TÉCNICO LISBOA

# P2CSTORE: P2P and Cloud File Storage for Blockchain Applications

**Marcelo Filipe Regra da Silva**

Thesis to obtain the Master of Science Degree in

## Information Systems and Computer Engineering

Supervisors: Prof. Miguel Ângelo Marques de Matos
Prof. Miguel Nuno Dias Alves Pupo Correia

### Examination Committee

Chairperson: Prof. Francisco António Chaves Saraiva de Melo
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Alysson Neves Bessani

**January 2021**

## Acknowledgments

I would first like to thank my supervisors Miguel Matos and Miguel Correia for the opportunity of doing this dissertation and learn from their knowledge and experience. Thank you for the discussions, for constantly challenging me when I needed and for making me set my own goals and rising my standards. I want to thank David Matos as well for assisting with the Cloud component of my thesis.

This research work is the last step of a long journey, the climax of years of hard work, learning, persistence, and motivation. I would like to thank my family, father, mother, and sister, for the inspiration, support, and patience throughout this journey. Without them, this would certainly not be possible. I would also like to thank my friends at Técnico for the support and company during these five years of hours of study, failures, successes, and learning. They were a vital part of my success. Finally, I would like to thank my girlfriend for being side by side with me, supporting me every day for the past five years, enduring my long coding nights, my absence due to study or projects, and the days in which I was grumpy and cranky.

You all combined made this possible, I share this success with you all, thank you.

## Resumo

A blockchain está a revolucionar o mundo. A blockchain consiste num registo de transações distribuído que pode ser usado para vários propósitos, por exemplo, para processamento de pagamentos, transferências de dinheiro, votação digita, armazenamento de dados, entre outros. É especialmente interessante usar a blockchain para armazenar dados, pois é um registo imutável que garante propriedades de segurança desejáveis. Porém, na tecnologia blockchain atual, continua a ser um problema o facto de que esta só poder armazenar, de maneira eficiente, dados com tamanhos pequenos. Também é relevante mencionar que blockchains públicas como o Ethereum cobram aos utilizadores por cada byte armazenado, tornando caro o armazenamento de ficheiros de grandes dimensões. Para resolver esse problema, propomos P2CSTORE, um novo sistema de armazenamento para aplicações blockchain usando subsistemas P2P e computação na nuvem. Dessa forma, pretendemos fornecer aos programadores de aplicações a flexibilidade de escolher o melhor local para seus ficheiros. O uso da blockchain para armazenar ficheiros, por exemplo certificados educacionais permite uma melhor avaliação da autenticidade desses ficheiros. A aplicação armazena *hashes* dos certificados na *blockchain* e os próprios certificados no nosso sistema de armazenamento. Com esta solução, podemos ter um sistema de armazenamento que funciona em cima qualquer blockchain. Os resultados obtidos neste trabalho reforçam o facto de ser relevante criar tal sistema com as duas possibilidades P2P e computação em nuvem. Isto verifica-se pois existe um bom desempenho em ambos, fazendo com que uma combinação seja possível e eficiente, aumentando a disbonibilidade sem comprometer o desempenho so sistema.



**Palavras-chave:** Armazenamento Distribuido, Blockchain, Certificados de Educação, Segurança, Confiabilidade, Disponibilidade

## Abstract

Blockchain is currently revolutionizing the world. A blockchain consists of a distributed ledger of transactions that can be used for several purposes, such as payment processing, money transfers, digital voting, data storing, among others. It is interesting to use blockchain to store data because it is an immutable ledger that will ensure desirable security properties. However, in blockchain technology, it remains a problem the fact that it can only store, in an efficient way, data with small sizes. Public blockchains like Ethereum charge the users per each byte stored making it expensive to store large files. To tackle this problem we propose P2Cstore, a new storage system for blockchain applications using both P2P and cloud subsystems. This way we aim to provide application developers with the flexibility of choosing the best place for their files. The usage of blockchain to store files, for example, education certificates allow a better authenticity assessment of these files. The application stores hashes of the certificates in the blockchain and the certificates themselves in our storage system. With this solution, we can have a storage system that works on top of any blockchain, that can store files of arbitrary size. Our evaluation corroborates the fact that such a system with both P2P and cloud is relevant and it works. This is the case because the obtained results outlined a good performance for both P2P and cloud. Therefore, a combination of both is possible and efficient, but it also improves availability without compromising system performance.

**Keywords:** Distributed Storage, Blockchain, Education Certificates, Security, Reliability, Availability

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Nowadays *data* is an important component of our world. We generate vast amounts of data daily, such as application logs, browser search history, medical records, education certificates, photos, among many other items that must be properly stored. The current best way to store files is by using distributed systems. This is the case because these system's components are located in different machines that communicate through a computer network by message passing. This makes these systems the best option for storage systems because by being distributed it is possible to scale the system by adding new machines, making it possible to scale almost infinitely. A *blockchain* [Und16, Pec17] is an example of a distributed system that can be used for storage purposes. Blockchain is a promising new technology that is generating a vast interest world-wide, Portugal included [Pec17]. A blockchain is an ordered log of transactions that runs in several nodes connected over the Internet. It is possible to append new transactions to the log, but not modify those already logged.

There are two types of blockchains, permissionless and permissioned. In permissionless blockchains, no one controls the blockchain. Everyone can contribute to the blockchain and hold a copy of the public ledger. In permissioned blockchains, an entity or set of entities decide who can perform transactions. Bitcoin and Ethereum are both permissionless blockchains, but Ethereum has the capacity of running code with smart contracts. Ethereum smart contracts are programs that can be executed by the Ethereum Virtual Machine.

Blockchain technology has several applicabilities such as payment processing, money transfers, digital voting, immutable data backup, data storing, among others. In our work, we will be focusing on the storage capacity of the blockchain. It is interesting to use the blockchain for storage because it can store certificate hashes that can later be used to enforce the authenticity of the associated certificate. After all, if a node attempts to store something it first must make a valid transaction on the blockchain, which ensures several properties like integrity, authenticity,

non-repudiation, all of which will be discussed in the next section.

As an example, project QualiChain (https://qualichain-project.eu/) is developing a blockchain-based system to enforce the authenticity of university certificates, [SVGC20, MTD19, KMK+19, KVH20]. The idea is to store certificate data in a blockchain, in such a way that when a company receives an application for a given position, it can check with the blockchain if the candidate certificate is authentic. A natural approach would be to store the certificates in the blockchain, but certificates are reasonably large (in the order of a few megabytes). Public blockchains like Ethereum charge the users per each byte stored making them expensive to store large files, so storing all certificates from all users in the blockchain would be very expensive.

Another approach, often used in blockchain-based distributed applications, also known as *DApps* [AW18], is to use an external storage system to store data. These applications typically store the hash of the file on the blockchain and the actual file in a separate storage system. *DApps* and smart contracts are very similar but should not be considered the same thing. A smart contract is a program that consists of an agreement between parties that is automatically executed once a transaction triggers it. *DApps* however, are decentralized applications that are executed in a decentralized peer-to-peer network.

To create a storage system for DApps, there are two common architectures: *peer-to-peer (P2P) and centralized storage. DApps often use as external file storage system a P2P file system* [Ben14, VC14, WBBB14, BG18, Swa19, WLB14], as many blockchains are also P2P systems [Nak08, Woo14].

There is a set of properties that we must have in consideration for a storage system. These properties are:

1. Availability, a system is said to be available if it can ensure that a certain file will be available at any given time.

2. Data integrity, consists of the capacity of the system to ensure that the data is consistent and has not been corrupted.

In a P2P system, there is no centralized server. In such a system, some nodes share resources to store files. If a node wants to get a particular file, it can request it from the network. These systems are typically based on volunteer nodes and therefore can be used free of charge. One example of such a system is the IPFS [Ben14] that will be described later in this document.

P2P systems have advantages when compared to other kinds of systems, for instance, these systems are easy to set up because there is no need for management servers. A P2P network system is less expensive as most of the nodes are personal computers, and each time one node

fails, the repair is not costly for the system. Since all nodes work as providers (servers) and clients the necessity of a centralized system does not exist.

Although P2P systems have advantages they also have some disadvantages. In a P2P system, most nodes are personal computers, therefore it is difficult to ensure data integrity. Availability is also harder to ensure in such systems. It is difficult in P2P systems since nodes can leave and join the network at will, making the files they store available or unavailable as they join and leave respectively.

This set of advantages make the peer-to-peer system an interesting approach for some types of applications. An example would be if one wants an application that has a high number of users using the system at the same time. In this architecture, the system scales as more users join in. The system could benefit from the high number of users to store data or execute computation, allowing it to grow in storage capacity and computation power as more users join the system. Also because more users are online at the same time the probability that a certain file is available is greater than if a small number of users were online at the same time.

There is also the centralized architecture in which client computers connect to a central server over the Internet. This client-server architecture allows providing high-availability by replicating the server and/or disaster recovery by doing backups and similar mechanisms. Today, with the popularity of the cloud model [AFG+10], this architecture is provided by many *cloud storage services* [Amaa, C+11, Gooa, Drob], that ensure high-availability but charge for bytes stored and downloaded.

The advantages that the centralized server architecture has when compared to P2P systems are, for example, the fact that there is centralized control. Servers can control access rights to resources. The allocation and distribution of resources across servers are also centrally managed. Due to this centralization, the management is also easier as all files are stored in the same place, being easy to find and store them. The client-server architecture also has the advantage of back-up and recovery, as all data is stored on a central server which could create back-up mechanisms for data loss issues.

The centralized storage approach has also several disadvantages. In this type of architecture if a lot of clients start making requests to the same server at the same time it could get slow, congested, and could even make the system not be able to supply the service. This problem could also be exploited by attackers to perform DDoS (Distributed Denial of Service) attacks.

When considering the advantages and disadvantages of both P2P and centralized server architecture, it is possible to understand that both have their merits and therefore they can both be used to create a storage system for DApps. Considering this we decided to concile

both solutions to create a more robust, highly available storage system. In sum, this work will leverage ideas from these two types of architectures — those based on P2P and those on clouds.

This work was used to develop an article *P2CSTORE: P2P and Cloud File Storage for Blockchain Applications* for The 19th IEEE International Symposium on Network Computing and Applications (NCA 2020).

## 1.1 Objectives

Our general goal is to create P2Cstore, a distributed file system for DApps that leverages the two types of distributed file storage systems previously described and allows developers to select among a wide range of configurations with different trust, cost, and availability trade-offs. To guarantee the high availability and integrity of the files we will leverage replication techniques. We will also use a Proof-of-Storage (PoS) mechanism to ensure that each system node is storing the content as promised. The PoS mechanism allows a client to check if a server has a file without downloading that file. We will use the storage of university certificates as a use case.

## 1.2 Thesis Outline

The remainder of the document is structured as follows. In Chapter 2 we discuss related work that is relevant to our solution. Chapter 3 describes the system implementations, in particular, the general architecture of the solution, the rationale behind our choices, the proof-of-storage algorithm, and several communication protocols. Chapter 4 evaluates the solution that we propose and discusses the associated results. Finally, Chapter 5 briefly concludes this thesis and proposes for future work.

# Chapter 2

# Background

The development of a storage system for blockchain applications using both P2P and Cloud storage providers can be analyzed under three main categories, the blockchain, the storage system, both P2P and Cloud providers, and finally the combination between both blockchain and storage systems.

P2Cstore is a storage system for blockchain applications. The design and development of this work was inspired by several other storage systems, in particular DepSky, IPFS, UniDrive, Storj, FileCoin [BCQ+13][BG18][WBBB14][TLS+15][Ben14].

The remainder of this chapter presents the work done by previous research that influenced the design and development of P2Cstore. It also presents the relationship of this work with our solution, P2Cstore, to contextualize it.

This chapter has the following structure. Section 2.1 introduces blockchain technology. In particular, it is focused on Ethereum and smart contracts as well as some of the blockchain properties that are interesting to our work. Section 2.2 presents several storage systems, both P2P and cloud-based storage systems. In Section 2.3 we discuss storage systems for the blockchain. In Section 2.4 we introduce Proof-of-Storage algorithms and describe why they are interesting. Finally, A description of the concepts discussed in this chapter is given in Section 2.5.

## 2.1 Blockchain

When blockchain started with Bitcoin [Nak08] the goal was to create a new way of making transactions between entities without the necessity of a middle-man. However, blockchain has evolved a lot since then.

Nowadays we have a great number of blockchains like Bitcoin, Ethereum, among many more. All of them share a set of characteristics that were first introduced and proposed with Bitcoin.

Blockchains are P2P networks connecting all participants and propagating the updates made to the network, like transactions and blocks of transactions that were verified by miners[Her19]. These miners consist of computers with high-performance graphics processing units. These miners use a set of consensus rules to make the verification of whether a certain transaction is valid or not, meaning that they add a transaction to a candidate block and then attempt to find a *Proof-of-Work* (PoW) that makes the candidate block valid to add it to the blockchain [Her19]. Proof-of-work mining consists of having a certain node execute a certain amount of computational work, like solving a puzzle, for a reward. These updates are all verified based on a standardized gossip protocol.

