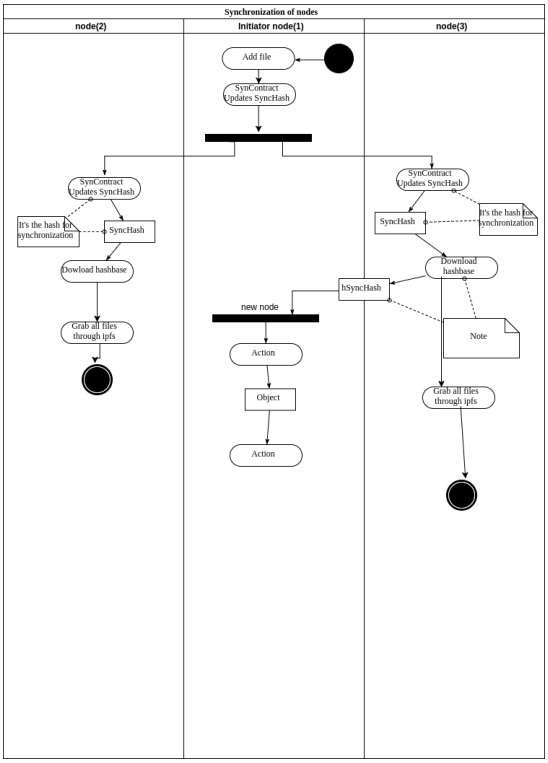


Figure 1: SYN_ACT

1

Figure 10.6: ACITVITY DIAGRAM

11 The User Annex: How to use ODH

11.1 The ODH command line

The ODH command line interface is currently the *only* way to interacts with the network. The next block presents available commands on CLI interface.

```
 Usage: odh [options]
 odh command-line interface of managing distributed files-
 Options:

-V, --version                    output the version number
--add <file>                     File Inclusion to ODH
--setup                          Setting up the network
   [contract compiling & deployement]
--add-remote <link>              File Inclusion to ODH
--from-ipfs-hash <hash>          Include files from an IPFS
   Hash
--sync                           Synchronize the new node
   with the current state of the network
--o-check <file>                 Check whether a file is an
   open document
--owning <file>                  Registering an ownership
   claim
--claim-of-ownership <hash>      Checking whether a claim of
   ownership is valid
--webui                          Lauching the web user
   interface
--vote-for-deletion <hash>       Vote of the deletion a file
   in the network.
--deletion-req <hash>            Supply a request for the
   deletion of a file in the network.
--full-node-req                  Lauching the process of
   becoming a full node
--light-node-req <hash List File> Launching the hosting of a
   set of files
```

```
--search <hash | name | regexp>   researching files in the
   network
--stats                          Display some statistics
   about files in the network
--peers                          Display the set of peer
   connected to the network
--convert <hash> <file.ext>      Convert a hosted file to
   different format according to .ext
--revoque-current-req            Suppend the deletion request
--fetch <hash | name>            Write in the current
   directory the file hashed <hash>
--id                             Getting the identity of the
   node
--garbage-collect                Remove unlinked or unpinned
   IPFS files
--reset                          Remove configuration files
--list-files                     List pinned files
--author                         The author
-h, --help                       outputs usage information
```

11.1.1 Setting up the network

In order to add content to the network the user *must*, first setup the network. The setup process assume that there is already a running instance of go-ethereum and go-ipfs daemons, if not the setup process will fail. To launch these to daemons type:

```
$ geth --datadir="./" --networkid 23422  --rpc
  --rpccorsdomain="*"
    --rpcport="8545" --minerthreads="1" --mine --unlock 0
    --rpcapi="db,eth,net,web3,personal,web3" --port 30303
       console&

$ ipfs daemon&
```

Next, the user can now setup the network for a new fresh use of ODH. For that the user must type:

```
$ odh --setup
```

