

MINISTRY OF NATIONAL EDUCATION



TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Data Reliability using Decentralized Systems

LICENSE THESIS

Graduate: Ionuț-Dan MATIȘ
Supervisor: S.L. Dr. Eng. Claudia Daniela POP

2020

MINISTRY OF NATIONAL EDUCATION



TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DEAN,
Prof. dr. eng. Liviu MICLEA

HEAD OF DEPARTMENT,
Prof. dr. eng. Rodica POTOLEA

Graduate: **Ionuț-Dan MATIȘ**

Data Reliability using Decentralized Systems

1. **Project proposal:** *This project aims to solve data reliability in a decentralized way, using two decentralized systems: Ethereum blockchain together with the Inter Planetary File System (IPFS) data storage.*
2. **Project contents:** *Introduction and project context, objectives of the project, bibliographic research, theoretical foundation, implementation, testing and validation, the user's manual and the conclusions.*
3. **Place of documentation:** Technical University of Cluj-Napoca, Computer Science Department
4. **Date of issue of the proposal:** November, 2019
5. **Date of delivery:** July, 2020

Graduate: **Matiș Ionuț-Dan**

Supervisor: **Pop Claudia Daniela**



FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

**Declarație pe proprie răspundere privind
autenticitatea lucrării de licență**

Subsemnatul(a) **Matiș Ionuț-Dan**, legitimat(ă) cu **CI** seria **CJ** nr. **144532** CNP **1970624125834**, autorul lucrării **DRDS – Data Reliability using Decentralized Systems** elaborată în vederea susținerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea **Calculatoare Engleză** din cadrul Universității Tehnice din Cluj-Napoca, sesiunea **iulie** a anului universitar **2020**, declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării și în bibliografie.

Declar, că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de licență*.

Data
07.07.2020

Nume, Prenume
Matiș Ionuț-Dan

Semnătura

Contents

Chapter 1	Introduction - Project Context	1
1.1	Project context	1
1.2	Decentralized systems	1
1.3	Challenges	2
1.4	Thesis structure	2
Chapter 2	Project Objectives and Requirements	3
2.1	Main objectives	3
2.2	Secondary objectives	3
2.3	Challenges	4
2.3.1	Link Ethereum with IPFS	4
2.3.2	Understand and improve IPFS	4
2.3.3	Minimize the operation cost on Ethereum	4
2.4	Functional Requirements	4
2.4.1	Allow users to upload data	4
2.4.2	Allow users to version data	5
2.4.3	Detect plagiarism	5
2.4.4	Detect data corruption	5
2.5	Non-functional Requirements	5
2.5.1	Visual Application	5
2.5.2	Usability	6
2.5.3	Performance	6
2.5.4	Extensibility	6
2.5.5	Security	6
2.6	Metrics used for evaluation	6
2.6.1	The price of storing data	6
Chapter 3	Bibliographic research	7
3.1	Regular storage	7
3.1.1	Centralized databases	8
3.1.2	Distributed databases	8
3.1.3	Cloud databases	9

3.2	Decentralized storage	9
3.3	Blockchain	12
3.4	Hybrid systems	14
Chapter 4	Analysis and Theoretical Foundation	16
4.1	Bitcoin blockchain	16
4.2	Ethereum blockchain	20
4.3	IPFS	29
Chapter 5	Detailed Design and Implementation	36
5.1	System diagram	36
5.2	Technologies used	38
5.3	Component diagram	40
Chapter 6	Testing and Validation	51
6.1	Description of data	51
6.2	Metrics	52
6.3	Experimental results	52
Chapter 7	User's manual	55
7.1	System requirements	55
7.2	Installation steps	55
7.3	Application start steps	56
7.4	User manual	56
Chapter 8	Conclusions	58
8.1	Addressed problem	58
8.2	Solution and personal contribution	58
8.3	Results	59
8.4	Further improvements	59
Bibliography		60

Chapter 1

Introduction - Project Context

1.1 Project context

Over the last few decades, the Internet infrastructure and size has increased dramatically and so has done the amount of information which was created and processed. Nowadays, almost every person has access to Internet and can inform himself or herself about various topics or read the latest news. In addition, every one can produce information and through the aid of the Internet and social media it will propagate at a rapid pace, whether it is good, bad, true or false. This propagation is done together with altering and the user may receive a corrupt piece of data. We can see this nowadays, as fake news seem to become a real issue.

However, some questions have arisen: how can someone validate the integrity of the information he or she is reading? How can one be sure that a certain person actually said a certain thing? How can you trace back the creator of the information without spending a large amount of time and resources?

In order to determine the reliability of some data, one can read from multiple sources, but this can become tedious and time consuming. This project (DRDS) tries to answer the questions raised above by using a decentralized solution, combining two existing systems: Ethereum and IPFS.

1.2 Decentralized systems

In the past decade an interesting technology has evolved and that is decentralized systems. It is consider somewhat revolutionary because this system does not have a central authority which takes the decisions, but it works based on a set of rules (called consensus) and the peers in the system are equal.

One of these decentralized systems is called Ethereum blockchain and has a unique property making the system transparent. Every transaction is available for inspection and the information within cannot be erased. The problem related to traceability is solved.

Unfortunately, you cannot store a lot of information in blockchain, as it costs a lot.

Fortunately, the blockchain is not the only existent decentralized system. There is another, named IPFS that aims to solve the issue that the blockchain has - the storage. It manages to do this by replicating only the needed information. Moreover, this data is content located using cryptography. This is of help when we check for data reliability.

1.3 Challenges

In developing the DRDS application some challenges will appear.

1. **technology maturity:** the technologies (Etheruem and IPFS) are rather new and they are developed at a high pace. A version of one may become obsolete or deprecated in months time and new versions usually create breaking changes.
2. **system integration:** even though both systems are decentralized, they are not not related to one another and we need gateway to allow them to communicate and work together
3. **visual project:** the final application needs to be easy to use and intuitive so that the final user has a pleasurable experience.

1.4 Thesis structure

1. **chapter 2:** the main requirements, objectives and features that will be present in this project
2. **chapter 3:** showcases the studies that I performed in order to be able to come with a solution to the previous stated issues
3. **chapter 4:** will show the system details, how they function and what makes them suitable for this project
4. **chapter 5:** will present the implementation of the solution together with the explanation required to understand it. Moreover, it will contain the system diagram and the interaction between the modules
5. **chapter 6:** will provide the results that I have obtained after the development of the application together with validations upon them
6. **chapter 7:** will explain in technical terms how to install, configure and start the application so that it can be used
7. **chapter 8:** will present the conclusions of the project together with future implementation that could be done to it

Chapter 2

Project Objectives and Requirements

2.1 Main objectives

This project has the purpose of studying, designing and implementing a solution for detecting data plagiarism and data corruption while providing the ability to version (edit) already existing data using and extending the features of two widely known decentralized systems: Ethereum and IPFS.

2.2 Secondary objectives

The below table presents the objectives that are secondary. Each row contains a brief description of the objective, and also a reference to the appropriate section in the thesis where that objective is discussed in more detail.

Table 2.1: Secondary objectives

Nr.	Objective	Description	Reference
1	Study of already existing approaches	Learn how decentralized storage is achieved	Chapter 3
2	Detail the two chosen systems	Blockchain and IPFS technologies will be explained	Chapter 4
3	Provide the implementation	Each module of the application will be explain from a top down approach	Chapter 5
4	Evaluate the solution	The implemented system will be evaluated according to the metrics	Chapter 6

2.3 Challenges

This section highlights the challenges that may appear in the creation of this project. While designing and developing the system, these should be taken into account.

2.3.1 Link Ethereum with IPFS

The first challenge that needs to be faced is to find a way to allow the two decentralized systems to communicate. Currently, both have their own independent API (application program interface) with which the user can interact. This project should gather both APIs and allow them to communicate freely, transferring information from one another.

2.3.2 Understand and improve IPFS

One of the main goals of this application is to detect if data was corrupted. This will be done on top of the already existing IPFS system. The first step that needs to be done is to understand it at its core and then find a way to improve its data corruption detection.

2.3.3 Minimize the operation cost on Ethereum

Every interaction with the Smart Contract deployed on Ethereum blockchain that adds or modifies data costs virtual currency, depending on the complexity of it. We need to minimize it as much as possible, so that the users do not consume their money unnecessarily. For example, let us say that we have a list of all uploaded files. If we perform an iteration over it every time we add a file, the price of interactions will grow constantly and it is not desirable.

2.4 Functional Requirements

This section presents the functional requirements of the project, i.e. what the system should do. By this, the basic behavior of the application can be understood.

2.4.1 Allow users to upload data

The main feature that this system provides is to allow users to upload data (any kind of file) in the system and then be able to see a list of all its uploads. However, there are some rules that the users should not break (they will be discussed below) when doing this action. Moreover, they should be able to see various details about the files, such as its actual size when stored on a node.

2.4.2 Allow users to version data

As data may change over time, users should be able to edit their uploaded files and bringing them up to date. Only the authors (owner) of the original files can alter them and no other users should be capable of doing it.

The system should detect if a user tries to edit the files of another other and forbid such an action.

2.4.3 Detect plagiarism

The system should be able to detect plagiarism. This means that a file can be uploaded to the system only one time. Neither the original user that first uploaded the file nor other users should be able to add or upload via edit (using the versioning capabilities) an already existing file.

The system should detect any such attempts and warn the user about the invalid action that he tried to perform.

2.4.4 Detect data corruption

The data the user uploads in the application will not be stored locally, but in an unknown location (based on some predefined rules), as a decentralized system will be used for storage. By this, a malicious node that holds the user's data can try to modify it, receiving in the end an advantage from this action.

The system should be able to identify any attempts of data modifications and prompt the original user (the one that owns the original data) that there have been alterations to the file and it should not be trusted.

2.5 Non-functional Requirements

These requirements define how a system should behave, not what it should do. This specifies the quality attribute of the whole project.

2.5.1 Visual Application

The application UI will be suggestive to the user. The marks that will show if certain files are valid or not will be green and respectively red, depending on the state of the information tested by DRDS system.

2.5.2 Usability

The application will run on a web browser. It will be easy to use and intuitive. The user will have buttons to interact with the application: for choosing the file, for uploading the file, for seeing the details related to the file and for editing it. Moreover, the user will see if the files he uploaded are still valid on the nodes.

2.5.3 Performance

The application performance will be measured in:

- the time the system needs to validate a file, depending on its size
- the accuracy of detecting the reliability of data

2.5.4 Extensibility

The application should be able to accommodate more functionality as new features will be added. Moreover, no major changes should be made when doing this.

2.5.5 Security

The application will be secure, as its systems are designed to be so. In this way, there is no central authority that needs to be trusted. Moreover, a large part of the system uses cryptography as its core.

2.6 Metrics used for evaluation

This section presents the metrics used to evaluate the project.

2.6.1 The price of storing data

The price of storing data will be the metric that should be computed. We need to calculate how much would cost to store all data on Ethereum blockchain and then compare it to the cost of storing it in IPFS.

Chapter 3

Bibliographic research

This chapter presents three types of solutions in the field of data storage together with their ways of assuring that the corresponding data remains reliable, i.e. it is not corrupted or altered in malicious ways so that an attacker can profit upon this. Moreover, the versioning of this data should be considered, as nowadays the information keeps updating at a high pace.

3.1 Regular storage

Nowadays, most of the data is stored on databases, which, at a high level, are defined as a software tool that allows the users to perform CRUD (Create Read Update Delete) operations. Usually, when we talk about databases, we refer to the SQL / Relational ones, which present data as tables with rows and columns.

This type of storage is mature, as it has been used almost for the past half of the decade. This maturity leads to well-written documentations, well-defined, widely-accepted standards and last, but not least, a vast area of usage examples so that any type of knowledge level can still learn from them.

Viewing things from other perspective, the one that controls the database or has the highest privilege can alter the data in any way he desires. In this way, we need to trust an external peer and this is not desirable in our case. Thus, we cannot totally assure the reliability of the data. This may become problematic in case important information must be kept in such a system.

The versioning feature can be easily implemented in various ways. Two of them would be inserting the new version of the data as a new entry instead of updating the old one or use a logging system that will store events when there is change. In this way, we would traverse the logs in order to get a specific version.

3.1.1 Centralized databases

The information is located and maintained in a single zone and various users access it from different locations. In terms of data reliability, the system may impose authentication steps and roles for the users. By this, only certain, high-privilege users can perform important modifications of the system. [1]

In terms of advantages, the centralized database minimizes redundancy of information and increases integrity of data, as we need to manage it only on one location. [2] It is often easier to use and maintain, as one does not need to build an entire architecture and synchronize the data.

In terms of disadvantages, a central database may represent a bottleneck if the traffic to it is too high and also constitutes a single point of failure. If the database is compromised, the whole system will go down. Without a back-up, all data would be lost. Another important factor refers to the location of the database and who will access it. If it is meant to be accessible worldwide, there must be positioned in a central spot, to decrease the latency that would be caused by long distances.

3.1.2 Distributed databases

The information is split among different servers or among various locations. [3] The information within those databases may be replicated (there will be copies of the same data in multiple places for back-up or redundancy purposes) using a specialized software that keeps track of changes, duplicated (there exists a master database from which information is copied to the slave databases) or completely different among the system. In the last case, we would need to know where certain parts of the information are stored so that we would not query all sub-parts.

To showcase the positive aspects, the availability and reliability is high.. In case of an unfortunate event, information is not destroyed completely, as it is distributed among various places. [3] Moreover, the system is modular, as new elements can be added or removed without affecting or break the whole system. This is very important as the application grows.

In terms of disadvantages, the complexity is much higher than the one in centralized databases, and the database managers or admins would require more experience and would have to work more in order to keep the system functional. There will also be a need for a specialized software that manages the whole system, as there are more parts that should be handled and the structure may be built upon several levels of hierarchy.

3.1.3 Cloud databases

In the last years, this types of databases have emerged and gained popularity in the technological world. You interact with the database as a free or paid service. [4] This means that the user is not involved in the system architecture and does not need to bother with the scalability and availability. The company that owns the service is responsible for managing the hardware and software of the database in exchange for the money of the users. [4]

In terms of advantages, the complexity of the system is transparent to the user and it should not concern him. Scalability is increased, as most providers aim to develop systems that meet any customer needs. They usually provide it by adding more nodes that cooperate by sending data between them. The performance may be increased as well, as the systems are developed in such a way to be efficient. In addition, the system may be more secure, as there are certain standards that must be met in order to provide such a service.

In terms of drawbacks, the pricing of such a service may be expensive. However, this depends on the scale at which the service is used and which service provider the user chooses to work with. There are already a few competitors on the market and the prices may be in the favor of the customer. Even though these cloud services are up at most times, a downtime of the servers could happen and you would not be able to use your database. However, these issue happens rarely and it is solved quickly.