This gossip communication protocol works based on the way epidemic virus spread. By doing so P2P distributed systems ensure that data is disseminated to all members of a group. This protocol allows transaction propagation through a simple process. In Ethereum, the transaction propagation starts with the originating node creating a signed transaction. Then it is validated and transmitted to all nodes directly connected to the originating node. These directly connected nodes are called neighbors. Each one of the neighbors validates the transaction and transmits it to their set of neighbors, except the one they received the transaction from. This way the transaction will spread across the network at a rapid pace, like an epidemic virus.

In general, we can define blockchain as a timestamped, distributed, and secure database that stores transactions between nodes in the network [Nak08]. These transactions consist of signed messages that consist of the transfer of cryptocurrencies from one account to another. When verified and accepted, transactions create a chain of cryptographically secured blocks that acts as a ledger. The blockchain also preserves the integrity of the network.

### Permissioned and Permissionless Blockchains

Blockchains can be permissionless or permissioned. Permissionless blockchains are characterized by having no owner. Everyone can contribute and hold a copy of the public ledger. To ensure the integrity of the ledger, nodes reach a consensus between each other to have a single state among them all. Any user can join or leave the network when he wants [Pec17].

Ethereum is an example of a permissionless blockchain [AW18]. It is a public ledger, therefore all transactions that happen must be visible to everyone to be verified. Also, no part of the Ethereum protocol involves encryption besides the digital signing transactions. This means that all communications with the Ethereum platform and between nodes are not encrypted and can be read by anyone.

In permissionless blockchains, we have a public and transparent ledger where everyone can

interact and contribute to it without a third party having to allow and control everything, instead it is the network of several peers that together make and control the network.

There also exist permissioned blockchains. This type of blockchain was created to solve another problem. Financial institutions soon noticed the growing popularity of permissionless blockchain. However, they realized that this type of blockchain had a public way of operating that did not correspond to their needs. They needed to ensure privacy for their client's data while controlling the network, who participated in it, and what each participant could do.

In this type of blockchain, the processing of transactions is done by a well-defined set of identified entities. Also, all data in the system is visible only to a limited set of entities, in the case of financial institutions only they can see the user's data and not everyone as it happens on permissionless blockchains.

Permissioned blockchains are intended to be used by entities that have some kind of trust in each other. Therefore, the entities able to add new blocks to the ledger are predefined. This way it becomes unnecessary to do proof-of-work mining to ensure consensus because it is not a public ledger.

This way institutions can control the blockchain better, meaning that they can allow limited read access to transactions and block headers to their users (clients). This way they can assure the safety of their client's funds in a more transparent and trustworthy way.

For our work, the permissionless blockchain is more appealing because we want the ledger to be public, we want for everyone to be able to participate in the network equally. Therefore we will focus more on Ethereum, which, as said before, is a permissionless blockchain.

**Ethereum**

All nodes are equal peers in the Ethereum network, however, some of them are operated by miners. To prevent attacks on the network, blockchains in general also use PoW, associated with a monetary incentive to all participants that will prefer to solve the challenge and earn the money for solving it, instead of attacking the network. PoW consensus has a significant drawback, which is the cost of electricity required to solve it, considering that it is like a service provided by the miners, they give electricity and computer power and, when they succeed to add a new block, they receive money incentive.

This innovation that first started with Bitcoin soon was noticed by enthusiasts, and new applications for this new technology started to be considered and developed by the community, other than for cryptocurrencies. They wanted to use blockchain for things like storage, digital voting, among other applications, however, this was not yet possible with Bitcoin. At first,

developers thought it was a good idea to build on top of Bitcoin, but soon they realized that this would be too challenging, and Bitcoin was too limited for the potentiality of the blockchain technology. Considering this a new blockchain soon started to be developed, Ethereum, based on Bitcoin and Mastercoin [But13]. Ethereum was built to be capable of executing code.

Bitcoin tracks the ownership and state of bitcoins. Ethereum tracks the state transitions of a general-purpose data store. This data store can hold any kind of data because it is a key-value tuple. Because of its storage capacity, Ethereum can, like a normal computer, load code into its Ethereum Virtual Machine (EVM), and run that code. The resulting state changes are stored directly into the blockchain. These EVM programs are called *smart contracts*, which will be discussed in more detail next.

Ethereum is Turing complete, unlike Bitcoin. A system is said to be Turing complete if it can be used to simulate any Turing Machine. As demonstrated by Turing, it is impossible to prove whether a program will terminate or not without running the actual program. Therefore Ethereum can not predict if a smart contract will terminate, or the time it is going to run, without actually running it. This could be exploited to create a Denial of Service (DoS) attack. In this type of attack, the goal is to make the system unavailable, preventing the system to provide the service. This attack could consume a lot of CPU power, memory and could compromise the functioning of the Ethereum network.

To tackle this problem, Ethereum implemented a measuring mechanism called *gas*. This mechanism works by having each instruction cost a certain amount of gas. When a transaction triggers the execution of a smart contract it must include a maximum gas cost that can be consumed while running the smart contract. The execution will be terminated by EVM if the gas limit is reached. The gas can be bought only with ether, the Ethereum coin. Because of gas, the smart contract execution is limited, meaning that it will run until the maximum gas limit is reached, protecting the network in the case that the program never terminates, preventing this kind of DoS attack.

As Ethereum developed, it became able to store data and execute programs, people soon started to realize that they could build Decentralized Applications DApps. With DApps, Ethereum became a platform for programmers. DApps are composed of a web frontend user interface and smart contracts on a blockchain [AW18].

Blockchain started to be considered as a storage mechanism, secure, and highly available. The problem was that the blockchain could only store small amounts of data. Also, the cost of storing a 256bit on the Ethereum network is 20000 gas. The cost of gas at the time of writing this document is 105 Gwei and each Gwei is 0.000000001 ETH [Woo14, etha]. We can easily

calculate the cost of storing a file with 1KB [ethb]. Considering that 256bit costs 20000 gas then 1KB costs 80000 gas which gives us a transaction fee (Fiat) of $3.00352. Having this into account it became necessary to develop a new storage system, to store more information while keeping all of the great properties of the blockchain.

It is important to mention that no part of the Ethereum protocol involves encryption. This means that all communications with the Ethereum platform and between nodes are not encrypted and can be read by anyone. This way it is possible for everyone to verify the correctness of state updates and to reach consensus.

However, cryptography still is an important part of the Ethereum protocol. For instance, ownership of ether by Externally Owned Accounts is established through a set of cryptography mechanisms like digital private keys, Ethereum addresses, and digital signatures.

Externally Owned Accounts or EOA is a type of account that has an ether balance, it can also send transactions, it is controlled by a private key and it does not have any associated code. Therefore these accounts are controlled by anyone who has the private key.

### Smart Contracts and Security on Ethereum

The Ethereum protocol has two possible types of accounts, *externally owned accounts* (EOAs) and contract accounts. EOAs are controlled by users and have a key pair associated. As opposed to EOAs, contract accounts do not have a public key. Therefore, contract accounts, are fully controlled by what was coded in the program. The *Ethereum virtual machine* (EVM) is where the program code is executed. These types of programs are also known as smart contracts. It is important to notice that the contract accounts can also have data storage associated with the code.

In the context of this work, the term smart contract is used to describe immutable computer programs that run in a deterministic way in the context of an EVM on the decentralized Ethereum network. They are said to be immutable because once deployed on the Ethereum network the code of the smart contract cannot be changed directly, the only way to modify it is to deploy a new instance.

These smart contracts are written using high-level languages, for example, Vyper [vyp] and Solidity [sol]. But, the EVM does not understand high-level languages, therefore for the smart contracts to run on the EVM, they must be first compiled to bytecode. Once these smart contracts are compiled, they are deployed on the Ethereum network using the contract creation transaction. These types of transactions are special because they go to the contract creation address 0x0.

When a transaction like this occurs the created contract account gets an Ethereum address generated from the transaction which will identify the contract account from that moment on, meaning that if someone wants to execute a function of that contract or send funds to it, it will have to make a transaction to this address.

All contracts on the Ethereum network, as said before, are inactive until a new transaction initiated from an EOA triggers it to execute something. However, a contract can still call another contract, and this contract could call another contract, and so on, but the first contract in a chain of contracts must always be called by an EOA transaction.

When writing smart contracts security is a concern. In the field of smart contract development, a simple mistake is costly and easily exploited, and usually, there is no way to recover the money that was lost. A smart contract executes exactly what is written for it to do. However, sometimes the programmer codes something that he did not intend, generating bugs and vulnerabilities. Considering that all smart contracts are public and that any user can interact with them by issuing transactions, any existing vulnerability can be exploited by everyone. Therefore it becomes of major importance to follow best practices and use well-tested design patterns when developing smart contracts. The security best practices considered are the ones common to most software development areas, as secure development is of great importance to all of them. They are, for example, defensive programming, minimalism/simplicity, code reuse, code quality, readability, and test coverage [AW18].

## 2.2 Storage Systems

We live in an era where storage systems are of major importance. Today we generate a lot of data, such as application logs, browser search history, medical records, education certificates, among many others. The current best approach to store such data in a fast, secure, and highly available way is through the usage of distributed storage systems.

Distributed storage systems store data on a set of multiple servers, which behave as a single storage system. In our work, we will focus on P2P storage systems and cloud storage systems, which we discuss next.

#### Peer-to-Peer Storage Systems

P2P systems represent a paradigm of distributed systems whereas data and computational resources are contributed by many hosts that are trying to provide a uniform state. In this type of system, all nodes contribute with computational resources like disk storage and CPU power. Next, we will look in more detail into the IPFS P2P storage system [Ben14]. We chose

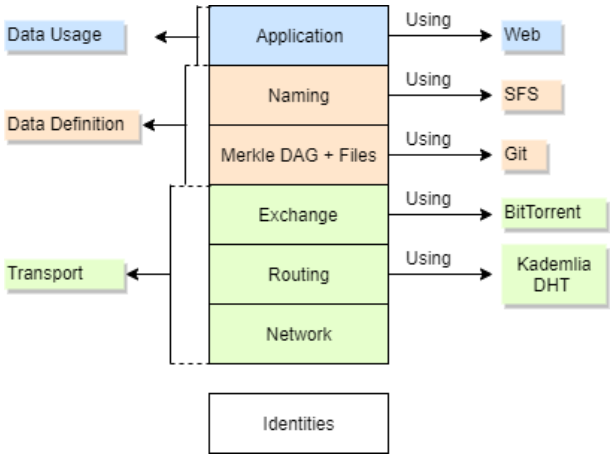


Figure 2.1: Overview of the IPFS Protocol stack of sub-protocols

to analyze this system because it is known and representative.

### IPFS Storage System

As time passed more applications, in particular, P2P distributed file systems, appeared, for example, AFS, Coral, among many others. For our work, we will be focusing on the InterPlanetary File System, also known as IPFS [Ben14].

IPFS consists of a P2P distributed file system that tries to use the best techniques and algorithms used by the existing and proven P2P distributed file systems, like distributed hash tables (Kademlia), block exchanges (BitTorrent), versioning control systems (Git), and self-certified filesystems (SFS). Even though a lot of systems reached a great number of users we still do not have today a "general file-system that offers global, low-latency and decentralized distribution" [Ben14]. Therefore, the idea is to create a file system to connect all computing devices with the same system of files, on a global scale.

IPFS is divided into a stack of sub-protocols where each of the layers is responsible for a different functionality within the system.

As we can see in Figure 1 there are seven layers. The Application layer is not part of the scope of our work and, therefore, it will not be covered. The identities layer is separated because it is not a direct part of the stack, it is instead a parallel sub-protocol that is fundamental to the proper functioning of the IPFS Protocol.

The Identities layer corresponds to the layer where each node that participates in the network gets identified. In this protocol, the nodes are identified by a NodeId. The NodeId consists of a unique fingerprint, given by a cryptographic hash of the public key of the respective node. This cryptographic hash is created using S/Kademlia's [BM07] proof-of-work crypto puzzle. Each node has to create an asymmetric key pair and store their public and private keys encrypted

with a passphrase. The nodes can instantiate a new node on every launch, however, they are incentivized to remain the same. Otherwise, they lose accrued network benefits. Upon creation, nodes exchange public keys and verify if the other's NodeId equals the hash of the other's public key, and only if these values are the same the connection remains, otherwise it is terminated.

Nodes in a P2P network need a way of communicating with each other, for that there is the Network layer. The IPFS Network layer offers the most important features, like transport, reliability, connectivity, integrity, and authenticity. IPFS does so by using existing technologies such as uTP [SHI+12] for transport and HMAC for authenticity. IPFS can be customized to the particular needs of the system, meaning that the protocols and technologies used in this layer can be adjusted and changed.