The program will then outputs something like this:

```
[ * ] Compiling the contract
[ * ] Compilation finished
[ * ] Accounts list:
[ '0xd7A01610F3839cCF162F1E1a3e4b6c8242EDFCD4' ]
```

```
0xd7A01610F3839cCF162F1E1a3e4b6c8242EDFCD4
624999999999000000000
```

The previous output is the process of smart contracts compilation and private ethereum account selection. Next, the use should type the password of his ethereum account.

```
ethereum account password: **************
```

Then the program will deploy these sequentially the synchronization, voting, deletion, ownership contracts on the blockchain, in with they we'll live forever.

```
[ * ] +> Ready to deploy SyncContract contract
[ ! ] +> Contract sucessfully deployed at:
   0xfBAB107943168a7898cfC156E26Ba07131316c5A
[ * ] +> Ready to deploy the Ownership contract
[ ! ] +> Contract sucessfully deployed at:
   0xddd661D1AB0edac74B2a857f9a1c77D69f6127D9
[ * ] +> Ready to deploy the Voting contract
[ ! ] +> Contract sucessfully deployed at:
   0x4c2291DdbDa210731Afb540fC0c79146FD52CE87
[ * ] +> Ready to deploy the DeletionRequest contract
[ ! ] +> Contract sucessfully deployed at:
   0x9580ed617A6b6E572543590bc184D0151cf79309
[ ! ] Network setup terminated finished.
```

At this level the odh system is ready to be used.

11.1.2 adding files to ODH

There are three ways on add a file to the network, from the local file system, remotely and from an ipfs hash. * From the file system The user must type something like:

```
$ odh ––add /path/to/file.pdf
```

- Remotely The user must type something like:

```
$ odh ––add–remote https://www.link.com/files/manifesto.pdf
```

11.1.2.0.1 from-ipfs-hash

Include files from an IPFS Hash

```
$ odh ––add–from–hash
   QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
```

11.1.3 sync

Synchronize the new node with the current state of the network bash $ odh ––sync ### o-check Check whether a file is an open document bash $ odh ––o—check path/to/file.doc ### Registering an ownership claim To perform ownership registering the user must type:

```
$ odh ––owning ~/Documents/Baudrillard/8.pdf
```

And it outputs:

Ownership claim registered successfully. File hash: QmZ6vPS45mVqLMh4NNemPYY8 Date 2018-07-18T06:17:02.913Z The user must keep the hash to prove the ownership.

11.1.4 Checking whether a claim of ownership is valid

Here the hash the choice to submit either the file or its hash.

```
$ odh ––claim—of—ownership
   QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
```

or

```
$ odh ––claim—of—ownership ~/Documents/Baudrillard/8.pdf
```

and it will outputs:

```
SUBMITED HASH  QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
Result {
  ’0’:
     ’77991722633929904953535340207214769289098226303112543 61’,
  ’1’:
     ’54057575330679965378331690010637880568473072983027528 37’,
  h: ’77991722633929904953535340207214769289098226303112543 61’,
  h_: ’54057575330679965378331690010637880568473072983027528 37’
     }
REGISTERED HASH:  QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
[ ! ] YOU ARE THE OWNER OF THE FILE
```

Moreover, it the submitted file have not been registered first, the program will output:

```
SUBMITED HASH  QmRabQ1zTJ3Nm9pB9gvvvaLG65T1poeBfbRCG9HvJitSTW
Result {
  ’0’:
     ’77991722633929904953535340207214769289098226303112543 61’,
```

```
  '1':
     '5405757533067996537833169001063788056847307298302752837',
  h: '77991722633929904953535340207214769289098226303112543 61',
  h_: '5405757533067996537833169001063788056847307298302752837'
     }
REGISTERED HASH:  QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
[ ! ] YOU ARE **NOT** THE OWNER OF THE FILE
```

11.1.5 Webui

Lauching the web user interface

11.1.6 Vote of the deletion a file in the network.

If there is a pending deletion request of a file on the network, each node can vote of its deletion by typing:

```
$ odh --vote-for-deletion
   QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
```

11.1.7 Deletion Requests

Supply a request for the deletion of a file in the network.

```
$ odh --deletion-req
   QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
```