3.2 Decentralized storage

This type of storage is characterized by a decentralized network which is spread across a variety of locations and where individual users or groups store data independently. They do so because there is usually a method that rewards them if they keep the data they offered to store available at most times. By this, the servers are not owned by a single company, but by each and every one of the members. Usually, anyone is free to join the decentralized system, but the old members have an advantage, as they are considered more reliable.

The actual data can be retrieved in a conventional fashion, i.e. request it and receive it. Depending on the actual system, the client data can be encrypted, as the ones that have access to the storage node cannot read it. Besides that, the uploaded file can be split into multiple parts, each part being stored in a different place, depending on a system rule. By this, an additional security layer is added and potential data leakage is diminished.

IPFS (InterPlanetary File System) [5]

This system provides a protocol to create an ingenious way to search information on internet. Currently, if you want to find an article on the web, you basically need the link (location) of that information. In the background, based on that link, the IP address of the server is received and then queried for that specific data. But there is a problem in this system. What happens if the server stops or the file is moved to other location? Basically, you cannot access that article anymore.

Instead, IPFS serves the files based on their content, not based on their location. [6] By this, the previous article could be served by another user that accessed it in the past, as they would store a copy of it. You can also be sure that is the original file, because each file is given a unique cryptographic hash. Any alterations of it would automatically change its previously generated “fingerprint”.

Regarding data reliability, the cryptographic hash makes the system tamper resistant. Moreover, IPFS provides a way of versioning files by using IPNS (Inter- Planetary Name System). With this, the user can create a stable link that will always refer to the latest version of a file.

Storj [7]

This system can be seen as a decentralized cloud created as an alternative to already existent services like Dropbox. It is similar to torrent, as data is split into blocks, encrypted and the shared among the network participants, which are called farmers. The system design provides security and reliability. The farmers basically share their unused hardware space to users that want to store data on the system. They get in return a virtual currency called SJCX.

A special feature that this system introduces is the proof that a file is retrievable. This verifies that the parts that every farmer host can actually be accessed by the clients when they desire. Storj system requires its farmers proofs that they are online and they still have the data they claim to do so. It does this by sending them puzzles that could be solved only when certain data (that is requested) is stored by the specific farmer. [7] This feature can make the system the most reliable one, as it forces each node to stay online as much time as possible. Ultimately, the online presence of all nodes makes the whole system usable even in production environments.

Unfortunately, there is no available information regarding the ways of versioning the information. The user will probably need to keep track of the files and their order or upload. This can get tedious if a bigger application is built onto Storj.

Swarm [8]

Ethereum system, which will be discussed in the future chapter, is composed of three core elements: the contracts, which represent the decentralized logic, the swarm, which represent the decentralized storage and whisper, which represent the decentralized messaging protocol. Therefore, Swarm is the so called database of the Ethereum idea of a fully decentralized web. However, it is completely independent and can be used as a single system all by its own.

Swarm, as the above system, is a decentralized storage solution. It is a system that was built together with Ethereum blockchain and is a part of a whole. It has as features the following: resistance to DDOS, peer 2 peer database and includes an incentive methodology. [9]

Swarm has two major features that set it apart from other decentralized distributed storage solutions. While other systems allow you to run your own node and host data on it, Swarm provides the hosting itself as a decentralized cloud storage service. By this, the "upload and disappear" practice is allowed, as the user can upload its content and then retrieve it later.

The other feature is the way it incentivizes the participants that offer their storage and bandwidth resources in order to provide global services to all participants that want to use the system.

Sia [10]

Along with the previous systems, this one is, as well, a decentralized data storage platform. The main idea is the same, it connects the users which have underutilized hard drive capacity (called hosts) to customers that search for data storage (called renters). The system is built on top of blockchain, which adds additional security and even costs less than cloud competitors. [11]

Besides the low prices that they offer (two times less than the closest competitor on the market), Sia also provides strong data security. It does this by Reed-Solomon Redundancy, which stores data on 30 devices located around the world. From that number, only 10 need to be online in order to store a file and if all hosts have a median reliability of 90%, the chances of data lost are extremely small.

Moreover, Sia provides encryption and distributes renter's files around the world across a decentralized network. The files are encrypted on hosts machines, each separate piece of data having a separate password so it can't be deduplicated. Besides this, to hold hosts accountable, they are monitored often to make sure they are indeed providing the files they promised to keep available. They are checked upon a number of variables and their reliability is around 90-95%.

3.3 Blockchain

The blockchain technology is based on blocks of information that are linked together, forming a chain, from where its name comes from. This system is completely decentralized, as each node that runs the software has its own copy of all the blocks. Depending on the blockchain type (as there exists multiple implementation of it), the information that can be stored in such a block is limited (1 Mega Byte in Bitcoin blockchain) or costs too much if the amount of data needed to be stored increases, in the case of the Ethereum blockchain.

What is special about this type of storage is the fact that it is immutable. Once data has been set into place in such a system, it can never be erased or updated, but only read. In terms of information versioning, not having an update feature may seem problematic, but it is not actually the case. We can add new information and consider it the most recent version of a previously uploaded data. By this, going through each link, we can see the whole history, as it is available to the large public.

Data and information integrity appear by default in such systems, as cryptographic hashes form the basis of them. If something changes in a block, its hash will change as well and this will propagate to the most recent node, invalidating all blocks in the path.

Storing data in Bitcoin blockchain

The Bitcoin blockchain was not designed to store data and was meant to handle just transactions. However, some users have found ways to store other types of data, not only transactions.

Historically, the first time information was stored on a blockchain was at its beginnings, when Satoshi Nakamoto (the creator of the Bitcoin blockchain) inserted a secret message in the first block (also called genesis block) in which he wrote “The Times 03/Jan/2009 Chancellor on brink of second bailout for banks”, this being a title from an article in The Times newspaper. We can draw at least two conclusions from this message. Firstly, it proves that the first block was mined after the beginning of 2009 and, secondly, the fact the new system was a protest against the way banks work, as it occurred after the 2008 financial crisis. Thus, Bitcoin was created as an alternative to the bank systems at that time.

One way of storing data is to insert it inside a coinbase transaction. The Bitcoin blockchain has a special transaction that is responsible for generating new bitcoins. This has a field which represents an address to which to send the created virtual currency, but the miner can add anything he wants there. [12] This type of transaction is inserted by the miner who mines the block.

Another way of storing data is to split it and encode it into fake bitcoin addresses. If these addresses are after encoded to hex and then attached to one another, a file or even image can be formed. [12] By using this technique, the bitcoin sent to that fake address are lost forever, as you cannot find the private key that would generate your custom message.

In 2013, the Bitcoin protocol was updated and a new type of transactions has been

added. This was able due to activating the OP_RETURN parameter and I will allow users to add 80 bytes of data. [13] Even though 80 bytes represent a small amount of information, it could be used to store a hash which could have a specific meaning such as an ownership of a car or attest that something happened at a specific point in time. For example, the transaction

8bae12b5f4c088d940733dcd1455efc6a3a69cf9340e17a981286d3778615684

It contains a hidden message that states the following “charley loves heidi”.

Some users found another location in a transaction where to store not just data, but code that can be run in the browser (javascript). The location is in the transaction output script, an area where the creator of the transaction specifies the steps required by the receiver so that the virtual currency can be further spent. Such an example is in the transaction

59bd7b2cff5da929581fc9fef31a2fba14508f1477e366befb1eb42a8810a000

The script would just display a message.

[12] . If the site would not escape the data as HTML, the previous piece of code would just pop an alert box. However, a transaction that aims to steal virtual currency could be possible.

Storing data in Ethereum blockchain

Unlike Bitcoin system, Ethereum is a blockchain that allows its users to store more than just transactions into it. Users can store actual data that can be retrived lately and they do not need to find hacks in the system to store tiny amounts of data. However, information stored on chain is expensive because it needs to be downloaded and stored on every node that runs the blockchain implementation. By this, the real size “on disk” of your uploaded data is the initial size multiplied by the number of nodes that downloaded it. At the time of writing, there are more than 5000 active nodes that maintain in the Ethereum network.

When we want to upload information in the blockchain, we pay for the following things: the price of the transaction, the price per byte of data and, if smart contracts are involved, we pay for the execution of the code in the smart contract. There are two known, widely accepted ways, of storing data on this system.

Firstly, data can be stored in a special place reserved for transaction input. When a user sends a transaction, besides the amount of virtual currency he wants to send, the address of the receiver and some parameters, he can add data to it. Technically, there is no limit to the amount of data, but the input consumes gas. The maximum amount of gas that can be consumed in a single block is 9 000 000, as of May 2020. Every 0 byte that is stored consumes 4 gas and every non-zero byte consumes 68 gas. When a user wants to send a transaction, it has to pay the base price of 21000 gas. From the maximum amount of gas and how much gas a byte takes, we can deduce the maximum amount of data that can be transferred. [14] We also need to consider the time in which the transaction(s) will be processed. If we decide to split the data in smaller chunks, we need to wait for each

transaction to processed by the Ethereum systems.

When a user wants to send a transaction, it has to pay the base price of 21000 gas. From the maximum amount of gas and how much gas a byte takes, we can deduce the maximum amount of data that can be transferred.

The second way to store data is to include it into a special space in blockchain reserved for Smart Contract data. The smart contract contains code that can be executed. You can think of it as another type of user, as it has a unique address, can send or receive virtual currency, but it also able to perform logical decision and behave in a way that was programmed by the author of the smart contract. The contract has specialized structures for data storage called types. At a basic level, one can store boolean, byte, integer and string types inside it. At a more advanced level, arrays and mapping can be used to customize the logic and structs to organize the way data is related. Moreover, inside the contract, you can trigger custom events that can contain data as well.

3.4 Hybrid systems

We have seen that the previously discussed solutions to our problem have both advantages and disadvantages. What we would like is something that maximizes the number of the positive effects and it can be done by combining the 2 types of systems. We can use the blockchain to store a data link or a unique identifier that leads to the actual data. In this way, a part of the tamper proof issue is solved, as data in the blockchain cannot be modified.

Related to authenticity of data, Acronis Notary system, described in [15], is a blockchain service which provides a universal solution for timestamping and fingerprinting any data objects and streams. Storing large data on blockchain is a very expensive, therefore their solution is to send the files to their system. After this, it produces a new hash based on the received ones. This last is the actual one that gets stored on the blockchain. A verification certificate is provided specifying the technical details of the document. Any time the document is displayed in UI, it is shown as a mark created by Acronis. By this, the system presents the user that his file is the same as the first one, at byte level. Their system also provides proof of integrity, which demonstrates that the data has not been corrupted.

Related to the combination of the two decentralized systems (Ethereum and IPFS), a team of three from the University of Abu Dhabi created an application to provide originality of various data. Their application uses Ethereum and IPFS in the same manner proposed by this project, i.e. IPFS is used for file storage with high availability whereas Ethereum is used to store hashes and control the interaction. [16] They also offer traceability into the history of data and different versions of it. In the paper, their solution is focused on online book publication, including the possibility of book translations. However, they do not specify any validations of the received data, assuming that the information retrieved from IPFS was not tampered.

The paper "Blockchainbased Database to Ensure Data Integrity in Cloud Computing Environments" [17] proposes a blockchain solution to ensure high data integrity which does not affect performance and stability. Their system is composed of two layers that provide different, but complementary behaviors. The first one ensures a distributed consensus that is lightweight and allows low latency and high throughput. The aim of this is to allow fast, reliable storing of evidences of all actions that happened inside the storage. Unfortunately, the first layer provides no reliability, as it is not verified. The second layer is designed to solve the data integrity issue which was not covered by the first layer. Its role is to store logged actions generated by the first layer. What is lacks is the performance, as a PoW needs to be done. However, they state that the two layers provide a good overall performance.

Chapter 4

Analysis and Theoretical Foundation

The purpose of this chapter is to explain the operating principles of the implemented application. This chapter will be split in two main parts. In the first part I will explain the blockchain technology and in the second part I will offer details about how IPFS works.

4.1 Bitcoin blockchain

When talking about blockchain, we must firstly refer to its original implementation - Bitcoin Blockchain - created by an individual (or more individuals) named Satoshi Nakamoto. Its (or their) system changed to an extend the world we live in, because a new type of currency (a digital one) was created. What was so special about it is the fact that there is no need for central bank or entity when exchanges of such currency are made.

An interesting analogy

One can think of blockchain by thinking about the following simple analogy. Let us say we want to build a wall, but no ordinary wall, a special one, as only one brick can be added at a certain level (yes, we build a wall of width equal to one brick).

Each brick is composed of special materials and we need to wait a fixed amount of time before the brick is ready to be put on the wall.

As the wall grows (in height), more and more bricks are added on top of each other. What makes the wall stable is the fact that every two bricks are glued together with a special adhesive let us say. If somehow the bricks are not glued very well and someone tries to take out a brick of this structure, the whole wall collapses, as it cannot stand on thin air.

There are also workers that lay the bricks one onto another, but not before inspecting them and making sure that they respect the brick norms. They are paid for each block they add on the wall.

Finally, what is really special about this wall is the fact that it is built simultaneously at different built sites, but respecting the previous mentioned rules.

Finally, to clear things up and speak in technical terms:

- **the brick:** a single block within blockchain
- **the brick materials:** the header of the block together with the transactions within the block
- **the fixed amount time before the brick is ready:** the estimated time at which a block is generated
- **the wall:** the blockchain structure
- **the glue:** the cryptographic hash that links one block to the previous one. Any change in the previous blocks invalidates this hash.
- **the workers:** the miners that add new blocks to the blockchain
- **the reward of workers:** the amount of virtual currency a miner gets for a block that he mines successfully
- **the build site:** a single node that runs the blockchain protocol. Each such site has an exact copy of the blockchain

What is a blockchain?

A blockchain is a continuously growing data structure that uses cryptography extensively to link groups of data called transactions in a chain manner, providing decentralization. [18]

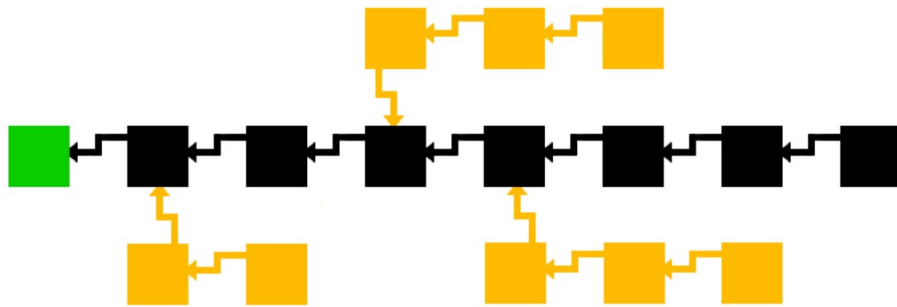


Figure 4.1: The basic structure of a blockchain

In the above image the following data is displayed:

- **Square:** represents a single block on the blockchain. These are built on top of each other, starting from the genesis block.