The Routing layer is required for the IPFS nodes to find peers who have particular objects. A *Distributed Sloppy Hash Table* (DSHT) based on S/Kademlia and Coral [FFM04] is used. The DSHT allows for an efficient lookup while making efficient use of storage and bandwidth. Based on DSHT, IPFS nodes can lookup other nodes by a decentralized clustering algorithm where nodes can find each other and form clusters according to region and size. IPFS DSHT is also used to find which nodes have some specific data by referring to the multihash of that data, meaning that if one node wants some data it asks the DSHT to find what are the nodes that can serve its request. What is stored on the DSHT are references, which are the NodeIds of the peers that have the data. Although the default is a DSHT, the routing system of IPFS can be changed according to the necessity of the application.

To exchange data IPFS uses a protocol called BitSwap, which is similar to BitTorrent. In BitSwap data is broken down into blocks. Each participant has a set of blocks they want and a set of blocks they have to offer. BitSwap works like a marketplace where peers exchange blocks with each other. In BitSwap nodes exchange blocks regardless of what files those blocks are part of, meaning that blocks are traded with all IPFS nodes. Blocks could come from totally unrelated files in the file system. Sometimes nodes do not have anything to exchange in the BitSwap marketplace but still need some blocks. In this case, nodes must work to get the blocks they want. What happens is that a node that wants some block but has nothing to trade, will seek the pieces its peers want, with lower priority when compared with the blocks the nodes want for themselves. This way nodes are incentivized to cache blocks because if they do cache blocks they will have a greater capacity to trade when they need to get some blocks from the marketplace. This works as an incentive for nodes to seed even when they do not need anything.

There are different strategies possible for BitSwap peers to employ. One example is by implementing a credit-like strategy. In this strategy, peers track their balance with other nodes

and peers send blocks in a probabilistic way to debtor peers according to a function, in this example lets use the debt ratio function. In this function as a node gets more debt the probability that it will receive the blocks that it wants reduces. As opposed to when a node gets more credit, by sharing blocks with other nodes, it will increase the probability of receiving a block it wants. It is important to mention that if a node decides not to send a certain block to a peer it ignores that peer for some time, this way free-loading, which is when some peers use the system without given anything back, can be avoided while allowing efficient data propagation. BitSwap nodes keep a ledger that tracks exchanges between other nodes. This way nodes can keep track of transaction history and avoid tampering.

The DSHT and BitSwap allow IPFS to form a P2P system that stores and distributes blocks in a fast and robust manner. On top of these layers, IPFS builds a Merkle Directed Acyclic Graph (Merkle DAG). Merkle DAG is a merge of Merkle trees with DAG. Merkle trees ensure that blocks exchanged on the P2P network are correct, unaltered, and not damaged in any way. They do this by using cryptographic hash functions to organize data blocks. They label leaf nodes with the hash of a data block, and then the non-leaf nodes are labeled with the combination of the labels of the previous nodes. Directed Acyclic Graph or DAG consists of a type of graph where there are no cycles, meaning that it is not possible to start from any vertex $v$ and follow a consistently-directed sequence of edges and get to $v$ again. The combination of these two structures makes Merkle DAG a data structure where hashes are used to reference data blocks and objects in a DAG. This DAG is a distributed, authenticated, hash-linked data structure that provides useful properties including, content addressing. All content is uniquely immutably identified by its multihash checksum, even links, tamper resistance. All content is verified with its checksum, if data is altered in any way IPFS can detect it, and deduplication if objects hold the same content. Therefore files that have the same hash, are stored only once.

On top of Merkle-DAG, IPFS also implements a model for file system versioning. The IPFS version control system has four major components:

1. Blobs, which represent a file and are objects that contain an addressable unit of data.

2. Lists, these types of objects represent a large or deduplicated file composed of several blobs concatenated together. Lists contain an ordered sequence of blobs or list-objects.

3. Tree, which represents a directory, a map of names to hashes can contain blobs, lists, or other trees.

4. Commit object, which represents a snapshot of the version history of any object. It can also reference any type of object.

IPFS Merkle DAG plus the version control system allows large files to be divided into small chunks.

Up until now, the IPFS stack forms a P2P block exchange where data is content-addressed and every object alteration changes its hash. It works fine to publish and retrieve immutable objects. It even tracks the version history of objects. However, we can not communicate new content and update it. It is still needed some way to retrieve a mutable state on the same path, meaning change objects while keeping the same path. To solve this problem, IPFS builds on the concept of a Self-certifying File System (SFS) by creating the InterPlanetary Name Space. Nodes can publish objects to their path which references their namespace, but they have to sign in with the user's private key to verify the object's authenticity. Considering that every link to one namespace has a NodeId on the path, it is not very user-friendly, meaning that it can be hard for a user to memorize a URL with a NodeId with a large set of characters, therefore IPFS uses some techniques to tackle this issue. As proposed by SFS, users can link with other users' objects directly into their own, this way it becomes more human-friendly. It is also possible to use URLs. For instance, the content is published as an immutable object and then its hash is published as a metadata value on the routing system, similar to the way DNS and URLs work.

In conclusion, IPFS consists of a highly available, robust, and secure P2P distributed storage system. IPFS solves some of the major problems of P2P storage systems as it has a great incentive for nodes to store data even when they do not need it, which helps to tackle the problem of unavailability. It is encrypted and uses cryptographic hashes for integrity checks, tackling the issue of integrity verification on untrusted nodes. However it has some disadvantages, for instance, it is likely to be more unavailable when compared to a central storage system. It could also be slower as the bandwidth speed depends on the nodes storing the data.

### Cloud Storage Systems

Cloud storage systems follow the classic client-server architecture. In this type of architecture, a server, or set of servers, hosts, delivers and manages services and most of the resources that are going to be consumed by the client [CDK05].

Cloud storage system clients are using these systems to store their data in a secure, highly available centralized server or cluster of servers. Clients are using the computation power and distribution capacity of remote storage systems to store their data, this way clients do not need to be concern about faults in communication, replication, encryption, decryption, and many other distribution and security issues because cloud storage systems take care of that for them.

Since the first cloud storage system many more appeared. Nowadays we have a lot of systems

like these such as, Dropbox, Onedrive, Google Drive [droa] [one] [goob] among many others. They all use client-server architecture. For our work, we will be focusing on two particular systems, DepSky and UniDrive. We chose to analyze these systems because they are known and representative.

### DepSky Storage System

DepSky is a cloud storage system that improves security, in particular integrity, confidentiality, and availability of information on commercial clouds [BCQ+13]. Information is stored through encryption, encoding, and replication of data across several clouds, creating a cloud-of-clouds.

DepSky addresses four major limitations of commercial clouds, which are loss of availability, which consists of temporarily unavailable connectivity, loss, and corruption of data, loss of privacy, because cloud providers have access to both data and metadata, and vendor lock-in, this is a problem because some cloud computing providers became dominant. DepSky tackles these issues by exploiting replication and diversity to store the data on several clouds (loss of availability). They use Byzantine fault-tolerant replication to store data on the clouds (loss and corruption of data). They employ a secret sharing mechanism and erasure codes to prevent the need to store clear data on the untrusted clouds (loss of privacy). And finally, they use a combination of cloud storage systems to prevent vendor lock-in.

In Figure 2.2 we have a diagram of the Depsky architecture, as we can see we have a set of clients that connect to the Depsky cloud of clouds to store and read data.

In DepSky the authors decided to use storage cloud services that do not allow the execution of users' code. Therefore the DepSky software library is implemented in the clients. This library offers an interface to object storage. It offers an abstraction to the clients to communicate with the clouds, allowing read and write operations.

The software library has to handle the heterogeneity of interfaces of each cloud provider. The data model allows for ignoring these details. The data model has three abstraction levels. The conceptual data unit is on the first level and corresponds to the basic objects that the algorithm works with. Each conceptual data unit is implemented in an abstract cloud as a generic data unit in the second level. The generic data unit (or container) has two types of files, the metadata which contains information about the file (such as verification data), and the files that store the data itself. Finally, on the third level, there is the data unit implementation, which consists of translating the container information into the specific constructions of each cloud.

In DepSky there are three types of parties: readers, writers, and cloud storage providers. Readers and writers are clients (can be the same process). It is assumed that only readers

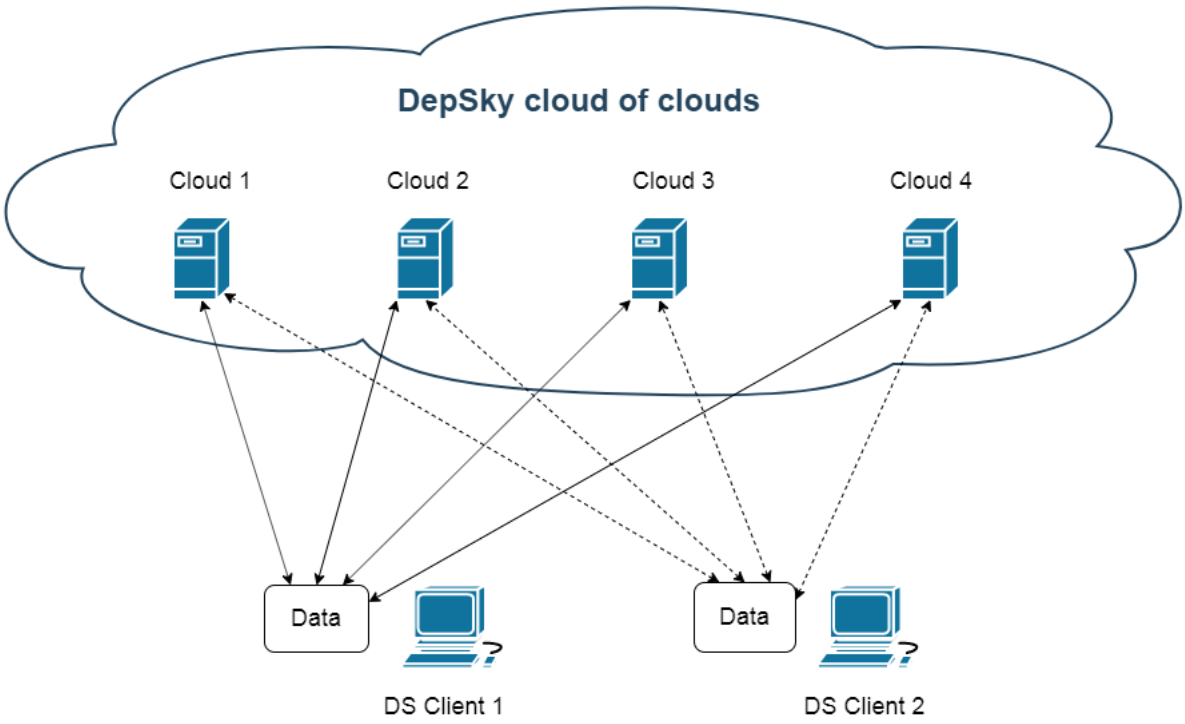


Figure 2.2: Depsky architecture [BCQ+13]

can fail arbitrarily, and writers can only fail through a crash. All writers of a data unit have a common private key to sign the data written and all readers have a common public key to verify the signatures. As for cloud storage providers, they fail in a Byzantine way. Each cloud is considered a passive storage entity. The protocols require a set of $n = 3f + 1$ storage clouds, to tolerate f faults. Additionally, the quorums used are composed of any subset of $n - f$ storage clouds.

As said before clouds can not execute code, so to overcome this the authors decided to implement a single-writer multi-reader register and provided a lock/lease protocol to allow several concurrent writers for the same data unit. In DepSky metadata and data are written in separate quorums. To ensure confidentiality the chosen method was to create a secret sharing scheme. The sharing scheme works by having a special party, a dealer, distributing a secret to n players, each player only gets a share of the secret. It is required at least $f + 1$ different shares to recover the secret, and no information is disclosed with $f$ or fewer shares. Each cloud just receives a share of the data being written, besides the metadata. This way no single cloud could have access to enough shares to reassemble the original data. Using a secret sharing scheme on the protocol with an information-optimal erasure code algorithm, prevents the linear increase of the cloud storage system cost, reducing the size of each share by $n/(f + 1)$ of the original data.

DepSky-A is a protocol used in DepSky that improves the availability and integrity of the

data stored on the clouds. It does so by replicating it on different providers using quorum techniques. The DepSky-A has a write and a read algorithms. On the write algorithm the first thing to do is to write the value on a quorum of clouds, then write the corresponding metadata. When a client wants to do its first write in a data unit, first it has to contact the clouds to retrieve the higher version number of that data unit, then update its maximum version number with the correct value for that data unit. For the read algorithm, the metadata files are fetched from a quorum of clouds, the one with the greatest version number is picked and the corresponding value is read and the cryptographic hash is found on the chosen metadata. Once the reader gets the first response that satisfies this condition, it cancels the pending requests and returns the value. Availability is guaranteed since the data is stored in a quorum of clouds with $n-f$ clouds and it is assumed that only f clouds can be faulty. The data and the signed metadata containing its cryptographic hash, together allow for users to verify if data was corrupted. Also, the read operation has to retrieve the value from only one of the always available clouds ($(n-f)-f > 1$).