11.1.8 Full node request

Lauching the process of becoming a full node

```
$ odh --full-node-req
```

11.1.9 Light node request

Launching the hosting of a set of files

```
$ odh --light-node-req hashlist
```

11.1.10 Verify the existence of a file

Verifying files in the network

```
$ odh ––exist QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
```

11.1.11 Statistics

Display some statistics about files in the network

```
$ odh ––stats
```

11.1.12 Peers

Display the list of peer connected to the network

```
$ odh ––peers
```

11.1.13 Converting files

Convert a hosted file to different format according to .ext

```
$ odh ––convert QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
   out.pdf
```

11.1.14 Revoke the deletion request

It suspends the deletion request, reversed only to the one who have submitted the deletion request.

```
$ odh ––revoke–req
   QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
```

11.1.15 Fetch files

Write in the current directory the file hashed

```
$ odh ––fetch QmZ6vPS45mVqLMh4NNemPYY8pQf37AwkBeTooemDRSLNxE
```

11.1.16 Node id

Getting the identity of the node

```
$ odh ——id
```

11.1.17 Garbage collection

Remove unlinked or unpinned IPFS files

```
$ odh ——gc
```

11.1.18 reset

Remove configuration files

```
$ odh ——reset
```

11.1.19 List of files

List pinned files

```
$ odh ——list—files
```

12 Installation of ODH: installation and deployment of ODH

12.1 Ethereum

12.1.1 Ethereum Geth: go-ethereum installation

The installation of the geth client is pretty straightforward on linux machines: #### From source First clone the geth ethereum from github to a directory of your choice

```
git clone https://github.com/ethereum/go-ethereum.git
```

NB: You should have a working installation of the Golang compiler before trying to compile it yourself. Then:

```
cd go-ethereum
make all
sudo make install
```

12.1.2 From a private repository

On Ubuntu GNU/linux, we proceed by typing the following commands on your terminal:

```
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum
```

12.1.3 On Arch linux

```
sudo pacman -S go-ethereum
```

Then write the following file under the name genesisblock.json

```
{
    ”config”: {
        ”chainID”       : 10,
        ”homesteadBlock”: 0,
        ”eip155Block”:    0,
        ”eip158Block”:    0
    },
    ”nonce”: ”0x01”,
    ”difficulty”: ”0x20000”,
    ”mixhash”:
       ”0x00000000000000000000000000000000000000647572616c65787365646c6578”
    ”coinbase”: ”0x0000000000000000000000000000000000000000”,
    ”timestamp”: ”0x00”,
    ”parentHash”:
       ”0x0000000000000000000000000000000000000000000000000000000000000000”
    ”extraData”: ”0x00”,
    ”gasLimit”: ”0x8000000”,

    ”alloc”: {
    }
}
```

That genesisblock.json file will be used to create the first block of the blockchain and every node in our private network *must* use the same genesisblock in order to synchronize with each other and take part on the mining process. Unlike the ethereum main chain which uses a discovery mechanism of nodes through the internet, our implementation is compelled to use adhoc mechanism of nodes discovery. Thus the complete list of the ip address of each peers must be provider at connection. Right now, we explore others ways of nodes discovery. There are other requierements for the geth client is that, every node *must have the same networkid* and the -rcpport and -port *must* be different.

The each node must run the following commands:

```
geth ——datadir=”./” init genesisblock.json
```