- **Arrow:** represents the link to the previous block. In implementation, this link is actually a cryptographic hash and it is stored on the actual block from which the arrow starts to point
- **Green square:** genesis block, the first block on the chain, which also contains the first state on the system and from which all begins
- **Yellow square:** represents a block from an orphan chain. The data within is not considered valid
- **Black square:** represents a block from the main chain. The data within it is considered valid when there are at least six blocks on top of it
- **Yellow chains:** represents orphan chains that exist outside the main chain. They appear in case two miners mine a valid block at the same time or if someone is trying to attack the blockchain
- **Black chain:** represents the main chain. The main rule is that the longest chain is considered the reliable one.

What is a block?

A block within the blockchain contains transactions to be added in the system. This block is composed of two subsections: the header and the body. The first is much more smaller than the second, containing only metadata. [19]

Table 4.1: Block structure [19]

Size	Field	Description
4 bytes	Block Size	The size of the block, in bytes, following this field
80 bytes	Block Header	Several fields form the block header
1–9 bytes	Transaction Counter	How many transactions follow
Variable	Transactions	The transactions recorded in this block

The block header consists of three sets of metadata as can be seen in the below table. The three sets are separated by double horizontal lines.

Table 4.2: Block header structure [19]

Size	Field	Description
4 bytes	Version	A version number to track software upgrades
32 bytes	Previous Block Hash	A reference to the hash of the previous block
32 bytes	Merkle Root	A hash of the root of the merkle
4 bytes	Timestamp	The approximate creation time of this block
4 bytes	Difficulty Target	The Proof-of-Work algorithm difficulty target
4 bytes	Nonce	A counter used for the Proof-of-Work algorithm

What is a transaction?

Transactions are considered the most important part of the bitcoin blockchain system, as everything is built around them. Through them virtual currency can be transferred from one individual to another. Each such transaction is composed of two parts: transaction input and transaction output.

Table 4.3: Transaction output structure [19]

Size	Field	Description
8 bytes	Amount	Bitcoin value in satoshis (10 ⁻⁸ bitcoin)
1–9 bytes	Locking-Script Size	Locking-Script length in bytes, to follow
Variable	Locking-Script	A script defining the conditions needed to spend the output

Table 4.4: Transaction input structure [19]

Size	Field	Description
32 bytes	Transaction Hash	Pointer to the transaction containing the UTXO to be spent
4 bytes	Output Index	The index number of the UTXO to be spent; first one is 0
1–9 bytes	Unlocking-Script Size	Unlocking-Script length in bytes, to follow
Variable	Unlocking-Script	A script that fulfills the conditions of the UTXO locking script
4 bytes	Sequence Number	Used for locktime or disabled (0xFFFFFFFF)

4.2 Ethereum blockchain

Ethereum is another form of blockchain, a second version to say so, which operates as an OS which contain smart contracts (with code that can be run / called). It works by changing the state of the EVM (Ethereum Virtual Machine). [20] By this, Ethereum has a globally accessible singleton state which changes with each transaction that is being processed.

As in case of the Bitcoin blockchain, Ethereum blockchain also has a cryptocurrency called **ether** which is generated as a reward to the miners that keep the system alive. The validity of this cryptocurrency is provided by the blockchain, which has the same basic structure as the Bitcoin, being a continuous expanding list of blocks which are linked using cryptography. By this, the structure is resistant to data modifications and open to verification.

Unlike Bitcoin, Ethereum provides a decentralized virtual machine called EVM (Ethereum Virtual Machine) which can basically execute code and is Turing Complete, meaning that any algorithm logic can be constructed with this system. *Ethereum operates using accounts and balances in a manner called state transitions. This does not rely upon unspent transaction outputs (UTXOs) as Bitcoin does. The state denotes the current balances of all accounts and extra data. The state is not stored on the blockchain, it is stored in a separate Merkle Patricia tree.*

The following table presents more differences:

Table 4.5: Bitcoin - Ethereum differences [20]

Property	Bitcoin	Ethereum
Block generation time	10 minutes	10-15 seconds
Amount of currency generated at each block	Halves every 4 years	Consistent rate, changes during hard forks
Algorithm for block verification	Bitcoin Proof-of-Work	Ethereum Proof-of-work (Ethash)
Transaction fees	Determined by transaction size	Determined by computation complexity
Accounting system	UTXO	Debit - credit
Address verification	base58check is used	any Keccak-256 hash is seen as valid

Cryptography in Ethereum

Blockchain technologies have at their core **cryptography** (besides other important technologies). This technology is very important and widely used in computer security. Cryptography is a vast area of science that uses mathematics in order to perform the following: secret writing of data between two peers in an unintelligible way so that a third party cannot read it, creating unique fingerprints (called hashes) so data can be referenced by a certain string and much more. [21]

Ethereum allows creation of data structures that are considered accounts. The first type, externally owned accounts (EOAs), are the ones that are own by one or more individuals which are real, humans. The other type are the contract accounts. They cannot be owned, because they own themselves through the code that has been written into them. It is important to note that the EOA that deploys the contract does not own it. He is just an aide so to speak.

EOAs are owned by the individual(s) that have their corresponding private key. The private keys are at the core of Ethereum's interactions performed by users. In fact, the account is directly derived from this key. By this, private keys are very important and they should be not transmitted, shown or stored in any way. If you do so, it is similar to allowing everyone to use your funds. The control the virtual currency is established with digital signatures which also involve the usage of the private key.

Public key cryptography

In this type of cryptography (also called asymmetric cryptography), keys come in pairs: private (secret) key and public key. Using an analogy, the public key can represent the physical card and the private key can be the pin which is needed for money retrieval. [21] We say that they come in pairs because the public key is derived from the private one.

Private keys

A private key is just a number, picked at random. This number needs to be 256 bits long. One could pick such a number by tossing a coin 256 times and writing the results on a paper. This, in fact, means that we need to pick a number between 1 and 2^{256} . However, this pick must be totally random and not predictable or deterministic.

The private key space size of Ethereum is extremely large, comparable to the number of atoms in the universe, which is 2^{80} . [21]

Public keys

A public key in Ethereum represents a point on a specific curve (elliptic curve). This point has both X and Y coordinates and, therefore, the public key is actual two numbers

joined together. These points satisfy the elliptic curve equation. X and Y coordinates are created from the private key by a calculation that can be done only from one way. This means that finding the private key from public key is impossible.

Formula: $\mathbf{K} = \mathbf{k} * \mathbf{G}$, where K is the public key, k is the private key, G is the generator point and * is the elliptic curve multiplication

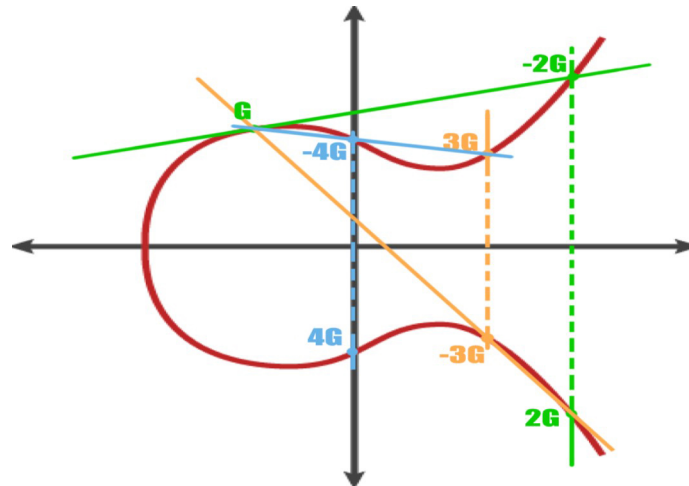


Figure 4.2: Elliptic curve cryptography point generation

Explanation for the previous image:

- **red line**: represents the elliptic curve function of type $y^2 = x^3 + ax + b$, where a and b are chosen so that the function is cryptographically secure
- **G**: the generator point, the starting point from which the calculations begin
- **green solid line**: tangent of elliptic function in point G
- **-2G**: the point where the tangent intersects the elliptic function
- **2G**: symmetric of -2G relative to OX axis
- **-3G**: intersection between line formed by G and 2G and elliptic function
- **Final point kG**: point which is obtain after adding G to itself k times. kG will have two coordinates X_{kG} and Y_{kG} from which the public key K will be formed

Cryptographic Hash Functions and Ethereum addresses

Besides public key cryptography, Ethereum also uses cryptographic hash functions. They are responsible for creating an address from a public key previously discussed. To make it less complicated, a hash function transforms any kind of data of any size into

a fixed output size. There are several such functions and Ethereum uses one to provide security to its platform. [21]

A cryptographic hash functions maps arbitrary size data to a fixed size output. This mapping is one way, as you cannot inverse the function, you cannot find the input if you are provided the output. Ethereum uses the cryptographic hash named Keccak, on 256 bits.

Properties of hash functions:

- **Determinism:** the same message produces the same hash output
- **Verifiability:** it is easy (not computational complex) to verify the hash of an input
- **Noncorrelation:** a minor change in the input message changes the output so much that it cannot be correlated to the initial input
- **Irreversibility:** the only way to find the input based on the output is to search through all possible inputs
- **Collision protection:** it should be very hard to find two messages that give the same hash output

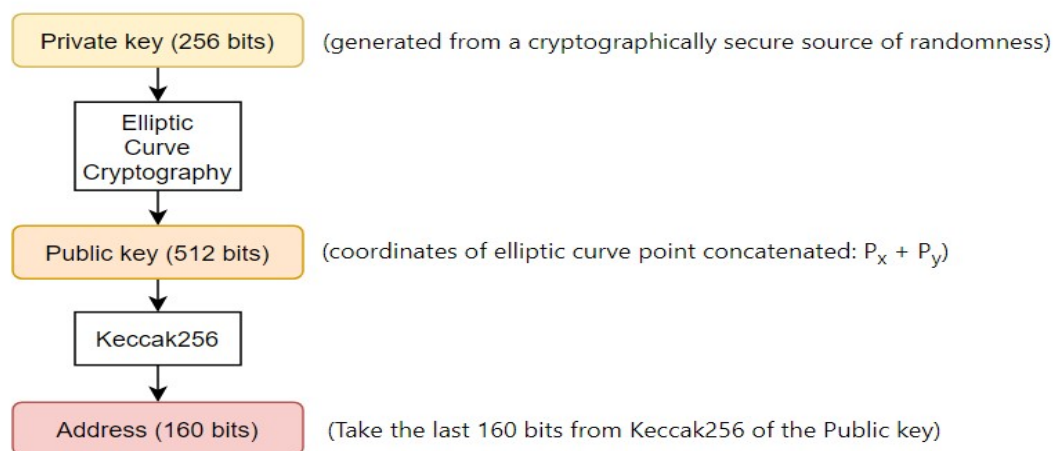


Figure 4.3: Ethereum address generation

Digital signatures

This signature is a mathematical formula that is composed of two parts. The first part consists of creating a signature from a message using a private key. The second part

consists of verifying that a certain signed message was truly signed by the respective public key.

The basic formula for signing:

$$\textit{Signature} = F_{sig}(F_{keccak256}(\mathbf{m}), \mathbf{k}).$$

Where \mathbf{k} = private key, \mathbf{m} is the RLP-encoded transaction, and F_{sig} is the signing algorithm.

To verify the validity of the signature, one needs to know the public key corresponding to the private key used for encryption. The function returns true or false.

$$\textit{Verify}(\textit{Signature}, \mathbf{K}, \mathbf{m})$$

Block structure

Every blockchain is built upon some general rules, which may or not be similar to the ones expressed in Bitcoin blockchain. Besides these rules, there are also similarities and differences in the block and transaction structure. This is what Ethereum's block header looks like:

- **Previous Block Hash:** this is what links the current block with its parent
- **Transaction root hash:** each block contains transactions and to be easy to certify their integrity we calculate a hash of them
- **Receipt root hash:** a receipt is generated for each transaction and we need to keep them certified as well
- **State root hash:** we have a state for each account in Ethereum and we need to keep a final value for all states
- **Timestamp:** the epoch Unix time when the block was created
- **Difficulty:** how much time it should take a miner to mine a block (greater difficulty means more time spent)
- **Nonce:** a value that is increased by a miner until the hash of the block meets the required difficulty
- **Gas limit:** maximum amount of gas this block should consume
- **Gas used:** how much has this block consumed
- **Extra data:** field where extra data can be stored
- **Number:** the number in the chain of the current block. Genesis block is number 1

Merkle Trees

In the previous list we saw three root hashes. These are generated from a very special data structure, called Merkle Tree. This structure allows creating a fingerprint of a large amount of data by using hash functions. These are applied in levels, up to the root node. This allows for partial verification and they are extensively used in Ethereum. [22]

The binary tree is the most usual form of Merkle Tree. It has two useful features. If something changes at any level, it will propagate through hashes up to the root level. With this, we can verify the integrity of a large amount of information using just a few hundred bits. The second useful feature is the one called "Merkle proof" by which you can verify the existence of a transaction and reliability a hash chain based on a path. To prove the point, let us say we want to check **Transaction 2**. We will only need the path from the root node to the hash of the transaction (**Hash 1234**, **Hash 12** and **Hash 2**) and the adjacent hashes to this path (**Hash 1** and **Hash 34**). This may not seem much, but when the number of transactions grows to hundreds, this verification is very efficient.

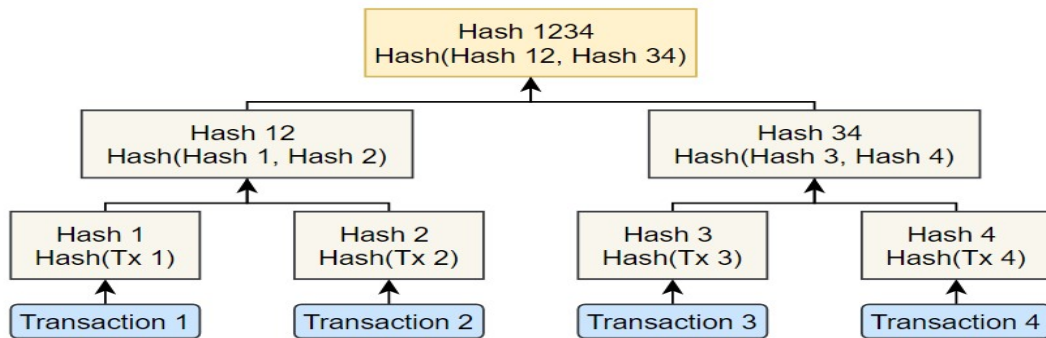


Figure 4.4: A basic binary Merkle tree

Based on this, Ethereum introduced three types of such tries, one for transactions, one for receipts (pieces of data that show the effect of each transaction) and one for state. These trees are more complex than the binary ones and are called Merkle Patricia Trees. The addition of them was due to the state. The Ethereum state is managed using a map data structure, having as key the account address and the value the corresponding data such as balance. [22] Therefore, a simple explanation for Patricia Trees is the fact that the value which is to find is stored under the path which is required to follow when searching the key. [22]

Transaction structure

Transactions represent communication between two parts, the sender being an external owned account which are then picked by the Ethereum system and then stored.