The second main protocol is DepSky-CA. First the data encryption, the key shares generation, and the encoding of the encrypted data on the write operation and the reverse process on the read operation. Second, the data stored on a certain cloud[i] is composed of the encoded block e[i] and by the share of the key s[i]. Third, to read the current value of a data unit are needed $f + 1$ replicas instead of just one as it happens on DepSky-A. Also, the writer stores $n$ digests of the metadata file, one for each cloud, instead of just one. The digests are accessed as different positions of the digest field of the metadata.

To improve DepSky-A read operation, the authors proposed that some criteria are created to sort the clouds and try to access them sequentially until the desired value is obtained. The sorting criteria can be anything that fits a specific situation, or could even be a combination of criteria. An example of a sorting criterion can be the access monetary cost (cost-optimal), the latency of query of metadata on the protocol (latency-optimal), or even a mix of the two.

The protocols presented up until now do not support concurrent writes. The proposed solution for this situation is a lock mechanism. This mechanism uses the DepSky cloud-of-clouds itself to maintain lock files on data units. These files indicate the writer and the time it has to write access to the data unit. The protocol works as follows, the process $p$ that wants to write goes to all clouds and lists the files, and tries to find the zero-byte file on a quorum of clouds that indicates the lock owner ID. If the owner ID on the zero-byte file is different from the ID of the process and the local time is smaller than a certain time $T+\Delta$, where $\Delta$ is a safety margin of the clock difference between the synchronized clocks of all writers than it means that the lock is assigned to someone else. If such a file could not be found, then $p$ can write a lock

file on all clouds, where $T = t + writer\_lease\_time$. Finally, $p$ will list once again all files in the data unit trying to find other lock files with $t < T + \Delta$ besides the one he wrote. If it finds such a file, it removes the lock file it wrote and sleeps for a small random amount of time before trying to run the protocol again. Otherwise, $p$ becomes the individual writer of the data unit.

DepSky uses additional protocols to provide different operations than reading, write, and lock. These protocols are, for example, creation and destruction, used to create a data unit or destroy it. Finally, there is also cloud reconfiguration which is necessary to move blocks from one cloud to another. It works by having the writer reading the data from the clouds, then it creates the data unit container on the new cloud and executes the write protocol on all clouds, including the new cloud. The one cloud that is being replaced deletes the data and informs the readers by writing a special file on the data unit container of the remaining clouds.

The key insight with this paper is that a combination of cloud storage systems by creating a cloud-of-clouds can help overcome the limitations of single cloud systems, through the usage of proper techniques such as Byzantine fault-tolerant replication, secret sharing, and erasure codes among others.

DepSky is, overall, a good solution for the cloud storage system on a more centralized architecture. However, it has some limitations, for instance, requires a global synchronization clock mechanism between clients and also the existence of an extra lock/release process in each write operation. Also on DepSky, there is a metadata file for each file. This design is a problem because of the high maintenance of massive tiny metadata files necessary creating a metadata overhead in the multi-cloud multi-device synchronization [TLS+15].

**UniDrive Storage System**

In this section, we describe UniDrive. It attempts to solve some of the challenges discussed before with DepSky. However in UniDrive, the authors point out what they consider is a drawback of DepSky, which is that DepSky requires a global clock synchronization mechanism among clients and also the existence of an extra lock/release process in each write operation, also on DepSky there is a metadata file for each file/directory whereas on UniDrive is a single metadata file, which is the SyncFolderImage that captures all of the metadata. UniDrive, therefore, proposes a slightly different approach than DepSky.

UniDrive solution for the previously mentioned problems is to create a server-less, client-centric design, as proposed by DepSky, having clients performing all the system computation [TLS+15]. It also relies on the basic file upload/download operation to transmit messages for locking and notification in the synchronization of multi-cloud multi-device, this idea diverges

from the DepSky lock-release mechanism. To improve performance, they split user data into smaller chunks, perform erasure coding to add redundancy, like in DepSky. Afterward, they schedule the distribution of these chunks to the multi-cloud to meet reliability and security requirements. The three major components of UniDrive are an Interface, which is a way to abstract the communication with the different cloud providers, the Control Plane, which is responsible for the replication of SyncFolderImage, which is a single metadata file that captures all the metadata, to all of the clouds and clients, and lastly the Data Plane that handles the task of data transfer and all the associated tasks.

The Control plane design is divided into two parts, first the Data Model, and the Metadata Synchronization. First, we will analyze the Data Model. In UniDrive there are two types of data, the content data, and the metadata. The actual file content is represented by the content data and it could be divided into smaller chunks, creating data-blocks, which are the basic unit for a file transfer. The metadata captures the status of the sync folder and all the updates that occurred to it. It is composed of three parts, a SyncFolderImage, a segment pool, and a ChangedFileList. SyncFolderImage maintains the file system hierarchy of the local sync folder and all its files. For each file in the sync folder, there is an entry in the image, containing snapshots of the respective file where each snapshot summarizes the metadata of the file. There is also a segment pool composed of segments. Each segment is identified by the tuple <Block-ID, Cloud-ID>, where the Block-ID is its sequence number and Cloud-ID identifies in which cloud the block is stored. Finally, there is also a ChangedFileList that records all changes to the local sync folder. After each successful synchronization, the ChangedFileList is cleared.

The second part of the Control plane is the Metadata Synchronization. The consistency of files is done through the usage of metadata files. There are two types of metadata updates, the local update, and the cloud update. A local update is generated locally and then is propagated to the cloud and other devices. The cloud update is a pending update in which a device needs to be synced up. This type of update occurs when there is a more recent version number on the cloud metadata file. This verification is done periodically. To commit a local update, a client must first acquire the lock, to avoid concurrency issues, then it must make the metadata up-to-date by downloading and merging with cloud update and finally commit the resulting merge to the cloud.

In UniDrive to ensure consistency, all updates to the multi-cloud must be serialized first. This is achieved by using a mutual-exclusive lock mechanism, through a quorum-based distributed locking protocol. This protocol works as follows, the device $d$ that is trying to get the lock must first generate a lock file with the name $lock\_d\_t$ to identify itself and then upload the lock file

to a specific lock directory across all clouds. Afterward, $d$ lists on each cloud all files stored in the lock director. It acquires the lock of a cloud if there is only its lock file on that cloud. To avoid conflicts there is a quorum number of locks that it needs to successfully acquire. If it can get the quorum, then it proceeds to upload the metadata. If it can not get the quorum then it is treated as a failure and the $d$ executes a random waiting time before trying to acquire the metadata lock again.

To handle conflicting local and cloud updates first they record the original metadata $vo$ then obtaining the latest metadata version from the ChangedFileList $vb$, downloads the cloud metadata $vc$, and try to merge $vb$ with $vc$. To find if there are conflicts are needed to de-serialize $vc$ and compare it with $vo$ to find the difference $\Delta C$. The same is done to find the difference, $\Delta B$, between $vo$ and $vb$. Afterward, compare $\Delta C$ and $\Delta B$. If it does not exist matching updates it is to directly merge $vc$ and $vb$ to get the up-do-data metadata $vu$ by applying $\Delta C$ to $vb$ or $\Delta B$ to $vc$, which as the same result. If entries have both local and cloud updates then the merged metadata $vu$ retain both updates and it is prompted to the user to handle the merge.

Sometimes repeated transfer of unchanged part of the metadata happens, because some parts of the metadata do not change a lot during time. To avoid this they split the metadata into a base file and a delta file, a delta being the altered part of the metadata. Updates are always appended to the delta and normally the only delta is transferred to the cloud. Once the delta reaches a certain size it is merged with the base, which is then transferred to the cloud and synced with the other devices. The delta file is cleaned afterward.

The Data Plane handles data block generation, transferring, and scheduling. It tries to improve sync performance by minimizing traffic, exploring faster clouds, and maximize parallel transferring. To minimize data traffic, it is used in UniDrive the content-based segmentation method to divide a file into segments. These segments are indexed by the SHA-1 hash of all their content. This way segments that have the same content will have the same name. In UniDrive, the data block generation is done via erasure coding. Data blocks are immutable for consistency and efficient concurrent purposes. Once a segment is created, new data blocks are generated. To reinforce security, non-systematic Reed-Solomon [ree] codes are applied to generate parity data blocks, this way their semantics are removed while preventing the cloud providers from inferring the original contents.

Data Block Scheduling is done through a scheduler that determines how to distribute the data blocks into the multi-cloud while keeping security and reliability requirements. A file is considered available when each of its segment has $k$ data blocks uploaded to the multi-cloud. And it is said to be reliable when each cloud received at least its fair share. UniDrive was

designed a few strategies for files to become available and reliable.

The first strategy is the basic upload scheduling where the schedule of files is done evenly to all the available clouds. This over-provisioning works by sending extra parity blocks to faster clouds even if they have received their fair share to mask performance disparity and better leverage faster clouds. Another strategy is dynamic scheduling for batch uploading which consists of the fact that in the case of batch file uploading, data blocks are scheduled in two phases, availability-first, reliability-second. Meaning that when a file becomes available, all network resources are assigned to the next file. When all files become available the transfer of blocks starts to fulfill the reliability requirement to the clouds that have not yet received their fair share. Also, there is the strategy of dynamic scheduling for download which consists of having eligible clouds sorted by connection speed and the request of the next block is always assigned to the idle connection of the fastest clouds. The next strategy that is worth mentioning is the in-channel bandwidth probing that intends to put the next download or upload to the available cloud that is faster, to do so they monitor the throughput of all the transferred data blocks that are happening at the time to each cloud, compute an average per-connection throughput of all clouds and then sort them.

## 2.3 Storage and Blockchain

In this section, we will analyze storage systems for the blockchain, namely, Filecoin, Storj, Blockstack. These storage systems are made considering the usage of a blockchain. The important thing to remark here is that these systems work alongside a blockchain, or could even use the blockchain as part of the system. We chose to analyze these systems because they are known and representative.

#### FileCoin Storage System

Filecoin consists of a decentralized distributed storage network. Filecoin operates on top of IPFS as an incentive layer. Customers spend Filecoins (tokens) for data storage and retrieval, and miners receive Filecoins (tokens) through data storage and service [BG18]. The Filecoin DSN (decentralized storage network) takes care of requests for storage and recovery through two verifiable markets: the Storage Market and the Retrieval Market, respectively. Customers and miners set the prices and send their orders to the markets for the demanded and provided products. The markets are run by the Filecoin network, which by using Proof-of-Spacetime and Proof-of-Replication, presented below, ensures that the information they commit to storing has been properly stored by miners. Finally, miners can create new blocks for the underlying

blockchain. A miner's influence over the next block is proportional to their storage amount in the network currently in use.

To better understand Filecoin we must define Proof-of-Replication and Proof-of-Spacetime. Proof-of-Replication (PoRep) is a new Proof-of-Storage (PoS). PoS is a cryptographic protocol used mainly to check a remote file's integrity. This is achieved by submitting an encoded data copy to a server and then executing a challenge-response protocol to test the data's integrity. PoS allows a user to check whether the outsourced data is stored by a storage provider at the time of the challenge. In PoS there are two participants, the provers and the verifiers. The provers consist of participants who are storing some data. Verifiers consist of participants who validate that the provers are storing the data. Proof-of-Replication is similar: it allows a server to tell a client that some data $D$ has been replicated to their unique physical storage. On Filecoin's scheme the prover $P$ commits to store $n$ distinct replicas of some data $D$. Then convinces the verifier $V$, that $P$ is storing each of the replicas through a challenge/response protocol.

Proof-of-storage schemes (also used in Siacoin [VC14]) can not prove efficiently that some data was being stored throughout a certain time. To address this question, they proposed Proof-of-Spacetime, where a checker can check if a prover stores his/her outsourced data for some time. The prover must generate sequential Proofs-of-Storage (in this case Proof-of-Replication), as a means of determining the time and composing the executions recursively to generate a short proof.

As said before, Filecoin has two markets: the Storage Market and the Retrieval Market. The two markets have the same structure but a different design. The Storage Market enables Customers to pay Storage Miners to store data. The Retrieval Market allows Customers to collect data by paying the data to Retrieval Miners. Customers and miners can set their offers and demand prices or accept current offers in both cases.

Filecoin provides end-users with two basic primitives: Get and Put. These primitives allow customers to store data at their preferred price and to collect data from the markets. While the primitives cover the default use cases for Filecoin, by supporting the deployment of smart contracts, it becomes possible to have more complex operations designed on top of Get and Put. Therefore, users can program new fine-grained storage/retrieval requests, classified as File Contracts and also generic Smart Contracts.

#### Storj Storage System

Storj is a decentralized file storage framework. The Storj Network [WBBB14] is an object store blockchain-based hash table that encrypts, shards, and distributes data to nodes around the

world for storage.