after the user should see something like this:

```
INFO [07—26|17:29:26] Maximum peer count
   ETH=25 LES=0 total=25
INFO [07—26|17:29:26] Allocated cache and file handles
   database=/home/moktar/workspace/bc_stuff/s/geth/chaindata
   cache=16 handles=16
INFO [07—26|17:29:26] Writing custom genesis block
```

```
INFO [07—26|17:29:26] Persisted trie from memory database
   nodes=0 size=0.00B time=10.617µs gcnodes=0 gcsize=0.00B
   gctime=0s livenodes=1 livesize=0.00B
INFO [07—26|17:29:26] Successfully wrote genesis state
   database=chaindata
   hash…=98053617a77a
INFO [07—26|17:29:26] Allocated cache and file handles
   database=/home/moktar/workspace/bc_stuff/s/geth/lightchaindata
   cache=16 handles=16
INFO [07—26|17:29:26] Writing custom genesis block
INFO [07—26|17:29:26] Persisted trie from memory database
   nodes=0 size=0.00B time=10.966µs gcnodes=0 gcsize=0.00B
   gctime=0s livenodes=1 livesize=0.00B
INFO [07—26|17:29:26] Successfully wrote genesis state
   database=lightchaindata
                                          hash…=98053617a77a
```

After that the user should create a new account:

```
geth ——datadir=”./” account new
```

Then launch the node with this:

```
geth ——datadir=”./” ——networkid 23422 \
                    ——rpc \
                    ——rpccorsdomain=”*” \
                    ——rpcport=”8545” \
                    ——minerthreads=”1”
                    ——mine \
                    ——unlock 0  \
                    ——rpcapi=”db,eth,net,web3,personal,web3” \
                    ——port 30303 console
```

After performing that on each node we should now connect them to each other according to their ip address: On the console of each node type:

```
const fs = require(”fs”);
const nodes_addr = fs.readFileSync(”peers.txt”).split()

const nodesid = admin.nodeInfo.enode.split(’[::]’)[0] + ’@’ +
   ipaddr
nodes_addr.map(addr => admin.addPeer)
```

example: admin.addPeer(“enode://47e61e304d802fb98403fbf877e1018d13044630a16

12.2 IPFS

12.2.1 Installation of go-ipfs

The canonical to install go-ipfs is from its hosting website http://ipfs.io/docs/install/. However users of Arch linux, NixOS, snap supported linux distributions like Ubuntu or Debian can install them directly from their package manager. * Arch Linux In Arch Linux go-ipfs is available as go-ipfs package.

```
    $ sudo pacman —S go—ipfs
```

- Nix

```
$ nix—env —i ipfs
```

- Snap With snap, in any of the supported Linux distributions:

```
  $ sudo snap install ipfs
```

For experimented users interested in building by themselves go—ipfs, here are the steps: * Install the golang compiler

You’ll need to add Go’s bin directories to your $PATH environment variable e.g., by adding these lines to your /etc/profile (for a system-wide installation) or $HOME/.profile:

```
export PATH=$PATH:/usr/local/go/bin
export PATH=$PATH:$GOPATH/bin
```

Then download and Compile IPFS.

```
$ go get —u —d github.com/ipfs/go—ipfs

$ cd $GOPATH/src/github.com/ipfs/go—ipfs
$ make install
```

12.2.2 Setting up an ipfs node

To start using IPFS, the user must first initialize IPFS’s config files on your system, this is done with ipfs init. See ipfs init ——help for information on the optional arguments it takes. After initialization is complete, you can use ipfs mount, ipfs add and any of the other commands to explore! # The developer Annex: Installation of the development environment @@ Dependency graph

Programming languages: * Solidity (For smart contracts) * EVM bytecode (For smart contracts) * Javascript (For the cli-client and server implementation) ## Tools used * GNU Emacs (A highly extensible editor from the GNU project) * NodeJs (Google's V8-based Javascript interpreter) * Npm (node package manager) * Git (distributed revision control system) * Truffle (toolbox for the development of smart contracts) * Go-ipfs (A golang implementation of the IPFS protocol) * Geth (A golang implementation of the Ethereum protocol) * Web3 (Library for interacting with an ethereum client through RCP calls) * IPFS-API (Library for interacting with the ipfs daemon through Http request)