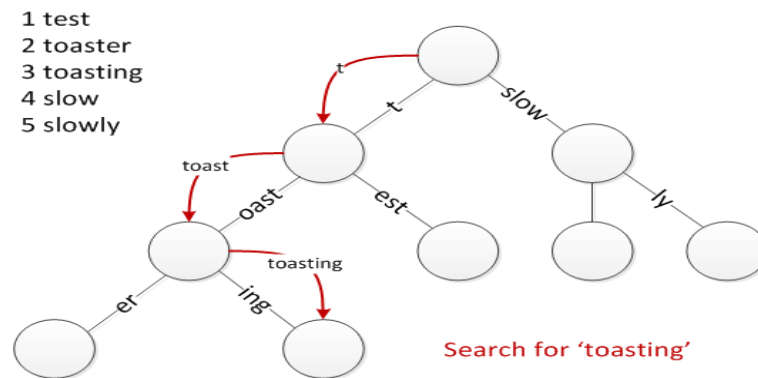


Figure 4.5: Example of Patricia Tree searching [23]

[21] By this, the transactions are the only actions that can cause a change of state of the Ethereum Virtual Machine or cause a contract's function to execute.

Each transaction contains the following data: [21]

- **Nonce:** A sequence number that is created by the External Owned Account and it is used to prevent transaction duplication
- **Gas price:** the price of gas (in wei) which the creator of the transaction will pay for the processing of the transaction
- **Gas limit:** the maximum amount of gas the creator of the transaction wants to use
- **Recipient:** the address to which the transaction is direct
- **Value:** amount of ether (virtual currency) to send to the recipient
- **Data:** payload of the transaction
- **v, r, s:** three components of ECDSA (Elliptic Curve Digital Signature Algorithm) of originating EOA

One can observe that there is no "from" address field. This is because from "v, r, s" we can deduce the public key of the account and from there, as we saw in a previous image, we can deduce the address.

Transaction Gas

There is another interesting feature in Etheruem and that is the concept of "gas". It is the fuel of Ethereum and it is a separate virtual currency with its own exchange rate against the principal currency ether. Gas is used to manage how many operations are done because of a transaction. Each such operation consumes a specific amount of gas (written in the white paper. [21])

There exists a separation between gas and ether because the value of ether may change abruptly and it may destabilize the system. Through gas, the transaction creator specifies the price he wants to pay for the processing. The higher the price, the faster the transaction will be added to the blockchain, as the fee will be given to the miner that mines the block.

Transaction value and data

Value and data constitute the main source of information of a transaction. Four cases arise from this.

- **No value, no data:** just a waste of gas, but it can be done
- **Value, no data:** payment transaction. It is a transfer of ether from the transaction creator to the recipient, which can be either an EOA or a smart contract
- **Value / no value, data:** when a transaction contains data, it is usually intended to interact with a smart contract. If there is value associated in the transaction, it usually means that the function called accepts ether and does something with it (store or perform some kind of logic) The data is separated into:
 - **Function selector:** This represent the four bytes of the hashed function name, to know which one to call. [21]
 - **Function arguments:** the arguments that should be passed into the function

Besides usual transactions described above, there exists a special transaction which creates a smart contract. This is similar to a deploy. This contact creation transactions are send to a special destination called zero address. The address is only used as a recipient. It is not an account or a smart contract. It is used only as a destination. When creating a contract through a transaction, you need to provide the compiled bytecode of the contract as payload. If you also provide value in this address, it will initialize the balance of the account to that value.

Smart contracts

We previously discussed the type of accounts in Ethereum and we reminded there the Smart Contract. These are not controlled by an entity, but they control themselves with the code written within them.

Giving a technical definition, a smart contract refers to a code that runs in a deterministic manner in the EVM, as part of the whole network. [21]

Properties of a smart contract

- **Computer programs:** smart contracts are just computer programs. They are similar to Java class
- **Immutable:** once written and deployed to the network, their code cannot be changed. The only way to make modifications is to delete the smart contract and redeploy the one with the correct changes.
- **EVM context:** smart contracts have access to their state, to the transaction that called them and little information about the most recent blocks. However, a contract can call a function of another.

Steps to create a smart contract:

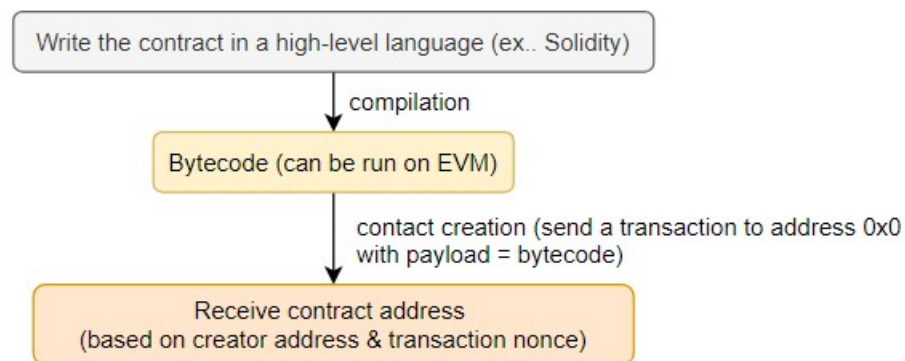


Figure 4.6: Contract creation steps

Smart contracts can be written in various high level languages such as LL, Serpent, Bamboo, Viper and Solidity. The last one is widely used and has the following features:

- **Data types and functions:** boolean, integer, address, arrays, structs, mappings etc
- **Global variables:** message call context (in a transaction the contract can access the address of the sender, the value being sent etc)
- **Built in functions:** cryptographic functions (keccak256, sha256), "this" keyword
- **Others:** smart contract support inheritance and function modifiers

4.3 IPFS

In the first part of this chapter we have discussed mainly about blockchain - how it is structured, what properties it has and how it behaves. This section will present IPFS (Inter Planetary File System) that will act as a decentralized database. IPFS was presented briefly in chapter 3 and now we will dive deeply in its structure.

What is IPFS?

IPFS is a protocol and peer to peer decentralized file system that has as a scope the changing of how we distribute information across the internet. It combines multiple technologies into a single one, creating thus a more powerful system, greater than each of its components. IPFS combines various already existing technologies such as DHT, block exchange and has the nodes in the network can work in a manner that does not require trust. [24]

It is composed of the following systems (some of which may be altered)

- **Distributed Hash Tables:** these are mainly used to distribute and maintain data in peer to peer system. We will talk about them in depth in the following pages.
- **Block exchanges - BitTorrent:** IPFS uses this concept to distribute pieces of files between nodes
- **VCS - Git:** these systems provide a way of storing file changes over time. Information within them is content addressed.

Why was it created?

We already have a protocol that is used to distribute files and that is HTTP. This protocol is good enough to send small amounts of information, even for companies with lots of traffic. However, as time passes, more and more data is created and needs to be used. This is not the only problem.

Let us consider a simple example. You and your 50 colleagues are in a classroom and your teacher asks you to access a certain website. Every student makes request to that specific website and he receives a response in return, the exact same response to be more precise, multiplied 50 times. Ideally, after one request has been made to that website (and the person who made it received the response), the further requests should be made to this first person or the subsequent ones that have the information. This way, we would not stress the external website and we would leverage already existing information which is near us.

Another problem raises when we search for a resource, find an appropriate link, but that link is expired or leads to a 404 not found. It may be frustrating, because you know

there must have been something useful there, but you missed it. IPFS helps with this issue by allowing its users to communicate and share data. This means that if someone holds a piece of information another user needs, the first user will share it and there is no need for a central server.

Related to the previous issue, the current HTTP protocol finds or serves data in a location based manner. This means that an HTTP request would look like:

`http://domain-name.com/resource.txt`

The domain name would be converted to an IP address by a DNS server and thus we would search at a location (IP location) for a specific resource (resource.txt in our case). What IPFS uses is content based address instead of location based. It uses a cryptographic hash to create a unique fingerprint of the data. We talked about such hashes earlier in this chapter. By this, whoever has that specific resource can serve us. An IPFS request would look like:

`/ipfs/QmXYDi9PbJjafcuRHDyLT4CtRmjtiDxEjaM2aYCtmKNZaj`

That long unintelligible string represents the hash of contents of the file. If the contents of the file change, so will the long string. We talked previously about the fact that we should not be able to find the original information from a cryptographic hash and this still holds true, but IPFS uses it to uniquely identify data and link that hash to it using an intelligent system DHT (Distributed Hash Tables). We will talk about it later.

Another great thing about IPFS is the fact that has a similar decentralized structure with blockchain and they can actually work together very well. We will see this in Chapter 5. Juan Benet, the creator of IPFS, states the following: IPFS connects all these different blockchains in a way that's similar to how the web connects all these websites together.

IPFS protocol is separated into sub-protocols:

- **Identities:** responsible for node generation and verification
- **Network:** responsible for managing the connection of peers
- **Routing:** responsible for locating information between nodes
- **Exchanging:** responsible for block of information exchange
- **Objects:** responsible for content-addressed objects with links
- **Files:** responsible for versioning files (similar to Git)
- **Naming:** responsible for creating a name system

DHT

Before discussing about DHT, we should talk briefly about Hash Tables (HT part of DHT). They represent a data structure that maps keys to values. This mapping is done using a hash function. We discussed about hash functions before (in Cryptography subsection). The main behavior can be seen in the next image. Mainly, we have a hash function that takes as input some data (for example, a string) and outputs a number between an 0 and some prime number - 1, which will represent the bucket that string belongs to. This is mainly done via a modulo function.

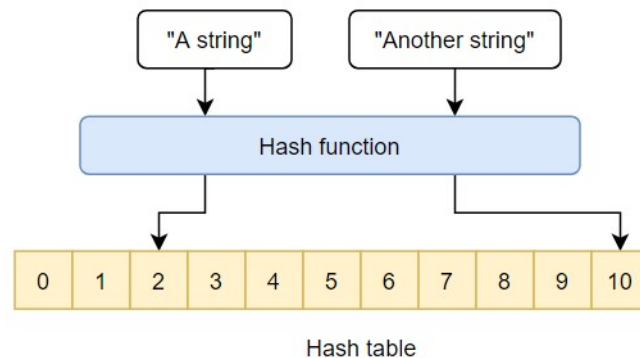


Figure 4.7: Behavior of a hash table

A problem arises when we want to add more buckets, thus increasing the previous prime number. We have used module to (among other functions) to distribute the inputs. With this new addition of bucket, we need to perform again all the previous additions, as they are not in the correct bucket after the modifications. This also happens when we try to remove a bucket. In a decentralized system, we could think of the buckets as the new nodes that join or leave the network. We cannot just recompute all values. We need a better solution.

Kademlia

Kademlia is a distributed hash table (DHT) Its mainly usage is in p2p (decentralized) computer networks. It presents the network structure and how information is searched through it. Each node has a unique id which serves as a data locator as well. Basically, data that is the closest after a hash function is stored on that respective node. In addition, each node stores a routing table that present information (about id and what data they may have on themselves) about the most closest node.

The problem with the simple hash table was the fact that all data needed to be moved after the addition or deletion of a bucket. We cannot escape this data repartitioning neither in Kademlia, but it is more efficient. If a new bucket is added, the hashed data that is the closest to it will navigate towards it, leaving the other buckets they were stored

in. By this, only a subset of our data needs to be moved between buckets and all the other stays in place.

Firstly, I will present the concept of xor distance. This is done by converting the two into bits then perform a bitwise xor operation (both bits at same position are the same => the result will be 0, otherwise 1).

The next image will present a Kademlia routing table and how distant are nodes to one another.

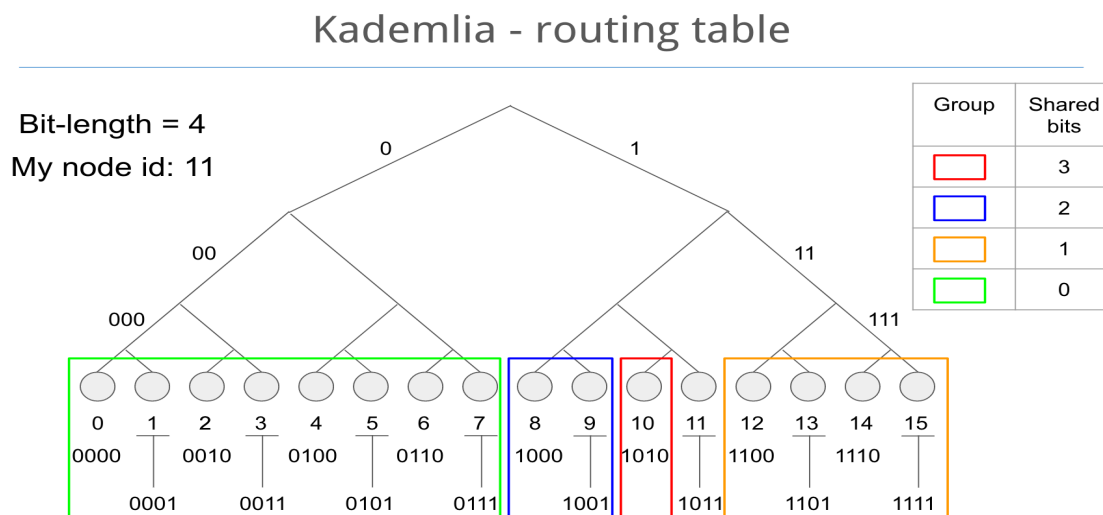


Figure 4.8: Kademlia routing table example [25]

The more bits two nodes share, the closer are they to each other. One interesting aspect we can find here is the fact that 7 and 8 seem very close to each other, but they are actually very far. They are as far as 0 and 15.

Together with a hash function (such as the one presented before) that creates hashes between an interval of 0 to 15, you could store respective information in these buckets. Of course, in real life not all nodes are available and running so gaps may appear in the list of buckets. However, data will still be put into the closest one based on xor distance between its hash and the available buckets.

IPFS uses an improved version of IPFS named S/Kademlia that is able to protect the network against malicious attacks. The NodeId generation requires a puzzle to be solved so a high amount of nodes cannot be generated in a small time. In addition, this new version allows the connection of trusted nodes even in mediums with a lot of attackers. [26]

Block exchange

When talking about block exchange, we must mention BitTorrent, which is a very successful p2p (peer-to-peer) file sharing system. This system distributes and coordinates pieces of files in a network of nodes that do not trust each other.