In the Storj network files are stored, like in Filecoin and Siacoin, on the free disk space of the system participants that make the P2P network. The users that provide the disk space are called storage nodes, and they receive Storjcoins for the disk space and bandwidth they provide to the system, meaning that they get paid for the data that they host on their disk.

In contrast with traditional centralized storage systems, in Storj, the storage nodes have zero access to the data that they are hosting, they are not able to view, alter, and in fact, no storage node even has a complete copy of any file. This is because files uploaded to the Storj network are previously split into shards, encrypted on the client's computer, and then the shards are divided between the storage nodes for hosting, without having neither of the storage nodes a full copy of the file. The client should be the only person who knows where all the shards are located. Storage clients choose how many storage nodes they want hosting their data and for how long they would like for them to do so. This contract remains active for as long as the client remains paying the required payments throughout the lifetime of the file storage. If the client stops paying the files will be forgotten.

In the Storj network, there are three major actors: metadata servers (Satellites), object storage servers, and clients, as we can see in Figure 2.3. Object-storage servers hold the majority of the data stored in the system. Metadata servers keep track of per-object metadata and where the objects are located on object storage servers. Clients provide a coherent view and easy access to files by communicating with both metadata and object storage servers. Storj uses Lustre's architecture to boost performance [lus]. Object-storage servers or Storage nodes consist of individuals or entities that have excess hard drive space and want to earn income by renting their space to others. Finally, metadata servers or Satellites operate in the node discovery system and they cache node address information, store per-object metadata, maintain storage node reputation, aggregate billing data, pay storage nodes, perform audits and repair, manages authorization and user accounts. Any user can run their Satellite. But the expected is to have users create an account on another Satellite hosted by a trusted third party.

Storj implements a distributed hash table, called Kademlia, with a basic decentralized caching service on top, so clients can locate all the shards of their original file. This hash table requires a private key to discover the shards. Without the private key, it would be extremely difficult to correctly guess the locations of a sharded file. The caching service will live independently in each Satellite and attempt to talk to every storage node in the network on an ongoing basis.

In the Storj network nodes can not be trusted, therefore there must exist some kind of

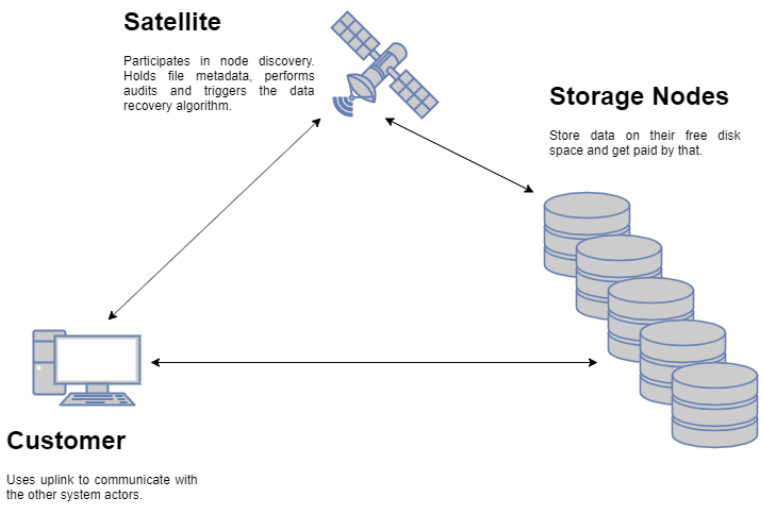


Figure 2.3: Storj System peer classes

audit to validate that nodes are returning data accurately. This happens by having Satellites sending a challenge to a storage node and expect a valid response, a proof-of-work challenge. As nodes go offline, some pieces of data can disappear with them. If this happens the Satellite will mark that nodes' file pieces as missing. This will trigger a lookup in a reverse index within a users' metadata database, identifying all segment points that were stored on that node. For every segment that drops below the appropriate minimum safety threshold, the segment will be downloaded and reconstructed, and the missing pieces will be regenerated and uploaded to new nodes. And the pointer will be updated to those new nodes.

### Blockstack Storage System

Blockstack [ANSF16] is a Storage System that allows users to not trust remote servers, such as DNS in fact, the Blockstack implementation uses zone files for storing routing information, which is identical to DNS zone files in their format, in the next paragraphs we will get into more detail. Blockstack removes any middle network failure points. Blockstack's services for identity, discovery, and storage services can survive failures of underlying blockchains.

As we can see in Figure 2.4, the three major components of Blockstack are a blockchain that is implemented using virtualchains and is used to bind public keys to the digital property (domain names). A peer network, called Atlas, provides a global discovery information index and finally a decentralized storage system called Gaia that provides high-performance storage backends, mainly clouds, without the introduction of trusted central parties.

The blockchain is used for two main purposes. First to provide the storage medium for operations and second, it allows consensus on the order in which operations are written. The

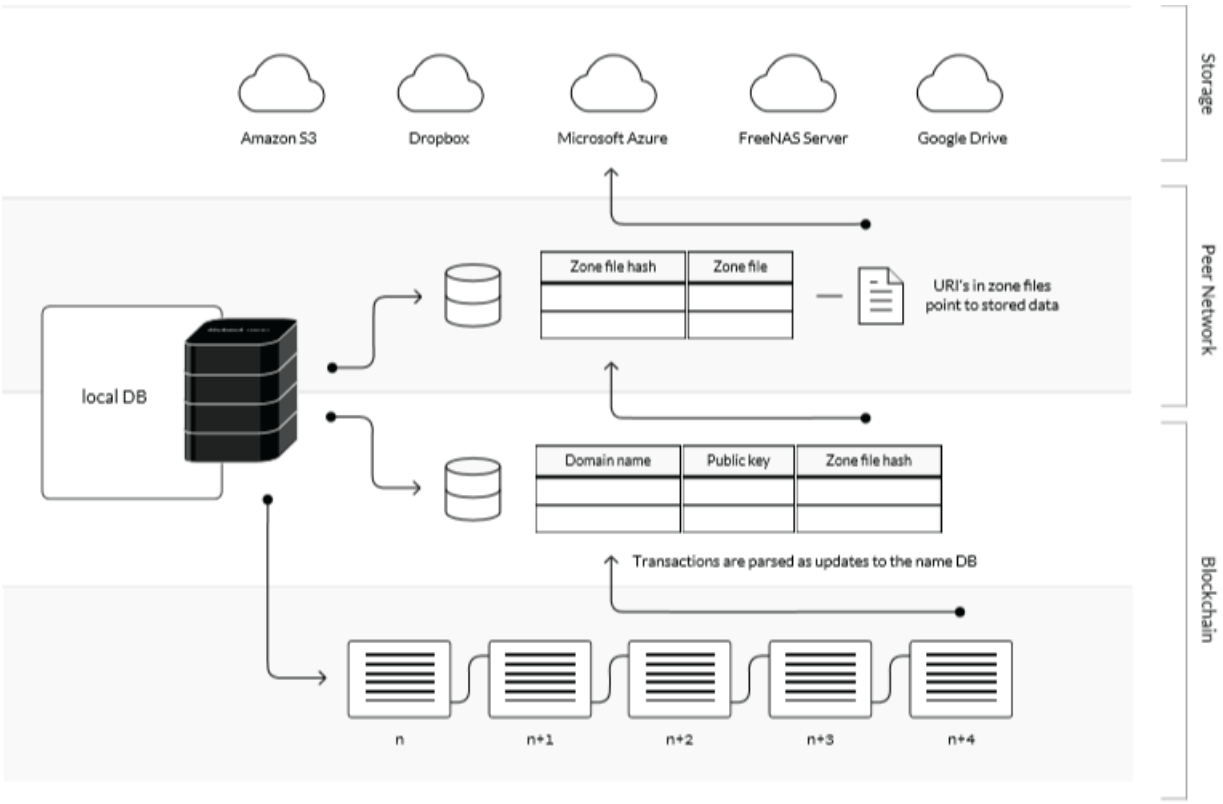


Figure 2.4: Blockstack Architecture [ANSF16]

operations are encoded on the underlying blockchain in transactions by the virtualchain. The virtualchain uses the blockchain as an abstraction of totally-order operations. Virtualchains are similar to virtual machines and their operations are coded as additional metadata into valid blockchain transactions. The logic for processing the virtualchain operations only take place at the virtualchain level even though the transactions are seen by blockchain nodes.

The storage and data discovery are separated in the Blockstack architecture, as represented in Figure 2.4. This way it is possible to have multiple storage providers such as clouds and P2P systems. This peer network, which performs the routing, uses zone files to store routing information. These files are stored on the peer network layer. Users do not need to trust this layer because testing the respective cryptographic hash of that data recorded in the blockchain, will verify the validity of any data record in this layer. In Blockstack's implementation nodes form a peer network for storing zone data, called the Atlas network. The cryptographic hash of the zone file is written in the zone files table only if it was announced previously in the blockchain. Nodes that participate in the peer network maintain a full copy of all zone files in the current implementation of the Atlas network because the size of these files is relatively small (4 KB per file). The data records that represent routes can be verified and, therefore, can not be tampered with.

The top layer corresponds to the storage layer. The actual data values are hosted here, in the storage layer. The stored values are signed by the owner key. Because data is stored outside the blockchain it can have arbitrary size and various storage backends. The user does not need to trust this layer as well. Users can verify the integrity of the data by comparing the stored hash with the blockchain. In the Gaia system, the zone file of users contains a URI that points to the stored data. Writing data also implicates the signing and replication of it. And reading the data clients have to fetch the zone file and the data and make the integrity verification. Blockstack also implements a decentralized naming system called Blockchain Name System (BNS). This decentralized system attempts to work like the DNS for the Internet, by giving the location of nodes and content. BNS system binds human-readable names to discovery data.

## 2.4 Proof-of-Storage

For a P2P storage system that stores data on peer's storage space, it is important to have some kind of cooperation incentive mechanism to prevent the system's collapse. One example is the fact that all of the available resources could be used up by nodes that do not share their resources with the network, this way correct nodes would leave the network making the system unusable. These types of attacks are the free-riding attacks and will be discussed further in the document. Considering this we decided to add a PoS mechanism to our solution.

The PoS mechanism allows a client to check if a server has a file without downloading that file. There is a lot of research on cryptographic mechanisms that provide such proofs [BJO09, AKK09, WWRL10, WLL15, WRLL10, dC14, SW08]. These mechanisms provide verifiability (data integrity can be verified using proofs) and unforgeability (a malicious server cannot forge a proof without having the files). These mechanisms are interesting but are heavy in terms of computation and space required, so they are adequate for clouds but not for P2P networks.

A variant of Proof-of-Storage, sometimes called Proof-of-Replication, allows verifying if some data $D$ has been replicated, i.e., if there are enough copies in the system [BDG17, Fis19]. Enforcing unique physical copies enables a verifier to check that a prover is not deduplicating multiple copies of $D$ into the same storage space.

An alternative to having to prove replication is to ensure cooperation [S$^+$07]. There are two main cooperation schemes; those based on reputation and those based on remuneration.

*Reputation* mechanisms have three stages [KBC$^+$00, RD01, DKK$^+$01]. (1) First, collection of information, meaning that peer reputation is built based on the observation of the peer, experience with it, and/or third-party recommendations. (2) Second, in the cooperative decision stage, depending on the information obtained, a peer will determine whether to cooperate with

another peer or not, depending on the reputation of that other peer. (3) Finally there is the cooperative assessment which means that after an interaction, a node must provide an assessment of the degree of cooperation of the peer involved in it.

*Remuneration* (or incentive) mechanisms consist of four main operations [Nak08, Woo14, BG18]. First, the negotiation process in which two peers may have to negotiate the interaction terms. Second, the cooperation decision, meaning that during the negotiation and based on its outcome, a peer will determine whether to cooperate or not. Third, there is the cooperation evaluation, in this phase, the requesting peer has to evaluate the provided service. Finally, the remuneration which can consist of virtual currency or real money or even bartering units, meaning quotas defining how a certain amount of resources provided by the service may be exchanged between entities.

It is relevant for our work to implement such a mechanism to ensure the sustainability of our system. Considering that it is based on P2P, it is important to ensure that every node is storing the files as promised. Otherwise, there would not be any incentive, or reason, for the nodes to work with the system, or even to use the system. The Proof-of-Storage mechanism is important to create an incentive mechanism that ensures that each node is storing the files as promised.

## 2.5 Discussion and Summary

The previously mentioned state of the art has some advantages and disadvantages. Table 1 represents a comparison of these system's integrity and availability. The first column describes the systems previously analyzed and our P2Cstore system. The second column describes the architecture used by each of the systems. The third column sums up the method used to ensure integrity for each system. Finally, in the fourth column, we explain briefly how each system attempts to ensure availability.