Like any *NodeJs* project, the ODH project is created with the npm (Node package manager) utility. It is essentially used to manage libraries and other dependencies of the project. If we take a look at dependency field of the projects configuration we see:

```
"dependencies": {
    "base58": "^1.0.1",
    "chalk": "^2.4.1",
    "co": "^4.6.0",
    "co-prompt": "^1.0.0",
    "commander": "^2.15.1",
    "express": "^4.16.3",
    "ipfs-api": "^22.1.0",
    "node-pandoc": "^0.3.0",
    "progress": "^2.0.0",
    "solc": "^0.4.24",
    "truffle-contract": "^3.0.6",
    "web3": "^1.0.0-beta.34"
  }
```

The projects is hosted on Gitlab and on Npm repositories, so we have to first download it.

```
$ git clone https://gitlab.com/alhasapi/odh-cli.git
```

Next, install the dependencies:

```
$ cd odh-cli
$ git checkout -b dev
$ npm install
```

Now we can start to develop and hack on ODH. <!- # Security, performance and future improvement ## The proof-of-work security issues [books: Mastering bitcoin and Security of Bitcoin and blockchain, Ethereum DOA case] ## Benchmarking blockchain synchronization and

IPFS peers finding ## Voting mechanism flaws and possible vulnerability exploitation ## Deletion and garbage-collection shortcomings ->

13 Conclusion and future work

We reached the end of our work, the questions and issues addressed here, concerns essentially design and implementation of a decentralized, peer-to-peer, open document sharing system. Throughout the document, the process of the design and the implementation of main features of the system were discussed. Designing decentralized apps yields, throughout its progress, the profound complexity of decentralized systems and its overwhelming heterogeneity. Dapps development is a challenge due to the lack of established rules, standards, and best practice. Our hope is that, it is the beginning of the switch or a leap to an entirely decentralized world of communication where there is no central authority behind the any operation concerning the management of personal data.

In a near future, we've planned to: - Re-implementation of ODH in Haskell - Switch to proof-of-stake based blockchains like (EOS or Tezos) - Implementation of a peer finding algorithm for auto-connection to peers on top of the IPFS SWARM protocol. - A formalization in Coq of all algorithms involved (peers finding, voting, synchronization) - Using program extraction methods to derive the program from the proofs. - Implementing a better Web UI, to make ODH user friendly and accessible for non technical people. - Decoupling components of ODH in order to make components of ODH reusable for other purpose (video sharing, NLP)

References

Alan A. A. Donovan, Brian W. Kernighan. 2015. *The Go Programming Language*. Addison-Wesley. http://www.gopl.io/.

Benet, Juan. 2014. “Ipfs-Content Addressed, Versioned, P2p File System.” *arXiv Preprint arXiv:1407.3561*. https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf.

Buterin, Vitalik. n.d. “Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.” {https://github.com/ethereum/
Paper}.

Conway, Melvin E. 1963. “Design of a Separable Transition-Diagram Compiler.” *Communications of the ACM* 6 (7). ACM: 396–408.

Cox, Russ, and William Josephson. 2005. “File Synchronization with Vector Time Pairs.”

Douceur, John R, Atul Adya, William J Bolosky, P Simon, and Marvin Theimer. 2002. “Reclaiming Space from Duplicate Files in a Serverless Distributed File System.” In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, 617–24. IEEE.

Lin, Hsiao-Ying, and Wen-Guey Tzeng. 2010. “A Secure Decentralized Erasure Code for Distributed Networked Storage.” *IEEE Transactions on Parallel and Distributed Systems* 21 (11). IEEE: 1586–94.

Martin, Keith M. 2012. *Everyday Cryptography*. Oxford University Press.

Maymounkov, Petar, and David Mazieres. 2002. “Kademlia: A Peer-to-Peer Information System Based on the Xor Metric.” In *International Workshop on Peer-to-Peer Systems*, 53–65. Springer.

Nakamoto, Satoshi. n.d. “Bitcoin: A peer-to-peer electronic cash system.” {http://www.bitcoin.org/bitcoin.pdf}.

Pike, Rob. 2009. “The Go Programming Language.” *Talk Given at Google’s Tech Talks*.

Quintard, Julien. 2012. “Towards a Worldwide Storage Infrastructure.” PhD thesis, University of Cambridge.