Based on [26], some of BitTorrent features are:

- the nodes which communicate with one another are rewarded and the ones that just use resources are punished
- parts of important, rare files are sent first when requested, so that the load is split uniformly.
- even though the current protocol is vulnerable to exploitative bandwidth sharing strategies, there exists an improvement called PropShare that is resistant to this practices

BitSwap Protocol

IPFS' protocol was inspired from BitTorrent as it has its main behavior: exchanging blocks between peers. They have two sets of blocks: **want_list** (the set of blocks they desire) and **have_list** (the set of blocks that they can offer). Unlike BitTorrent, a node from BitSwap can ask and receive any part of a large file (unrelated to the main one) without downloading it as a whole.

As in any type of such system, there are blocks (or files) that are wanted more than the others and are considered more valuable. By this, even though some nodes should not be interested in keeping them, they are rewarded to cache and distribute them across the network when needed.

Some features of this protocol: [26]

- **BitSwap Credit**: this represent incentivization formulae for nodes when they store or retrieve data. The peers take into consideration the amount of sent data. Over time, the nodes that have debt are no longer sent information.
- **BitSwap Strategy**: the chosen strategy (among a few) should maximize the trade performance and the whole exchange while preventing leechers from exploiting and degrading the system. Moreover, the strategy should be resistant to known and unknown attacks.
- **BitSwap Ledger**: the nodes track the amount sent to one another and keep history of the transactions. If the two ledgers do no match, it will be destroyed and recreated empty.

Object Merkle DAG

On top of DHT and BitSwap technologies IPFS added an Object Merkle DAG, a directed acyclic graph where the connections between objects are hashes created by cryptographic functions of the targets which are part of the sources. From this, three useful properties arise, based on [26]

- **Content addressing:** all content (including links) is uniquely identified by a multihash checksum. We will talk about it later.
- **Tamper resistance:** all content is verified with its checksum. If data is corrupted IPFS is able to detect it.
- **Deduplication:** all objects (including links) that hold the same content are stored only once.

This feature of IPFS can be applied for folders as well. When you add a folder to IPFS you get in return a multihash of that folder (you can view it as a root) and multihashes for each element within the uploaded folder. By this, one can access the whole folder or just parts of using the respective received multihash.

But this is not all you could access the sub files or folders directly from the root hash, like in default file systems. The next image presents this concept in a very clear fashion.

```
PS C:\Users\Ionut\Desktop> ipfs add --recursive .\root_folder\
1 B / 3 B [=====] 33.33%
added QmY4ZacGXPTqRmTVEGSC6PPMgn1AhMD1wPBTH7PeU3qtAf root_folder/another_folder/another file.txt
2 B / 3 B [=====] 66.67%
added QmWYddCPs7uR9EvHNCZzpguVFNfHc6aM3hPVzPdAEESMc root_folder/file1.txt
3 B / 3 B [=====] 100.00%
added QmT9SanPHnSH5AsBqy2xZbstw4rAw5znFPkmjKvDCMdVuF root_folder/file2.txt
3 B / 3 B [=====] 100.00%
added QmT45J9BLEtmqwgTSakKe9hw6eIdMwUzPtGpmrzhavZ2mj root_folder/another_folder
3 B / 3 B [=====] 100.00%
added QmNUhFC1NVtor9xrAjkjgvxEydRjS1qLzXvHAVZH8TyjPZ root_folder
3 B / 3 B [=====] 100.00%
```

Figure 4.9: Adding a folder to IPFS

The following figure explains how a file is split into blocks if necessary.

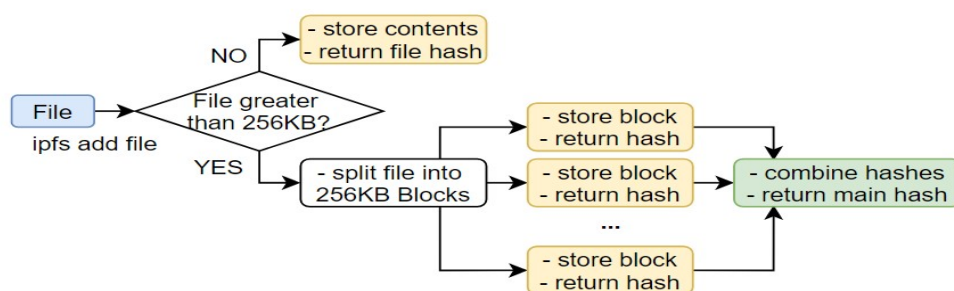


Figure 4.10: Add a file to IPFS

How is the CID created?

We already know that data is located through its content, not its location. [24] A content identifier or CID is a self-describing content-address identifier. It is created from the actual data it contains. Based on the cryptographic hash function which was used to generate the CID, it can be of various lengths. Usually, in IPFS, sha-256 is used to generate such hashes.

But one can discover an issues with approach. In future, IPFS may migrate to other cryptographic function that will be used to generate content identifiers (because some vulnerabilities may be found) and end up with two types of hashes in their system and they would no be able to differentiate between them. IPFS came with the idea of multihash formats, a self-describing hash which follows TLV (type-length-value) pattern.

<algorithm-used><length-of-generated-hash><actual-hash>

For example: <sha-256><32 (bytes)><hash>

In addition, IPFS created a table that contains identifiers for every cryptographic function and much more. This will be used instead of the name of the function. It is important to note that the resulting CID fields will be interpreted into hex. Therefore, the previous example would look like:

<12><20><hash>

Bits without separation: 0001 0010 0010 0000 ...

In order to represent the result in a more compact way (instead of 1s and 0s), we need to encode it. When IPFS was created, **base58btc** (created at bitcoin beginnings) was used. Because every hash would start with 1220 (sha-256 code and 0x20 length), their respective encoding would start with Qm... This is referred as CIDv0 as it encompasses the **multihash**.

CIDv1 has more metadata within it and is far more extensible. It specifies:

- **multicoded prefix**: which encoding was used on the data (the way the data is structured. For example: raw string, json etc)
- **version of cid**: 0000 0001 (for CIDv1)
- **multibase prefix**: which base encoding was used (for example, base58etc)

Chapter 5

Detailed Design and Implementation

5.1 System diagram

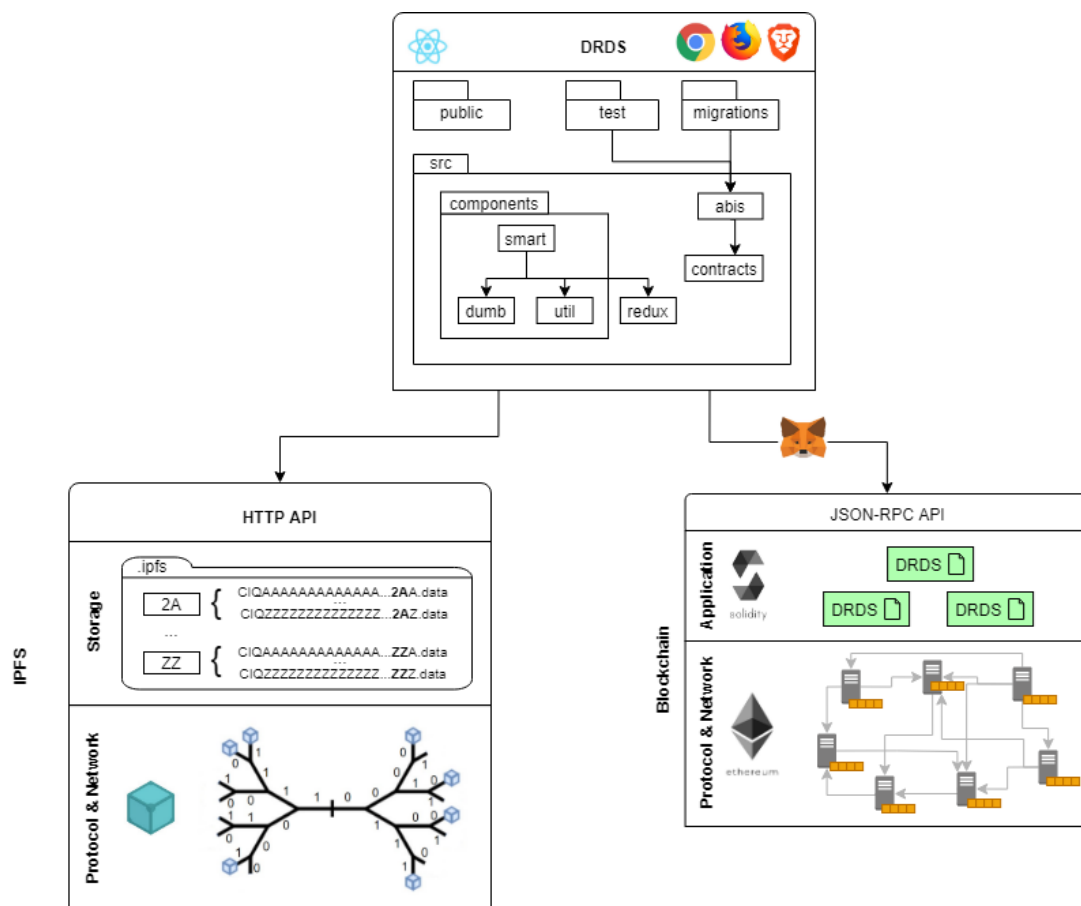


Figure 5.1: High level architecture

At a high level, the system is composed of three main parts: the DRDS (Data reliability using decentralized systems) application, the Ethereum node and the corresponding smart contract and the IPFS node. Each of them will be further analyzed and presented from a lower perspective.

Starting from the top, I will explain the role of each component and how it integrates in the whole system.

DRDS

It is the main application that I have developed. It is a browser based system, i.e. it is accessible from a web browser. This can be seen from the diagram, as in the right of the DRDS module there are three logos (Google Chrome, Mozilla Firefox and Brave browser). There are only three because these are the only ones that support or include the Metamask plugin which enables the main application to interact with the blockchain network.

On the left part of the module we see the React logo. This is because the application was developed using the React library together with some other third-party libraries (downloaded from npm) that will be discussed below.

Ethereum

On the bottom right part of the diagram we observe the Ethereum network which represents the blockchain part of the system. Even though it is displayed as a single module, in the implementation may contain a lot of nodes (thousands) that communicate to achieve a single state (the consensus).

This module stores the smart contract and the state of our system, the transactions that have been processed and even the events that have been raised during certain transactions. The icon represents the Ganache tool, which was described in the technologies section above.

Metamask is used to carry the transaction requested from the client app to the node instance to which it is attached. Javascript Object Notation - Remote Procedure Call (JSON-RPC in short) is the protocol that is used for communication. From a developer point of view, the calls made to ethereum network feel as if the smart contract was in the same object space as the rest of the application. In reality, we only have a proxy (a wrapper) on which we call the functions. This proxy then calls the respective method over the network and responds back with required data.

If the function called modifies the state of the contract (and therefore needs a certain amount of virtual currency), Metamask will prompt the logged user a transaction panel from which the transaction can be accepted or denied.

IPFS

On the bottom left we can see the IPFS node which represents the storage part of the system. You can think of it as the database of the application, even though it is much more than that. The node is able to do the following:

- **store files:** this is done by creating a sort of file system, where each block of data resides in the corresponding folder (determined using a special function)
- **connect to the main network:** the local node can join the public, global network of nodes and interact with them by exchanging files (if needed). Moreover, this node can be queried from the central gateway which is *ipfs.io/ipfs/<hash-of-file>* and it will serve the specific file. However, this process is not happening instantaneously.
- **create a graphical user interface:** various information can be seen here, such as the total amount of data stored, the number of connected peers

The communication between DRDS and IPFS node is done through an HTTP API. There exists an npm package ("*ipfs-http-client*") that does this automatically under the hood and you only need to call predefined functions that will convert to requests.

5.2 Technologies used

Programming languages

The two programming languages that were used are Javascript and Solidity. Javascript was used to build the actual application that links the two main systems (Ethereum and IPFS) together. Solidity was used to compose the smart contract that are deployed to the Ethereum network. In the incipient phase of the project I have started with Java instead of Javascript, but I have quickly realized that the development of the features in this language are slow and sometimes many versions old than the up to date Javascript ones. This was due to the fact that Java is strongly typed and has very rigorous rules.

Besides being easy and fast to implement features in Javascript, it also the language that is widely used for websites (as the frontend part) and it has many frameworks built around it (for example, React, which I will talk about later). One of the goals of this project was to create an application that was visual and that which presents the end user a pleasurable experience and it surely can be done with Javascript.

I have chosen Solidity to write the smart contract logic because it is the primary language used on Ethereum. It is similar to Java, having the overall structure of code. Being the most popular, a large community that helps each other has grown around it and there are many tutorials which helped me during the development.

Metamask

Most of the Internet nowadays is centralized, i.e. there exists a central server that has all the control. We access this type of Internet everyday through our browser mostly. However, the decentralized Internet is not as simple and you need special applications to connect to it. Ethereum is one of this decentralized applications (also called dapp) and you need special software to connect to it.

Metamask is such a software. It is a browser extension that allows its users to connect to the decentralized Ethereum network through the comfort of their usual browser (for example Google Chrome), without the need of being part of the whole network as an independent node. Moreover, it can manage your Ethereum wallet which contains ether virtual currency and allows other applications to connect to it (provided you allow them to do this) and interact to with the smart contracts that have been deployed there.

Truffle

When we write code, we at least need to compile it and then run it. This also is applicable to code written in Solidity. We can do this by hand, running commands from terminal, but the process of compiling and then deploying to a network is tedious and somehow complicated. Truffle is a decentralized application environment which helps developers with the process of creating a dapp. It contains a testing framework and a pipeline of actions that can be run with just one command. By this, smart contracts are compiled, linked, deployed and managed in an effortless manner.

Ganache

The previous framework Truffle is part of three sister applications that make development of decentralized application easier. The second application that I have used is Ganache which is a blockchain emulator. One way of running locally an ethereum node is to create a new private network, configure and start it as a miner. However, it is difficult to work with it, resource consuming (both CPU and memory) and you can only see console logs. Ganache emulates this behavior and adds a user interface on top of it.

IPFS node

When working with IPFS, you need to connect to a node (similar to Ethereum approach). One way of attaching to a node is to use an external service (such as Infura) which allows you to communicate through an API and store information remotely. However, for my project I had to use a local node, because I wanted to demonstrate the corruption issue. IPFS allows you to run a node on your computer.

Moreover, if you run a local node, you can join the public network and serve files from your local machine. This serving is done using the official gateway which can be found on IPFS official site: <http://ipfs.io/ipfs/<cid-of-file>>

React

We have previously seen that Javascript is the best choice when working with projects that are under development (such as Ethereum and IPFS), because it is very flexible and you do not need any special constructs around it (because it is dynamically typed). As we needed a starting point, a framework for Ethereum code, we need a starting point for our Javascript application and that is React.

React is a Javascript library used for frontend that manipulates HTML DOM (document object model) by creating (reusable) components to change the information within a page, without the need of refreshing it.