For integrity, IPFS uses a URL with a file's cryptographic hash to ensure integrity. DepSky, UniDrive, and Storj use metadata files with verification data. Both of these techniques work well in their particular systems. The usage of a URL for integrity checks is great for IPFS because it is the way that files are discovered in the system. With this technique, they can use URL with the hash of the file both to find files and for integrity checks. The metadata files are great for the other three systems because it is not necessary a URL to locate files as they use centralized servers to store them (DepSky and UniDrive). This way clients can verify the file's integrity by storing metadata files with the necessary information. In the case of Storj, since they use satellites, the idea is very similar to DepSky and UniDrive because a node goes to a satellite to

find where a certain file is, not by a URL. Blockstack uses the hash of blockchain transactions which in case of the existence of a blockchain is a great idea due to the previously discussed blockchain properties.

We have two types of system architectures, P2P, and Centralized as said before. IPFS, Storj are P2P systems and ensure availability by spreading across a certain number of peer's files. DepSky and UniDrive, which are systems with centralized architecture, use multiple cloud providers to ensure availability, spreading the files between several clouds. Blockstack can be P2P, Centralized, or even both, therefore they use both of the previously mentioned techniques, multiple cloud providers, and multiple peers to store data to increase availability.

The last line on the table consists of our system, the P2Cstore. Here we can see that it ensures integrity similarly to IPFS, through the URL of the file and its cryptographic hash. This way it is simple to verify the file's integrity. For availability, we use both cloud storage providers and P2P to improve the overall availability. When compared with others it is easy to see that it attempts to use part of each, meaning that it attempts to use clouds like DepSky and Unidrive to store data, and also P2P like IPFS and Storj. It also uses a DHT similarly to both Storj and IPFS for node/file discovery.

In conclusion, depending on the scenario the state of the art has advantages and disadvantages, in the next chapter, we present the design of the P2Cstore.

| System | Architecture | Integrity | Availability |
|---|---|---|---|
| IPFS | P2P | Through URL with file's cryptographic hash | It is a peer network and files are replicated to increase availability |
| DepSky | Client-server | Using metadata files that contain verification data | Multiple cloud storage providers to increase availability |
| UniDrive | Client-server | Using metadata files that contain verification data | Multiple cloud storage providers to increase availability |
| Storj | P2P | Using metadata files that contain verification data | It is a peer network and files are distributed to increase availability |
| Blockstack | P2P and/or Client-server | Through the underlying blockchain | Multiple storage providers to increase availability, clouds and/or P2P |
| P2Cstore | P2P and/or Client-server | Through URL with file's cryptographic hash | Multiple storage providers to increase availability, clouds and/or P2P |

Table 2.1: Systems comparison

# Chapter 3

# P2Cstore

The goal of this dissertation is to create a storage system for blockchain applications, P2Cstore. This system is envisioned to work in parallel with any blockchain application. P2Cstore is composed of two major components which are a P2P network and Cloud storage providers. By using the combination of P2P with cloud storage providers, we intend to improve the availability of the system as well as give its users the choice between both architectures, or even a combination of them. We also decided to create an incentive mechanism as well as a Proof-of-Storage algorithm to incentivize nodes to cooperate with the network instead of working against it.

In this chapter, the conceptual design of the P2Cstore will be explained in detail. Section 5.2 describes the trust assumptions as well as the requirements that should be met by the system. Section 3.2 describes the components that make P2Cstore. In Section 3.3, we will describe the system in more detail, namely the system entities, the system model, the software architecture, the basic operations, and the system interface. In Section 3.4, we describe the Proof-of-Storage (PoS) Algorithm, how it works, why it is important to have it, and how it prevents some attacks to the system, like a free-riding attack. Finally, Section 3.5 provides a summary of the concepts addressed throughout this chapter.

## 3.1 Trust Assumptions and Requirements

To focus on the problem that we are tackling we considered the following trust assumptions:

1. We assume that the cryptographic primitives are secure;
2. We do not trust individual peers or individual cloud storage providers;
3. A fraction of peers can leave the system and cloud storage providers can become unavailable at any time;

4. Each node can only use the same amount of P2P storage that it gives to the system

The system has some requirements that are important to mention here.

1. **Data Freshness**: When a client requests a file from the system, the system must return the most recent version of the stored file.

2. **Data Integrity**: The system must ensure that data has not been changed in any way. In case that it has the system should be able to know and to get the original version of the file.

3. **Data Availability**: The data must always be accessible to clients. The system must be able to ensure that at any given time a certain file will be available to be read by a user.

#### Education Certificates Storage on the Blockchain

These requirements are the most important because of the type of system we want to implement. This is the case because, as was mentioned in the Chapter 1, the project QualiChain (https://qualichain-project.eu/) is developing a blockchain-based system to enforce the authenticity of university certificates. Our work is aligned with this project and attempts to create a storage system for the blockchain to store education certificates. This was the initial motivation. In the context of this project, it was developed an ecosystem in which education certificates can be verified through the Ethereum blockchain proposed by Serranito et al. [SVGC20].

The ecosystem is based on two smart contracts. The Consortium Smart Contract (CSC) manages the Higher-Education Institutions that are members of a consortium of HEIs. The HEI Smart Contract (HSC) stores authenticators (i.e., cryptographic hashes) of the education certificates issued by an HEI. There is one HSC per HEI of the consortium.

There are two typical workflows. (1) A student graduates in an HEI. The HEI issues the certificate and stores the authenticator of the certificate associated with some id in its HSC. (2) The ex-student applies for a job and provides her certificate. The company inserts the certificate, the id of the certificate, and an id of the HEI in an application, that contacts the CSC and the HEI's HSC to get the authentication and verify the certificate.

The HSC does not store the certificates, that have to be stored externally. That work uses IPFS for that purpose. P2Cstore can be used instead of IPFS with the benefits we mentioned, e.g., flexibility and improved availability using replication.

Knowing this, the first requirement is important because the most recent, and up to date certificate could be the only one valid. Therefore it is of paramount importance that the system ensures this property. The next requirement, data integrity, is also relevant because the clients

must have some way to verify and be certain that the certificate that they are reading is not tampered with and that it is trustworthy. Considering that we can not trust the cloud system providers nor the peers of the P2P network, it is important to ensure this integrity property. Lastly, data availability is important in our particular case because, for example, if a user sends a certificate location (URL) for a company he wants to be sure that the company will be able to read his certificate to verify its authenticity. Once again the system must ensure this because nodes can leave the network, crash, or have malicious behavior, for instance, they can delete files and say that they are still storing them.

When designing and implementing the P2Cstore system we had all these requirements into consideration. However, in the development process, we decided to improve upon our initial idea and to also create an incentive mechanism that helps to ensure that the users will not work against the system but will instead work with it. Our incentive mechanism monitors the storage usage of each node and verifies if each node is storing the files it is supposed to. In the functional evaluation process (qualitative evaluation) we assess whether or not these requirements are met by our system.

### 3.1.1 System Model

The P2Cstore system is composed of a set of *nodes* that communicate by message passing. Nodes can be *online* or *offline*. For the system to properly function, nodes have to be online at the same time. Nevertheless, nodes that are offline during some operations can become online and recover later. Clocks do not need to be synchronized.

A node is considered *correct* if it follows the algorithm, otherwise it is *faulty*. The system tolerates several types of node failures: a node can go offline and back online repeatedly; nodes may go offline indefinitely; a node may tamper with the content of the files it stores.

Nodes use Kademlia DHT [MM02] to find which nodes are storing some specific content. Each node has a node ID and the Kademlia algorithm uses the node ID to locate values on the network.

We assume the communication is reliable and secure. We do not present a specific solution for how to obtain this as there are several, e.g., using the TLS protocol [DA99] or the DTLS protocol [RM12].

It also important to mention that in our system model the files are identified by a URL.

In the next sections, we will describe the system architecture in more detail as well as the interaction protocols.

## 3.2 P2Cstore Components

Before starting to describe our system it is important to understand the components and concepts that make it. In particular the Kademlia DHT, the JClouds library, the cloud storage providers, and how they are set up as well as how they interact with our system and finally the concept of Content-addressable P2P System and why it is relevant for our work.

### 3.2.1 Kademlia DHT and Content-addressable P2P System

As described in Chapter 2 the IPFS system uses the Kademlia Distributed Hash Table. We decided to use a similar approach, meaning that we decided to use a Kademlia DHT as the DHT of our system. We decided to use this solution because this way nodes can easily lookup other nodes by a decentralized clustering algorithm. With Kademlia DHT it is also possible to find which nodes have some specific data by referring to the multihash of that data, multihash consists of "a protocol for differentiating outputs from various well-established hash functions, addressing size + encoding considerations" [mul]. In sum, if one node wants some data, it requests it to the DHT to find what are the nodes that can serve its request. What is stored on the DHT are references, which are the node identifiers, which consist of random numbers regenerated upon node registration that identify the nodes on the DHT that have the data. The fact that our system is content-addressable makes it more secure and gives the nodes more privacy, because this way, a client of the system that attempts to read from it does not needs to know from which node it is reading the file. It simply needs the key to the file (the key consists of a URL that has the following structure *node_ID/file_key*), which consists of a hash of the contents of the file and it can read the file. In our solution what happens is that upon adding a file to the system the file is automatically renamed and the new name consists of the hash of the file contents. This way it is possible to get files from the P2P network and the clouds just by knowing the file key, which is the hash of the contents.

It is also important to explain how the distance between nodes is calculated in Kademlia. To calculate the distance between two nodes, Kademlia computes the XOR operation on the two node IDs, then it takes the result as an unsigned integer number. This integer is considered as the distance between the two nodes, so for smaller values the nodes are considered closer together, for larger values they are considered further apart from each other.

### 3.2.2 JClouds and Cloud providers

The fact that our system uses Cloud Storage Providers alongside the P2P storage system gives each client the possibility to choose if they want to use Clouds to store their data or not. If they

decide that they want to use clouds they can register to a cloud storage provider, and with the proper credentials, they can use these clouds to store the system files. This is the case because commercial cloud storage providers come with a cost, and only by registering on such services, getting the proper credentials, and authenticating themselves through the system is possible for the particular client to access these features of the system. However, to improve upon our solution and make the system more available, we decided to make it possible for a client that does not has the cloud credentials to access them to read files through one node that has the proper credentials.

In our work, we choose to use the Amazon Simple Storage Service (Amazon S3), which is an object storage service [Amab]. We also choose to use the Google Cloud Platform (GCP) which as a Storage feature. We choose to use these cloud storage providers because they are the most common and also because they are included in the library JClouds. JClouds consists of an open-source multi-cloud toolkit for Java that provides an abstraction to the development of applications that use several cloud providers, with this tool it is easier to interact with several different clouds without the necessity of learning how to communicate with each and everyone specifically, instead we use JClouds to abstract the interaction with the clouds. With the help of JClouds, we were able to easily implement the cloud storage providers because JClouds works like an interface between the developer and the actual clouds, this way, it is possible and relatively simple to implement a new cloud storage provider if that ever becomes necessary.

## 3.3 P2Cstore System

In this section, we describe P2Cstore in more detail. To properly understand the P2Cstore system we will first describe its entities, what they are in the system, and what they can do. Afterward, we will describe the system model.

### 3.3.1 P2Cstore Entities

P2Cstore is envisioned to be used in a model where there is an interaction among three types of entities: cloud storage providers, storage peers, and client readers (see Figure 3.1).

- **Cloud providers** - are commercial public infrastructures that provide to their clients data storage and high levels of availability.
- **Storage peers** - Nodes that store content from other nodes and participate in the routing algorithm. Storage peers can perform all the operations on the system. They make up the Peer Network. These participants use the network like readers but also provide the

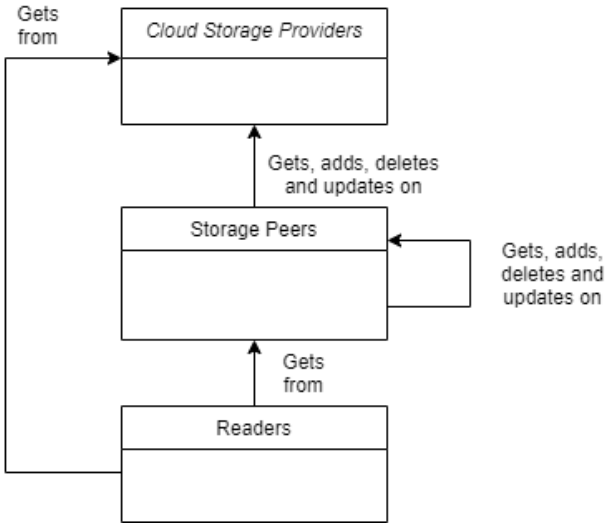


Figure 3.1: System Entities and their relationships

storage and due to that fact can also store content on the network/system, working as servers/storage providers as well.

- **Readers** - Nodes that do not store content from other nodes nor have to give storage space to the system. This type of participant can only read from the system.

We make the distinction between readers and the storage peers that work like clients and can read from the storage system because these readers do not have to participate in the system, do not need to share storage space, and can not store or delete any data, they can simply read. This was added to allow the sharing of files among people that do not want to participate in the system but with whom someone who does participate wants to share a file. For example, knowing the HEIs are the storage peers, Human Resource recruiters could want to see education certificates from candidates without wanting to participate in the system. The sharing of the file by some node in the system with someone else would work by having the node simply send the key of the file, (which is how nodes can identify files in the storage system as said before), to that other person/node and this person/node can then request the file to the system without participating in it. To increase availability we also use Cloud storage systems. It is important to mention that these clouds do not execute any code, they are only there for storage purposes.

In Figure 3.1 we can see an abstraction of the Entities in the system. We can see the three entities previously described, the operations each one can perform, and their interactions with each other. In particular, the Readers can perform the get operation from the Storage Peers, and from the Cloud Storage Providers, if they have valid credentials, the storage peers can perform add, get, delete and update operations from other storage peers and from the Cloud Storage Providers.

Every time a node adds a file to the system a key is generated. This consists of a hash of the file contents plus the owner ID. This key is stored in the DHT as a new entry every time an add operation is done, and as said before, when combined with the owner ID is used to locate the file in the system either for reading or deletion. This key can be shared with other nodes to enable them to see the file content, associated with the shared key.

### 3.3.2 P2Cstore Overview

The problem that we solve, as described above, is to create a storage system to store generic files for DApps.

P2Cstore is based on a fairness condition: each node can only use the same amount of P2P storage that it gives to the system (however, it can use more in the cloud which might incur some costs). This way the sustainability of the system is ensured. One could think that this way it is not worth it to use the system, given that an organization can only use what it gives. However, it allows replicating files in a set of nodes, improving availability.

This guarantee is enforced with two mechanisms that prevent a node to use storage space in other nodes without providing space in his (free-riding). The first is an extension of the Kademlia routing table with extra data about the used storage and the storage given to the system by the node. Every time a node $A$ wants to add some content to the system it does verification on the routing table to see if a set of nodes on the network have available storage for that file or not; when the node $A$ finds a destiny node $B$ to store the file it will verify on the routing table if the node $B$ has available space for that content. The second is a Proof-of-Storage algorithm, explained in Section 3.4.

It is worth mentioning that both the storage peers and the readers can communicate directly to the clouds, however, if a node $A$ does not have access to the cloud but he wants a file that is stored in the cloud owned by some other node $B$ who does have access, $A$ can access to the cloud through $B$. This is done by having the node $A$ requesting a file $f$ that is stored only on the cloud of node $B$, and, if $B$ agrees, it can request on behalf of node $A$ and send him the file himself after retrieving it from the cloud.

### 3.3.3 Software Architecture

In Figure 3.2 and Figure 3.3 we can see two diagrams of the system architecture. In Figure 3.2, we have a generic architecture of the system, as described above it is composed of peers of the peer network that also provide storage, by readers. And lastly by cloud storage providers. In Figure 3.3 we have a more detailed image of the Peer Network, in particular its two major

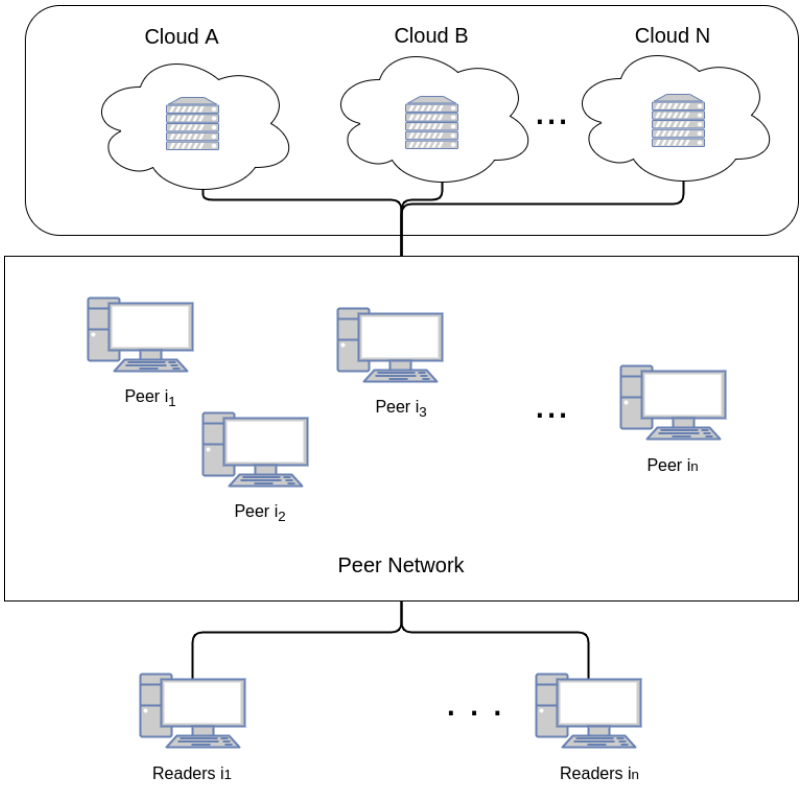


Figure 3.2: P2CSTORE System Architecture Overview

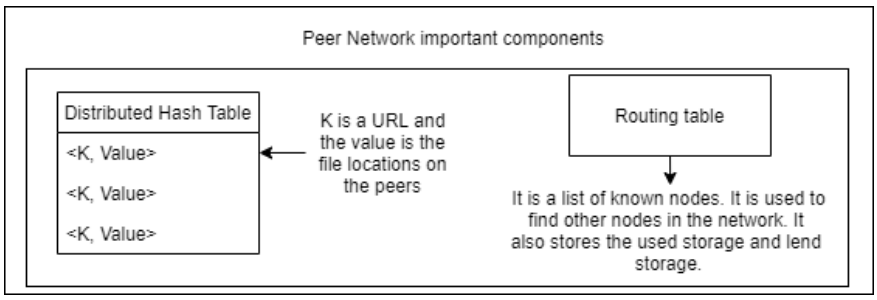


Figure 3.3: P2CSTORE P2P Network components

components, the Distributed Hash Table (DHT) and the Routing Table. As described in the image, the DHT is composed of a set of tuples $< K, Value >$. This K corresponds to the URL of the file, and the Value stores the nodes that have the file stored. In the Routing Table, we have the list of known nodes. It is also stored as a node characteristic the amount of used storage and lend storage for each known node.

### 3.3.4 Basic Operations

In any storage system, some operations are straightforward. For instance, it must be possible to read data from the storage system and to write data. It is also important to be possible to delete data as well as update data. All operations follow a particular interaction protocol. In Figure

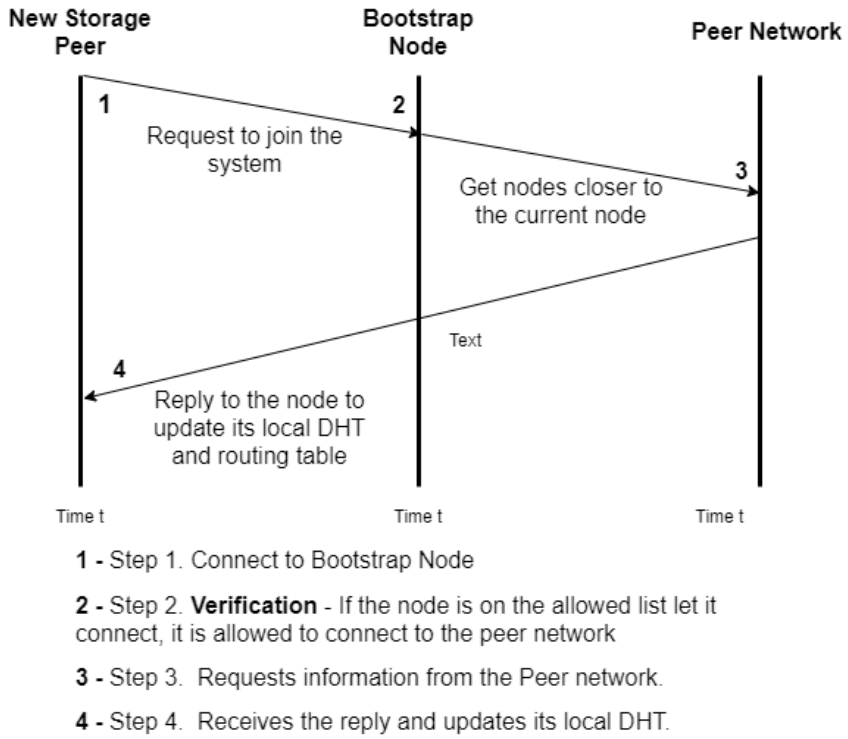


Figure 3.4: Add New Storage Peer to the System Protocol Diagram

3.5, Figure 3.6, and Figure 3.7 we have three diagrams of the Add, Get and Delete operations, respectively. It is important to mention that even though the Storage Peer 1 is represented outside of the Peer Network, it belongs to the Peer Network, it was placed outside because is behaving like a client and is performing operations in the system.

There are several interaction protocols in the system. These protocols make up the majority of the interactions that are expected to happen during a normal system operation. Next, we will also analyze in more detail how each of these operations works on our storage system and the associated interaction protocol.

### Adding a new Storage Peer to the system

Before any node connects to the network it is important to first create some Bootstrap Nodes. These nodes will be later used by each new node (Storage Peers/Readers) to connect to the network. This is the case because we use a P2P network and we need a node, or set of nodes, which we call Bootstrap Nodes, to be known by every new node before they join the system. What we decided to implement was this set of Bootstrap Nodes that are known before the start by each of the new users. In Figure 3.4 we have a simplification of the interaction that happens every time a new Storage Peer attempts to connect to the network.

In Figure 3.4 a New Storage Peer is attempting to connect to the P2P network. To do that it connects first to a Bootstrap Node, represented in step 1. Before being accepted in the network,

a verification is made to ensure that the New Storage Peer IP belongs to the list of enabled IPs, step 2. This list has all the IPs that are allowed to join the network. If some node wants to join the network and it is not on the IP list it has to request someone on the network with permissions to add his IP to the list. We choose to add this feature because we considered that it is safer to trust nodes that are already in the network than nodes that want to join. We believe that it is worth to make this verification/acceptance of new nodes only possible for Bootstrap Nodes because they are the ones that make up the network initially and therefore are considered safe.

In the example of Figure 3.4 let us assume that the node is on the IPs list and is accepted to join and access the network. Once it connects to the BootStrap Node and is accepted to join the network it will request the Peer Network to give it the DHT information and the routing table information, in particular the nodes that are closer to it, step 3. Once he receives the reply from the system it updates its local DHT and routing table and is added to the peer network, step 4.

#### Get Operation

Figure 3.5 has a description of the get operation protocol. The get operation retrieves a particular file from the system and reads its contents. Let us assume that we have a certain reader $r$ that has a certain URL of a file $f$ and wants to read it from the system. First, $r$ must use the URL to look up the value associated with it in the local DHT to find the NodeId of the supplier node/nodes of the file $f$ and if he has it he can just make the routing to the node and perform the request. If he does not have that information on the DHT he can query that information to the network and then perform the routing to the node and make the request. If for some reason, the NodeID is offline he once again has to go to the DHT to find out where is another node with the file could supply it. And it will request every node that could be online and have the file until it gets it.

As we can see in Figure 3.5 the Storage Peer 1 is attempting to get a file with a key $K$ from the system. To be able to do that he searches on the DHT for the nodes that are storing the file with the key $K$, step 1, in our particular case these nodes are at Storage Peer 2, 3, and 4 and sends the request to each of them, step 2. If the cloud functionality is enabled a request is also sent to the cloud.

Once each node receives the request, it will search for the file locally using the given key $K$ and send it as a reply to the requesting node, in this case to Storage Peer 1, step 3.

Once the file is received by Storage Peer 1 it is necessary to perform a proof of integrity to

ensure that the file was not corrupted in any way, step 4. This is performed by calculating the hash of the contents of the file and comparing the resulting hash with the file key, which, as previously said, is the hash of the file contents. If the two hashes match, then the file was not corrupted therefore it is stored locally and can be used/read by the Storage Peer 1, otherwise, an error is thrown and the file is rejected.

It is important to explain that the get operation is divided into two possible operations, the get operation and the get-shared operation. The get operation only requests to the user the file key, not the node ID of the node that stored the file in the system. The system then computes the URL by getting the node ID from the local node. For example, let us assume node $A$, with a node ID of $nodeIDofA$ adds a file $f$ to the system, and this file $f$ has a key $k$. If node $A$, later on, wants to read $f$ from the system, it can simply execute a get operation. The system requests the key of the file $f$, which in our case is $k$, afterward, the URL is generated by simply getting the local node ID of node $A$, which is $nodeIDofA$ in our example, and concatenating it with the key $k$, as such - $nodeIDofA/k$. This URL is then used to retrieve the value from the system.

The get-shared receives as input from the requester the full URL to the file. It receives the node ID of the node that stored the file and the key of the file. With this operation, it is possible to share files with other nodes. Only by knowing both the node ID and the key of the file it is possible for a node to get a file from the system that it does not own. This operation can be performed by both Readers and Storage Peers.