NPM

We talked above about some open source packages. These are created and uploaded to NPM (node package manager). It is the largest software registry (library) and contains hundreds of thousands of code packages. It allows the developers to create and download Javascript modules and it is totally free. It has a console interface and the most basic command is the one that downloads a package and stores it into your project:

5.3 Component diagram

In this section I will present in depth each module from the previous system diagram.

DRDS

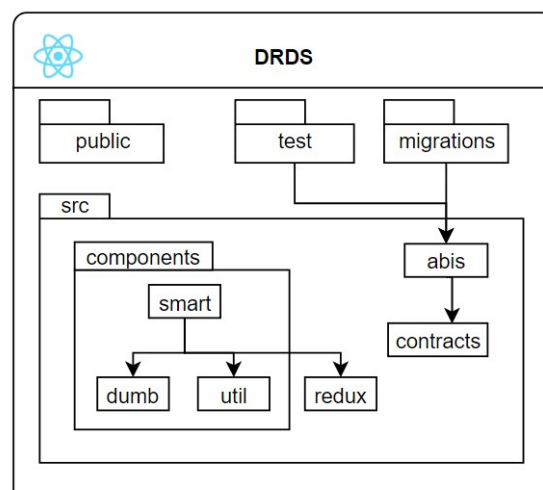


Figure 5.2: DRDS Architecture

DRDS is a React application combined with a Truffle one. At the root of it, there are two important files that represent the global configuration of the project. There are not shown in the previous package diagram, but they are at the same level with "public" directory.

- **package.json**: presents basic information using of the react application as key-value pairs such as name, description, author, the dependencies used in the project and some scripts that can be run to start the application. For example, some of the dependencies used in this project are: ipfs-http-client, redux, truffle, web3 etc
- **truffle-config.js**: presents key-value pair information regarding the ethereum configuration. Here the user can specify the network that is being used, the folder of the contracts, the folder of the abis and which compiler is used for the smart contract

At the root level we can also find the following packages:

- **migrations**: contains the migration files that are used to deploy the smart contracts on the Ethereum network. Truffle framework needs this package so that is able to manage the deploys. These files are prefixed with a version number and, at "*truffle migrate*" command, are run in the order of their version.
- **public**: contains the images of the application and the index.html, the only page of the application. As the user interacts with the system, the page remains the same, but the components move or change. This is the "Single page application" paradigm used by React and some other libraries or frameworks.
- **node_modules**: we discussed earlier about the fact that some libraries are downloaded using "npm". This is the location of the download and all required dependencies reside here. Node modules are not shown on the diagram because they are generated (or downloaded) when "*npm install*" command is run. In terms of dependencies, all project packages depend on them.
- **test**: this folder holds the tests that are run against the smart contract. One can test here all use cases that may appear in their smart contract and be sure that it does what it is intended. It is done by using the "chai" together with "chai-as-promise" npm dependencies.
- **src**: this is the source folder, where the core of the application resides. I will explain it in more detail below.

Src folder is the central source point of a React application, as here the development is being performed. At the root of this folder there is a

- **index.js** file that is the entry point of the application. Here, the react project is injected into the root element of *index.html*.

Besides this file, `src` folder contains four more folder which will be discussed briefly below. In no particular order:

- **contracts:** this folder contains the actual solidity code for the smart contract. This folder is needed by the Truffle framework so that the smart contracts can be compiled and then deployed on the Ethereum network. I will present the code in the Ethereum module.
- **abis:** this is the build package, which contains the smart contract files in a json key-value pairs format. This is very helpful, because the transformation allows my application to interact with the contract using Javascript programming language. You can think of it as an adapter. Portions of these files will be shown briefly in the Ethereum module.

Redux is the third package inside the `src` folder. Besides being a package name, Redux is an entire library downloaded through npm that manages the global state of an application. Out of the box React has a *state* attribute in every class that extends *Component*. However, this state is not applicable to the whole application and the management can be tedious throughout the various locations.



The screenshot shows two code files in a code editor. The top file, `App.js`, has line 80 with the code `this.props.onAccountAddressChange(accounts[0]);`. The bottom file, `ethReducer.js`, contains the following code:

```

1  const initialState = {
2    contract: null,
3    accountAddress: '',
4    fileHashAddedEvents: [],
5    fileEditedEvents: []
6  };
7
8  function changeEthState(state, property, value) {
9    return {
10     ...state,
11     [property]: value
12   }
13 }
14
15 function ethReducer(state = initialState, action) {
16   switch (action.type) {
17     case "ACCOUNT_ADDRESS_CHANGE":
18       return changeEthState(state, property: "accountAddress", action.value);

```

Figure 5.3: Redux flow

Above I have attached portions of code where Redux is used. The order of function calls is from top to bottom and they do the following:

1. **App.js, line 80:** this function is called when a Metamask account change is detected. This in turn calls
2. **App.js, line 253:** this function propagates the event ("ACCOUNT_ADDRESS_CHANGE" in our case) to the reducer (takes care of the logic that needs to be run when the event occurs)
3. **ethReducer.js, lines 17-18:** the action type is detected and the attribute *accountAddress* of the main ethereum state is changed to the value received as parameter (*accounts[0]*)

After the change is registered, it can be further used in the whole application, by accessing it from a global variable attached to the React component. It may seem a lot of work just to store a variable, but as the project grows, the management is very important.

Components directory inside src contain the pure React files that are being rendered on the screen. At the root of this package we have two files: *App.js* and *App.css*.

App.css contains all the stylings of the application, including the button colors, the way elements are placed in the html page and so on.

App.js is the most important file of the application, as it coordinates all requests, checks and decision that need to be done throughout the application. The file also contains a local state based on which certain decision are taken. Therefore, *App.js* performs the following:

1. **checks for Metamask:** here resides the checking of Metamask. If the extension does not exist, the user is not logged in into or it denies the application permission to it, the metamask status of the state will change accordingly.
2. **loads blockchain data:** if the user accepted the connection of DRDS to its Metamask account, this function is called. Its role is to
 - set in the global state the account used (received from Metamask)
 - instantiate the DRDS Smart Contract (by using its abi and its corresponding address from the selected network) for further use
 - get the uploaded files of the current user
 - get the events of the smart contract (for demonstration purposes)
3. **get events from contract:** this function queries all events of the current user generated in the DRDS smart contract by an interaction with it. These events are stored on Redux as well.

There are two different kind of events (*fileAddedEvent* and *fileEditedEvent*) and show various information such as: the block hash, the block number, the transaction hash, the name of the file that generated the event etc.

4. **gets uploaded files of the user:** this method calls *getUploadedFiles()* function from the smart contract. For each uploaded file, it gets the latest version (as editing is possible). For each latest version, it stores it on the global state, and in the end it validates the files (the corruption detection part).
5. **validates the files of the user:** after the latest version of the files are queried from the blockchain, they need to be validated against the information stored on the IPFS node (to check for file corruption, as the data is stored on untrusted nodes). This is done in this function via a call to one of the utils functions (*getValidatedFile()*), which makes http requests to the IPFS node. The actual code will be revealed when the IPFS module is explained in detail
6. **component rendering:** based on the url of the request, this file decides what component to render further

The decision of component rendering is made by the following piece of code (based on the received URL):

```
case "connected":
  if (this.state.loadedFiles === true) {
    componentToRender = <HashRouter>
      <Switch>
        <Route exact component={SmartHomepage}
          path="/" />
        <Route exact component={SmartFileInformationPage}
          path="/uploaded-files/:fileHash" />
        <Route exact component={SmartFileLinkPage}
          path="/uploaded-files/:fileHash/file-link/:linkHash" />
        <Route exact component={SmartFileVersion}
          path="/uploaded-files/:originalHash/version/:selectedHash" />
        <Route exact component={SmartDebugHash} path="/debug-hash/:fileHash" />
      </Switch>
    </HashRouter>;
  }
  break;
```

Figure 5.4: Redux flow

Besides these files, components package contains three packages, each with a different role:

1. **dumb:** these files only present the information received as parameters. They are just functions that perform the rendering and allows the user to extract code that can be used in other places (thus removing duplication). However, rendering and html code also happens on the files contained in smart package. The dumb files are:

- **metamask related:** *ConnectToMetamask.js*, *MetamaskNotInstalled.js*: presents the error pages in case Metamask is not able to connect
 - **files related:** *UploadedFilesPanel.js* - displays each file received from Ethereum and IPFS using a *FilePanel.js* for each uploaded file. *LinkPanel.js* - displays link information based on block splitting which happens on IPFS. *UploadFileForm.js* is used when a user uploads a new file in the system.
 - **validation related:** *InvalidFileIcon.js*, *ValidFileIcon.js* - these are used to show whether the uploaded files are valid or not.
 - **global:** *MyNavbar.js* - is present on all pages, shows the address of current user and contains the form that redirects the user to the page that performs the hashing debugging
2. **util:** this package contains util functions that are used in multiple places of the application. They do not belong to a single component, but they may help more than one. We have
- **DRDSUtil.js:** has a function that gets the preview of data to be displayed on the *SmartFileInformation.js*
 - **IPFSUtil.js:** has functions that interact with IPFS, one for getting partial data to be displayed and one to validate it. Moreover, this file uses the "ipfs-http-client" npm dependency.
 - **ReactUtil.js:** has a function that gets the parameters from an url. This is used extensively when navigating to an older version of some file, as all the required data is passed through url parameters
3. **smart:** they can store a state (besides the global one) and make use of the dumb components by providing them the information to be presented to the end user. We have five smart components, one for each of the urls that can be accessed in our application:
- **SmartDebugHash.js:** component that finds where a file corresponding to a hash will be stored on the computer. If that specific file is modified, a corruption is produced.
 - **SmartFileInformationPage.js:** based on the hash received via an url parameter, a file is presented together with its links. Moreover, versions of this file are loaded and the user can navigate to them.
 - **SmartFileLinkPage.js:** presents the information of one link (or block) from a file provided that link is valid. If not, an error is shown.
 - **SmartFileVersion.js:** presents an old version of a file. A user can arrive on this page if he selects from a dropdown of this *SmartFileInformationPage.js*

- **SmartHomepage.js**: this is the page that the user sees. From here, he can upload files to DRDS, he can edit them, he can see various information about them (the block in which the transaction that stored the file hash) and can see the integrity of the file.

Only the most recent version of each file is presented here.

Ethereum

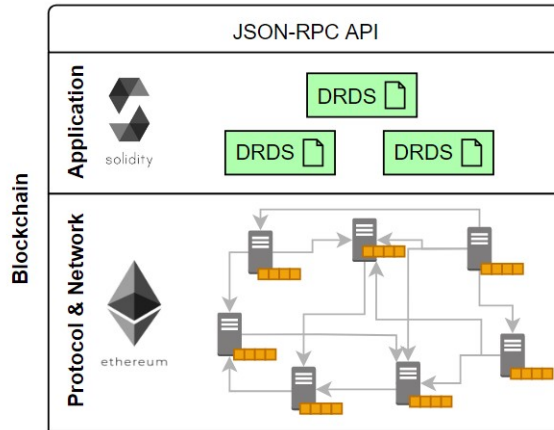


Figure 5.5: Ethereum architecture

The second module of the application is represented by the ethereum blockchain. As presented in the chapter 4, it is a network of nodes (peers) that run based on a protocol and consensus.

The DRDS smart contract contains as fields the following:

- **FileInfo struct**: this represents a block of data composed of a file name and a file hash. I use this custom datatype to work with files (to send and receive them)
- **four mappings**: these contain the state of the contract
 1. **mapping(string => address) fileHashes**: maps a file hash to a user address. By this, we know who uploaded a specific hash
 2. **mapping(string => FileInfo) firstVersionOfFile**: maps a file hash to a FileInfo, as we need to know first version of a file
 3. **mapping(string => FileInfo) latestVersionOfFile**: maps a file hash to a FileInfo, as we need to know latest version of a file
 4. **mapping(address => FileInfo[]) uploadedFiles**: maps a user address to a list of FileInfos, because we need to know all the uploaded data of a user.

- **two types of events:** when a file was added to DRDS and when a file was edited in DRDS

These are the two main functions of the contract.

```
function addFileHash(string memory _fileName, string memory _fileHash) public {
    require(fileHashes[_fileHash] == address(0), "File already in the system!");

    fileHashes[_fileHash] = msg.sender;
    uploadedFiles[msg.sender].push(FileInfo(_fileName, _fileHash));

    firstVersionOfFile[_fileHash] = FileInfo(_fileName, _fileHash);
    latestVersionOfFile[_fileHash] = FileInfo(_fileName, _fileHash);

    emit FileHashAdded(msg.sender, _fileName, _fileHash);
}

function editFile(string memory _oldFileHash, string memory _newFileName, string memory _newFileHash) public {
    require(fileHashes[_oldFileHash] == msg.sender, "You did not upload that file!");
    require(fileHashes[_newFileHash] == address(0), "The new version of your file was already uploaded!");

    FileInfo memory _firstFileInfo = firstVersionOfFile[_oldFileHash];
    string memory latestVersionFileInfoHash = latestVersionOfFile[_firstFileInfo.fileHash].fileHash;
    require(keccak256(bytes(latestVersionFileInfoHash)) == keccak256(bytes(_oldFileHash)),
        "You can edit only the last version of your file!");

    latestVersionOfFile[_firstFileInfo.fileHash] = FileInfo(_newFileName, _newFileHash);
    firstVersionOfFile[_newFileHash] = FileInfo(_firstFileInfo.fileName, _firstFileInfo.fileHash);
    fileHashes[_newFileHash] = msg.sender;

    emit FileEdited(msg.sender, _firstFileInfo.fileHash, _newFileName, _newFileHash);
}
```

Figure 5.6: DRDS Smart contract Add and Edit functions

The **plagiarism detection feature** is implemented inside *addFileHash()* function. Having a mapping that stores "string -> address" key value pairs, we know if a certain file hash (string) was uploaded in our system (the corresponding value is different from address 0x0). Using a built in functionality of Solidity, we can force the desired behavior: a file can be uploaded just one time, no matter by which user. An exception will be raised otherwise and the execution will be aborted. This will trigger even if the original author tries to upload the same file twice.

File versioning feature is also implemented in the above piece of code. After some basic validations, we modify the latest version of the original file to the one uploaded. Moreover, the one uploaded will have their original version set correspondingly. At the end of the *editFile()* function we emit an event that will be stored on the smart contract. In DRDS we iterate over these and associate them to a specific file so that the user can see their edit history.

After the contract is transformed into a json object, we are able to call the functions directly from javascript and Truffle together with Metamask will take care of the rest.

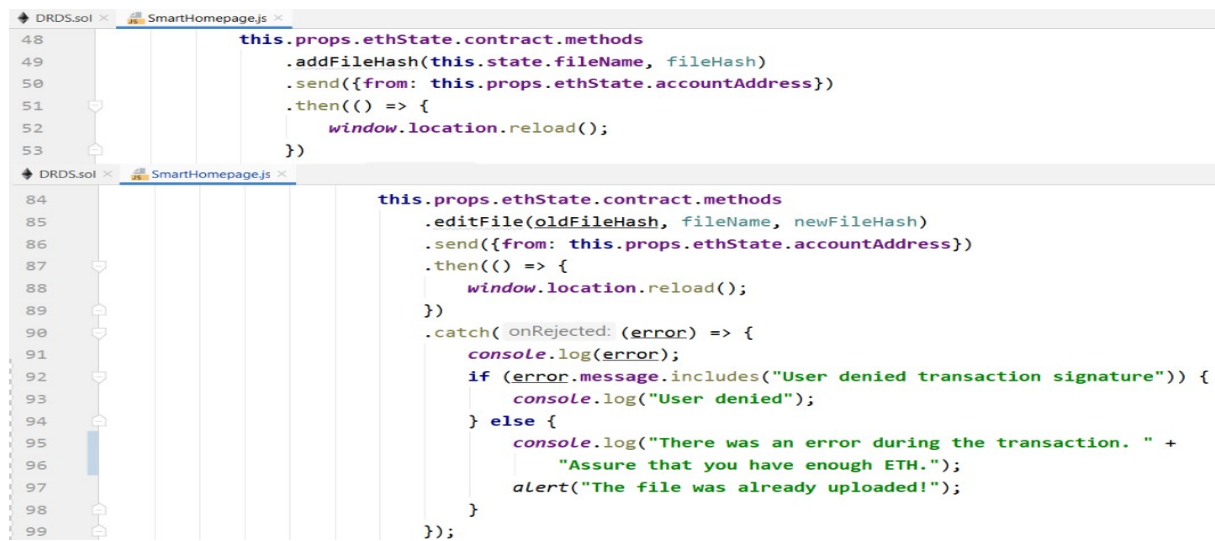


Figure 5.7: Smart contract calls from React application

IPFS

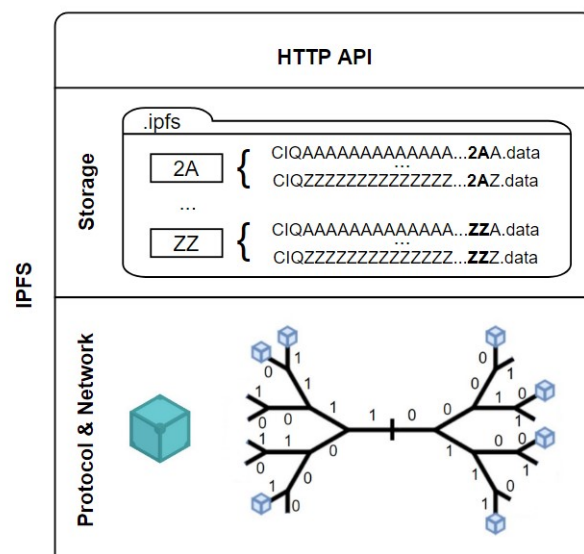
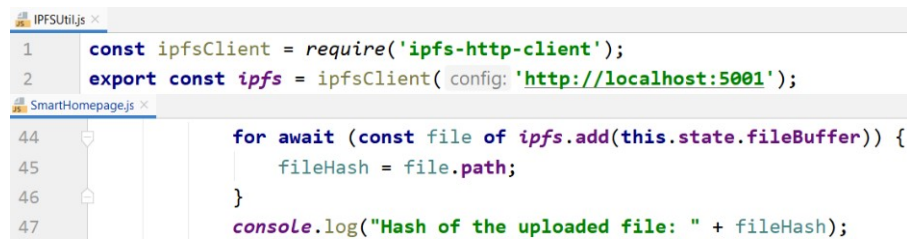


Figure 5.8: IPFS architecture

For demonstration purposes, the IPFS stack has been grouped into two large layers. The first layer consists of the storage, the way IPFS saves and deduplicates data. The algorithm was explained in chapter four. The second layer represents the topology of the network and resembles Kademlia, which was also explained. The calls to IPFS from DRDS were made using npm:



```

1  const ipfsClient = require('ipfs-http-client');
2  export const ipfs = ipfsClient( config: 'http://localhost:5001');

SmartHomepage.js
44  for await (const file of ipfs.add(this.state.fileBuffer)) {
45    fileHash = file.path;
46  }
47  console.log("Hash of the uploaded file: " + fileHash);

```

Figure 5.9: Interacting with IPFS from React

One of the most important function if not the most important is the one that **verifies data corruption**. As presented in chapter 4, IPFS is a decentralized system and the information may reside in unknown nodes that may want to corrupt data for personal purposes. The corruption attempts should be detected both on the block level (links) and top level (actual file that stores the links).

One important note: IPFS is able to detect the corruption attempt, but only at one level. Therefore if a link becomes valid, it is unable to tell that the whole file that contains the link is invalid and tries (and does not succeed) to load the file.

My implementation is the following:

1. **check file level:** we try to access the data from a file hash that the user uploaded. This file may be larger than 256KB and therefore will store the references to its links that contain the data.

If there is a hash mismatch, we know the file is corrupted at the highest level, we cannot access its links and therefore we mark it as invalid.

2. **check link level:** if we arrive at this point, we know that the highest level is intact and need to check its links.

For each one of its links (retrieved at 1.), we perform the same operation: we try to access the link data. If an error occurs due to hash mismatch, we mark the link as invalid and the whole file invalid. It is important to note that the other links may be still valid and information can be seen in them.

One important note here. While we perform the validation at the two levels of data, we try to store it (provided it is valid) in the global storage so that it is retrieved only once: at page loading. By this, we eliminate the redundant calls to IPFS.

After the homepage is fully loaded and all user's files are checked for integrity, the user can move at a fast pace throughout the application, as data is accessed from the global storage and no subsequent network calls are made except for the ones related to file versions. We cannot store all the history of all files, as it is practical.


```

20 export async function getValidatedFile(fileHash, dataToCopy) {
21   let isValid = undefined;
22   let linkResult = [];
23   let totalSize = 0;
24   try {
25     let nodeData = await ipfs.object.get(fileHash);
26     isValid = true;
27     totalSize = nodeData.size;
28
29     let links = [];
30     nodeData._links.forEach(link => links.push({size: link.Tsize, hash: link.Hash.string}));
31     for (let j = 0; j < links.length; j++) {
32       let linkIsValid = undefined;
33       let partialDataToDisplay = '';
34       try {
35         partialDataToDisplay = await getPartialData(links[j].hash, {maxChars: 2000});
36         linkIsValid = true;
37       } catch (e) {
38         if (e.message.includes("block in storage has different hash than requested")) {
39           linkIsValid = false;
40           isValid = false;
41           partialDataToDisplay = "INVALID FILE";
42         }
43       }
44
45       let updatedLink = {
46         ...links[j],
47         linkIsValid: linkIsValid,
48         partialDataToDisplay: partialDataToDisplay
49       };
50
51       linkResult.push(updatedLink);
52     }
53   } catch (e) {
54     if (e.message.includes("block in storage has different hash than requested")) {
55       isValid = false;
56     } else if (e.message === "Failed to fetch") {
57       alert("You are not connected to an IPFS node!");
58     } else {
59       console.log(e);
60     }
61   }
62
63   let partialData = "INVALID FILE";
64   if (isValid) {
65     partialData = await getPartialData(fileHash, {maxChars: 2000});
66   }
67
68   return {
69     ...dataToCopy,
70     totalSize: totalSize,
71     isValid: isValid,
72     links: linkResult,

```

Figure 5.10: File validation function - recursively on the links

Chapter 6

Testing and Validation

This chapter has the purpose of detailing the way testing and validation of the developed application is done. Based on some predetermined metrics, some tables will be showcased together with explanations.

6.1 Description of data

As we work with files, we will test our system regarding to them. I have decided to use three files with different attributes for testing.

The files are:

- **test-file.txt**: a text file of size equal to **47 bytes** containing the phrase: *"This should be a text file of size below 256KB."*
- **fibonacci.txt**: a text file of size equal to **949 kilobytes** containing the first 2999 numbers of Fibonacci sequence.
- **test-image.jpg**: a jpg file (an image) of size equal to **8,62 megabytes** which presents a room with tables and chairs.

These three files cover the basic storing and splitting functionality of IPFS. The first file is under 256KB and therefore it will be stored directly as a file. The second and third files are over 256KB and therefore they will be split into links of 256KB. *fibonacci.txt* will have 4 links whereas *test-image.jpg* will have 35.

The image was added as a test file only to demonstrate the behavior of IPFS. It does not care about the nature of the stored files (txt, jpg or any other kind) as it perceives them simply as bytes.

When the splitting is done (for second and third file), the files will contain a json object with IPFS links to the blocks that constitute the file. By this, the size of the overall object will increase by a small percentage.

6.2 Metrics

The only metric that will be used in evaluating the solution is **the cost of storing data**. We will compare the overall price (in Ethereum virtual currency and then actual dollars) required to store data on chain (on Ethereum blockchain) to the cost of storing it on IPFS.

Moreover, we will take into account the **transaction throughput** which will be recorded on blockchain, as the more gas a transaction require, the fewer of them will be stored on the chain (due to the gas limit).

We will perform the evaluation for all the three files discussed above as they cover the basic use cases of DRDS.

6.3 Experimental results

Ethereum network runs on gas (as explained in Chapter 4). For the experimental results I have used the following values at time of writing:

- the cost of one gas: 25 Gwei = 0.000000025 ETH (standard price)
- the cost of one ether (ETH): \$224.83
- gas limit: 11 497 817 gas / block
- cost of storing 1 byte in a block (through a transaction): 68 gas (from Ethereum yellow paper)
- minimum cost of a transaction: 21 000 gas (from Ethereum yellow paper)
- **we do not take into account the transaction fees**

From these, we can conclude that we can store:

$$\text{maxDataPerTransaction} = (11497817 - 21000)/68 = 168.776 \text{ bytes}$$

Formulae

Formula for calculating the number of ether required for **one** transaction:

$$\text{Ether} = \text{nrOfGas} * \text{gasPrice} = (21000 + \text{bytesOnTransaction} * 68) * 25\text{Gwei}$$

Formula for calculating the ether required for a full transaction.

$$Ether = maxNrOfGas * gasPrice = 11497817 * 25Gwei = 0,287$$

Formula for calculating the number of ether required for **n** transactions:

$$Ether = etherFull_1 + ...etherFull_{n-1} + etherForLastTransaction$$

Formula for calculating the price in \$:

$$Price = ether * priceOfOneEther = ether * 224.83$$

Formula for calculating the transaction throughput on the chain:

$$NrOfTransactionsPerBlock = maxNrOfGas / medianGasPerTransaction$$

There are on average **140 transactions per block**. This data is calculated by:

$$AvgTxPerBlock = totalNrOfTxPerDay / totalNrOfBlocksPerDay = 900.000 / 6.400 = 140$$

This number is calculated in normal conditions, when Ethereum is used as a global state machine and not as a storage platform.

By this, we can calculate the average gas consumption of one transaction:

$$TxAvgGasConsumption = maxNrOfGas / AvgTxPerBlock = 497.817 / 140 = 82.000$$

We can deduce the average amount of data (non-zero bytes) sent in a transaction:

$$AvgDataAmount = TxAvgGasConsumption / GasFor1Byte = 82.000 / 68 = 1.205 \text{ bytes}$$

Storing files on Ethereum

Table 6.1: Results for the three files when using only Ethereum

File name	File size	Transactions required	Ether required	Price (\$)	Average tx per block
test-file.txt	47 bytes	1	0,000604	0,135	140
fibonacci.txt	949 kilobytes	6	1,614	362,875	15.16
test-image.jpg	8,62 megabytes	52	14,658	3.295,558	1.19

From the above table we can clearly see how expensive is to store data on Ethereum blockchain and how much change it produces inside the system. From an average of 140 transactions per block, we can reduce them to less than 10. While computing the results,

we assumed that the data stored had no bytes of value 0 (as they need fewer gas when stored). However, the change would not be major and, therefore, we needed to explore alternatives.

Storing files on IPFS and linking them in Ethereum

The actual file storage happens in IPFS and the hash to the file is stored on blockchain. This means that every file, no matter how large it is, will have 46 characters, based on the IPFS' algorithm of generating the CIDv0 (which was explained in Chapter 4). Together with the file hash, the name of the file and the address of the sender is stored on the smart contract.

On the IPFS side, the user has multiple choices, depending on his needs. He can: run its own node and manage the cost of the hardware. On the other hand, he can use a third party system which have both free and paid plans such as **Infura** or **Pinata**.

We assume that user wants to store data below 1GB and he uses a free plan. By this, he needs to pay only for the data stored in Ethereum. The amount of data is:

$$\text{Amount} = \text{IPFS hash} + \text{fileNameLength} + \text{senderAddress} = 46 + 32 + 20 = 98 \text{ bytes}$$

Table 6.2: Results for the three files when combining systems

File name	File size	Transactions required	Ether required	Price (\$)	Average tx per block
test-file.txt	47 bytes	1	0,000695	0,156	140
fibonacci.txt	949 kilobytes	1	0,000695	0,156	140
test-image.jpg	8,62 megabytes	1	0,000695	0,156	140

From the above table we can clearly see that storing data on IPFS and linking it in Ethereum costs way less. Even if the amount of data increases, the paid plans are nowhere near the amount requested in Ethereum.

Moreover, the average transactions per block remain normal and they are not disruptive to the overall system. We can clearly see the benefits of this alternative both in terms of cost and impact on the network.

Chapter 7

User's manual

7.1 System requirements

A system that wants to run DRDS needs to have the following requirements:

- **operating system:** Windows (greater or equal than 8), Linux, MacOS
- **RAM memory:** at least 2GB
- **Internal memory:** at least 4GB
- **preinstalled applications:**
 - Truffle - Ganache
 - Go implementation of IPFS node & path variable to the application
 - NodeJS (comes with npm)
 - Web browser: Google Chrome, Mozilla Firefox or Brave Browser
 - Metamask extension (for Google Chrome or Mozilla Firefox)
 - In case you want to modify the code, you can use a code editor such as JetBrains Webstorm or Microsoft Visual Studio
 - source code is downloaded in a local folder

7.2 Installation steps

These steps are run only once.

1. in the application download folder, open a terminal and type *"npm install"*. This will download the dependencies required for the application to run.
2. in the application download folder, open a terminal and type *"truffle migrate --compile-all --reset"* to deploy the DRDS smart contract on the Ganache node.

7.3 Application start steps

1. open Ganache and press "Quickstart" to create a new Ethereum blockchain
2. open a terminal and enter *"ipfs daemon"* to start the IPFS node
3. in the application download folder open a terminal and enter *"npm start"*
4. after the React application starts (it will open a browser tab), accept the connection of DRDS to Metamask. From Metamask select the local network in port 7545 and log in with an account from Ganache (first account, for example)
5. at this step you should be able to see the homepage and enjoy the application

7.4 User manual

This section will present screenshots from the application that will guide the user.

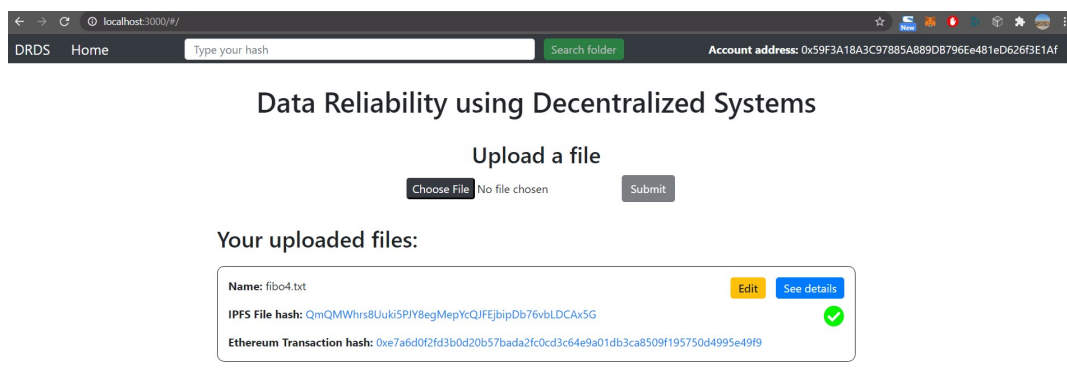


Figure 7.1: DRDS Homepage: user added files

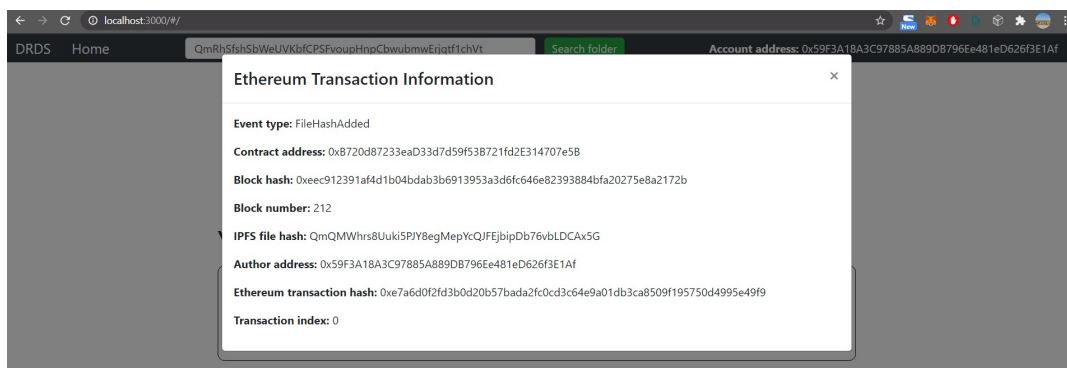


Figure 7.2: DRDS Homepage: information about the transaction

When a user wants to see details about his files:

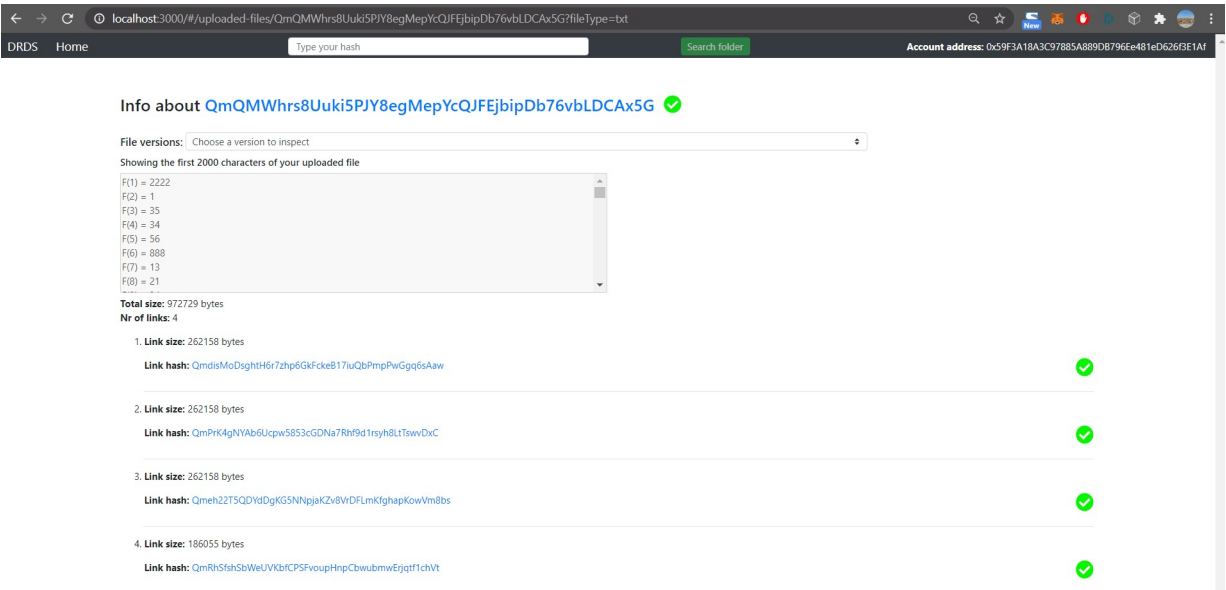


Figure 7.3: DRDS Information page: information preview and links

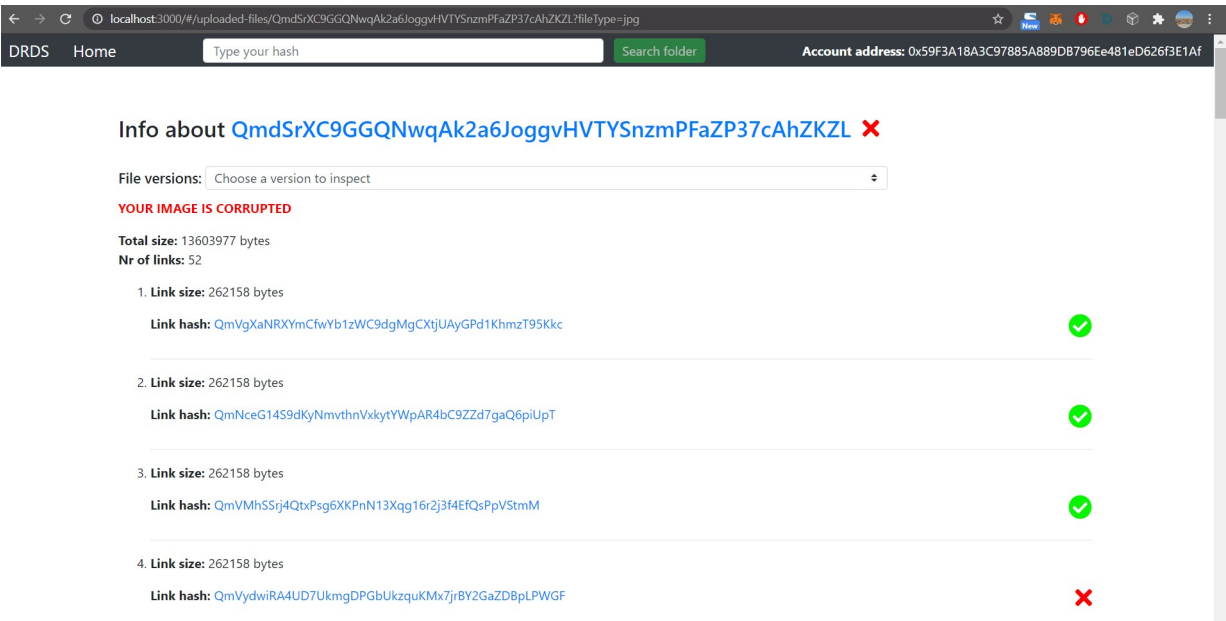


Figure 7.4: DRDS Information page: corrupted link

Chapter 8

Conclusions

This chapter concludes the presentation of the DRDS and presents the problems that were addressed throughout the system. Moreover, solutions that were proposed and implemented will be shortly discussed, together with the results that were generated. Further developments will be presented at the end of this chapter.

8.1 Addressed problem

Nowadays, to be correctly informed means to be powerful. Unfortunately, this becomes more and more difficult due to amount of information created and processed by the internet. Plagiarism and data corruption (alteration) seem to be widely present in our life and their detection may become tedious and time consuming. If you combine these issues with an authority that possesses (centralizes) all the power, you could end up in a very bad position.

By this, a solution to these above stated problems could be a decentralized system that is able to detect plagiarism and data corruption. Moreover, as data is constantly changing and updated, a versioning of it would be necessary, but only by the user that uploaded it in the first place.

8.2 Solution and personal contribution

The proposed solution was to integrate two decentralized systems (Ethereum and IPFS) in order to create a newer, better solution. These systems are not meant to work together and therefore a sort of intermediary had to be created so that the communication could be done in a flawless manner.

On top of this combination, several features have been added such as: plagiarism detection (implemented at smart contract level), data corruption (implemented on top of IPFS) detection and the possibility to version (edit) already uploaded data without losing the history of editing (implemented at the smart contract level).

8.3 Results

Regarding the results, the decentralized system integration was performed successfully and all the wanted features have been implemented. The plagiarism and data corruption detection work as expected. Moreover, each user has the possibility to edit its own files and see its editing history.

Besides this, an analysis was performed in Chapter 7 regarding the cost of storing data. One solution was to do so using just the Ethereum blockchain, but we observed that it was unfeasible even with a few hundreds of kilobytes, reaching prices of hundreds of dollars at the current time of writing.

However, when combining Ethereum with IPFS, we have seen that the price drops dramatically (under \$1 per file), no matter the type of file or its size. This was due to the implementation of IPFS and the way it creates a hash (an unique identifier) of data using the CIDv0 algorithm.

8.4 Further improvements

Regarding future improvements to this project, someone could do the following to improve the actual system:

- **detect file / link deletion:** DRDS is able to detect whether certain file was altered, but when it comes to deletion, it remains stuck, as it tries to load a file or link that does not exist, even though it is marked as present
- **Metamask connection:** at the application start, Metamask asks the user directly if he wants to connect its wallet to DRDS. This should be done, as stated in Metamask documentation using a button, not at system start
- **network configurations:** DRDS connects to a local IPFS node and Ganache using hardcoded configurations. An improvement would be the ability to select the IPFS network to be used (local, infura or pinata). Regarding the blockchain network, Metamask allows the user to select these features at runtime, making the process more configurable

Bibliography

- [1] S. Sam, “Types of databases,” 2018. [Online]. Available: <https://www.tutorialspoint.com/Types-of-databases>
- [2] Wikipedia, “Centralized database,” 2020. [Online]. Available: https://en.wikipedia.org/wiki/Centralized_database
- [3] W. EN, “Distributed database,” 2020. [Online]. Available: https://en.wikipedia.org/wiki/Distributed_database
- [4] Wikipedia, “Cloud database,” 2020. [Online]. Available: https://en.wikipedia.org/wiki/Cloud_database
- [5] P. Labs, “Ipfs powers the distributed web,” 2020. [Online]. Available: <https://ipfs.io/>
- [6] T. Terade, “What is decentralized storage?” 2018. [Online]. Available: <https://medium.com/bitfwd/what-is-decentralised-storage-ipfs-filecoin-sia-storj-swarm-5509e476995f>
- [7] S. Labs, “Decentralized cloud storage is here,” 2020. [Online]. Available: <https://storj.io/>
- [8] Swarm, “Swarm storage and communication for a sovereign digital society,” 2020. [Online]. Available: <https://swarm.ethereum.org/>
- [9] V. Tron, “Swarm alpha public pilot and the basics of swarm,” 2016. [Online]. Available: <https://blog.ethereum.org/2016/12/15/swarm-alpha-public-pilot-basics-swarm/>
- [10] Sia, “Decentralized storage for the post-cloud world.” 2020. [Online]. Available: <https://sia.tech/>
- [11] L. Tech, “Sia - decentralized cloud storage,” 2018. [Online]. Available: <https://medium.com/luxor/sia-decentralized-cloud-storage-7de576542497>
- [12] K. Shirriff, “Hidden surprises in the bitcoin blockchain and how they are stored.”

- [13] D. Bradbury, “Bitcoin core development update 5 brings better transaction fees and embedded data,” 2013. [Online]. Available: <https://www.coindesk.com/bitcoin-core-dev-update-5-transaction-fees-embedded-data>
- [14] M. Zuidhoorn, “Why do we need transaction data?” 2019. [Online]. Available: <https://medium.com/mycrypto/why-do-we-need-transaction-data-39c922930e92>
- [15] A. I. GmbH, “Acronis notary: a new way to prove data authenticity via blockchain,” 2020. [Online]. Available: <https://www.acronis.com/en-us/articles/data-protection/>
- [16] K. S. Nishara Nizamuddin, Haya Hasan, “Ipfs-blockchain-based authenticity of online publications,” 2018. [Online]. Available: https://www.researchgate.net/publication/325899234_IPFS-Blockchain-Based_Authenticity_of_Online_Publications
- [17] R. B. F. L. A. M. E. Gaetani, L. Aniello and V. Sassone, “Blockchainbased database to ensure data integrity in cloud computing environments,” 2017. [Online]. Available: https://www.researchgate.net/publication/318223974_Blockchain-based_database_to_ensure_data_integrity_in_cloud_computing_environments
- [18] Wikipedia, “Blockchain,” 2020. [Online]. Available: <https://en.wikipedia.org/wiki/Blockchain>
- [19] A. M. Antonopoulos, *Mastering Bitcoin: Programming the Open Blockchain*. O’Reilly Media, 2017, vol. 2.
- [20] Wikipedia, “Ethereum,” 2020. [Online]. Available: <https://en.wikipedia.org/wiki/Ethereum>
- [21] G. W. P. D. Andreas M. Antonopoulos, *Mastering Ethereum: Building Smart Contracts and DApps*. O’Reilly Media, 2018, vol. 1.
- [22] V. Buterin, “Merkling in ethereum,” 2015. [Online]. Available: <https://blog.ethereum.org/2015/11/15/merkle-in-ethereum/>
- [23] Wikipedia, “Patricia tree,” 2012. [Online]. Available: https://en.wikipedia.org/wiki/File:An_example_of_how_to_find_a_string_in_a_Patricia_trie.png
- [24] ProtoSchool, “Anatomy of a cid,” 2019. [Online]. Available: <https://proto.school/#/anatomy-of-a-cid>
- [25] X. P2P, “Kademlia - routing table,” 2017. [Online]. Available: https://xorrip2p.github.io/public/images/kademlia_routing_table_in_tree.png
- [26] J. Benet, “Ipfs - content addressed, versioned, p2p file system,” 2020. [Online]. Available: <https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.p>