### Add Operation

To add a file, a participant inserts the path to the file, which in turn represents the file that is then added to the system, and the system returns the corresponding URL, this way the client can afterward read the file or share it by sending the URL to other entities. The number of nodes that will store the file, both peer nodes or cloud storage providers is configurable by the client, giving the freedom to choose how much redundancy it wants and how much it is willing to pay.

In Figure 3.6 we have a diagram of the protocol for the Add operation. The node Storage Peer 1 is attempting to add the file $f$ to the system. To perform this operation the system first needs to know how many replicas of the file does the node Storage Peer 1 wants to have in the system. This value is configurable by Storage Peer 1. By looking at this example let us assume that he only wants three replicas on the Peer Network.

Therefore it sends the request to add the file $f$ to each of the three online nodes that have

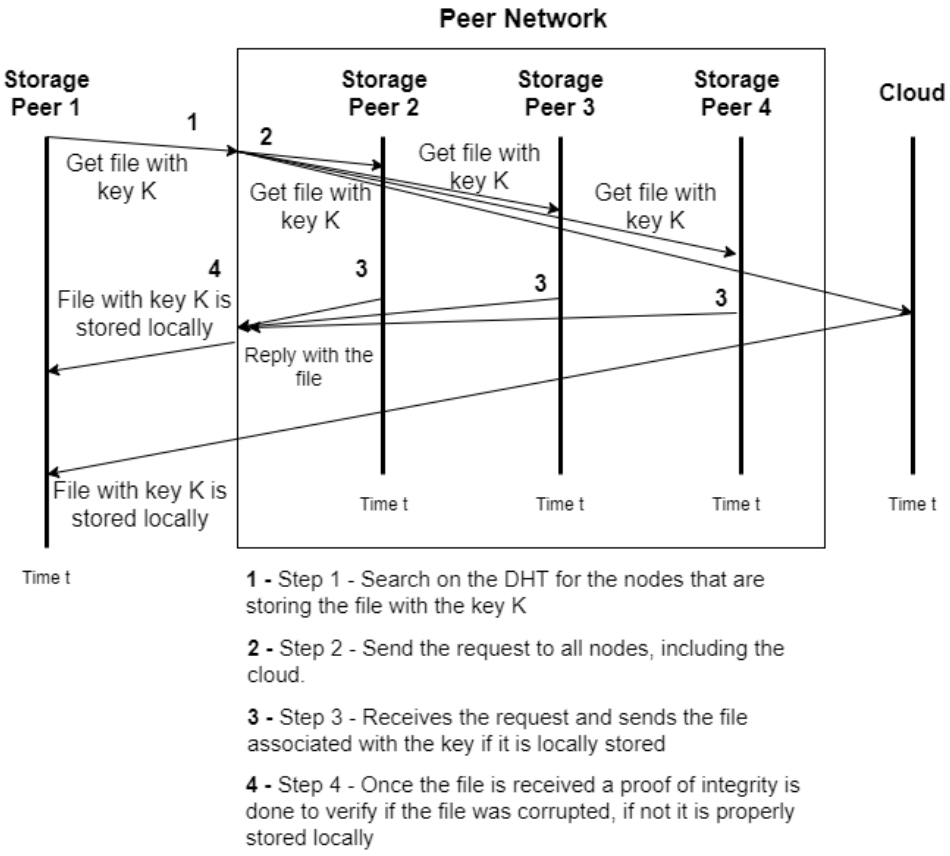


Figure 3.5: Get Operation Protocol Diagram

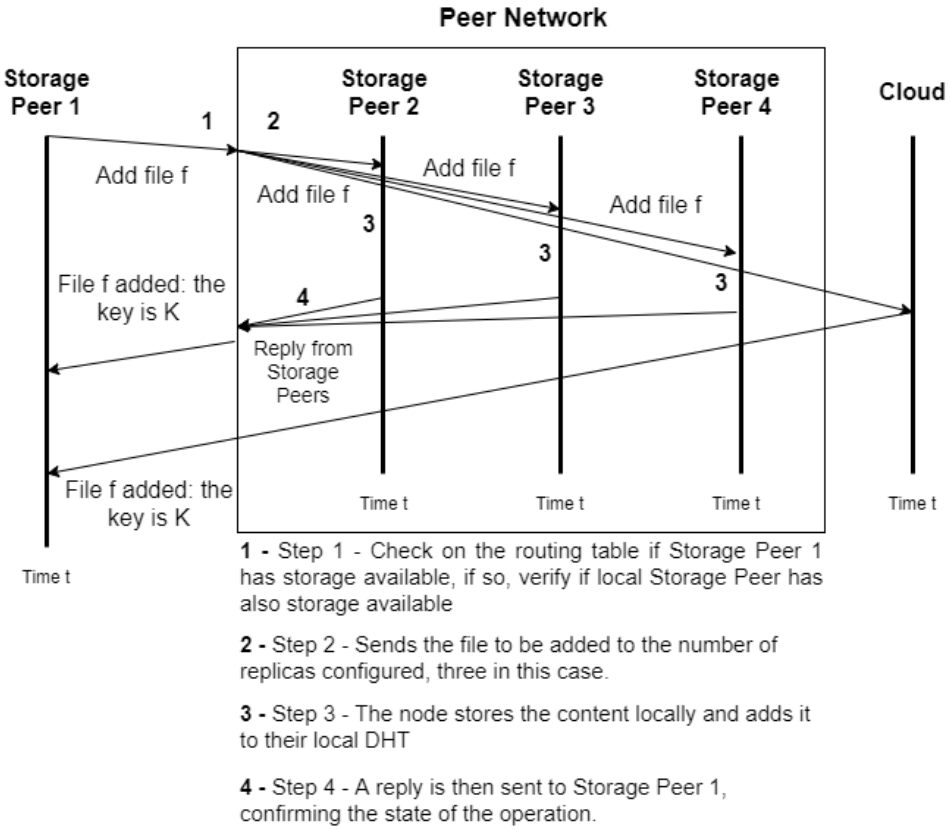


Figure 3.6: Add Operation Protocol Diagram

available storage, and are closer to him, step 2. It also sends the file to the cloud if that functionality is enabled. In our example, these nodes are Storage Peer 2, 3, and 4.

Before sending the request, the system first verifies whether or not Storage Peer 1 has the available space to store the content, if so the request is prepared for sending, step 1. What this means is that the system generates a hash of the contents of the file, renames the file to the resulting hash, and then adds it to the DHT, only once this is finished is the request sent to the other nodes. After receiving the request, each node stores the content locally and adds it to their local DHT, step 3. The file is stored in a folder with the node ID of the requester (Storage Peer 1 in this case) as the name. Also, the available storage for the Storage Peer 1 is updated to include the addition of the file. It works by multiplying the file size for the number of replicas in which the file was stored and then add this value to the already used storage for this node. For example, if a node $A$ wants to add one file $f$ of 1MB to four different nodes, and if it does so successfully, the used space of node $A$ is updated by multiplying 1MB by four, which is 4MB and then add this value to the already used space of node $A$.

Finally, Storage Peer 1 receives a reply from the system with the confirmation of the addition of the file into the system and the respective URL of the file (the key of the file), step 4. This key/URL could be later used to read the file, delete it, or update it on the system.

#### Delete Operation

To delete a certain file from the system an URL must also be given to the system. With this URL the system will first remove the file entries from the DHT and then remove the file from the storage system, it is done in this order to ensure that if a client wants to read a certain file during its deletion it will not be able to find it on the DHT even if it still exists in some node in the network.

In Figure 3.7 we have a diagram of the protocol for the Delete operation. The node Storage Peer 1 is attempting to delete a file with a key $K$ from the system. To be able to do that he searches on the DHT for the nodes that are storing the file with the key $K$, represented in step 1, in our particular case these nodes are at Storage Peer 2, 3, and 4 and sends the request to each of them, step 2. If the cloud functionality is enabled a request is also sent to the cloud.

Before sending the request, the system deletes the file from the DHT. Upon receiving the request each of the Storage Peers deletes the file locally, step 3. Afterward, each node sends the reply to Storage Peer 1, step 4.

Also, the available storage for the Storage Peer 1 is updated to include the deletion of the file. It works by multiplying the file size for the number of replicas in which the file was stored

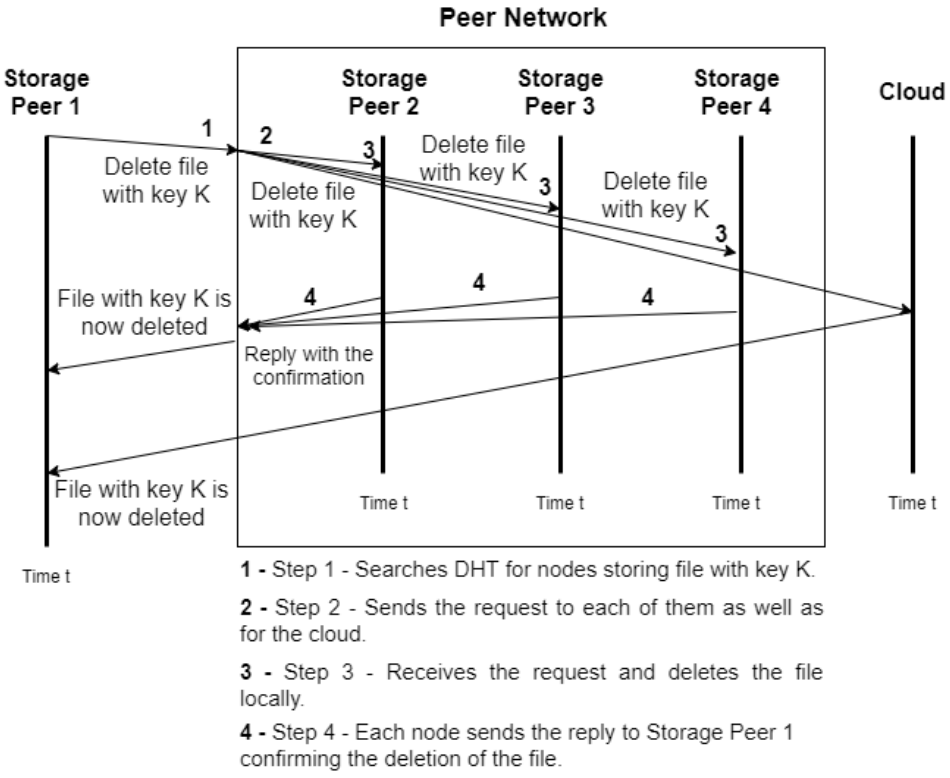


Figure 3.7: Delete Operation Protocol Diagram

and then subtract this value to the already used storage for this node. The used store value is stored in the routing table.

Finally, Storage Peer 1 receives a reply confirming that the file was deleted, from all nodes.

#### Update Operation

For the update operation, a client gives the URL of the file and the new file it wants to add to the system, replacing the file with the updated one. Afterward, the system replies with the new URL. We do not describe the protocol for the update operation because it consists simply of a delete operation followed by an add operation.

### 3.3.5 User Interface

For simplicity the user interface is terminal-based. However, it is possible to create a Web interface for the system, considering that the back-end remains the same, or very similar.

Each Storage Peer to interact with the system to launch a new terminal and start the program. Then he is presented with the P2Cstore terminal. This terminal responds to a set of commands/operations. For instance, it is possible to perform all the previously described operations easily. Note that bss-terminal stands for Blockchain Storage System terminal, hence bss-terminal. For example, to perform the add operation the interaction with the system inter-

face would be the following:

```
bss-terminal: add
File path: Desktop/file.txt
File added: FILE_KEY
bss-terminal: ...
```

In the example above, the system user types add which is responsible to perform the add operation. Then the user is presented with a request for the path to the file that the user intends to add to the system. In this particular example, the path is *Desktop/file.txt*. Then the user receives the key of the file that was just stored. This key, as previously described is the hash of the content of the file.

Some extra commands are available to the users, for example, the *help* command that lists all the possible operations. These commands are mainly to allow the user to know his situation in the system, for example, he can list the files that he has stored. In the A a description of the interface of each operation is presented.

## 3.4 P2Cstore Proof-of-Storage Algorithm

In this section, we will present the algorithm for proof-of-storage (that we designate PoS for short, but not to be confused with Proof-of-Stake). The rationale behind the necessity for such an algorithm is to prevent free-riding attacks, namely an attacker that performs such an attack benefits from the system without contributing its fair share. Systems that suffer from this vulnerability either run at reduced capacity or collapse entirely because the costs of the system weigh more and more heavily on the remaining honest peers encouraging them to either quit or free ride themselves.

In the algorithm nodes play two roles: a *prover*, $P$, that is a node attempting to convince a *verifier*, $V$, that it ($P$) is storing some data, $D$. $V$ issues a challenge, $c$, to $P$ that answers it with a proof $\pi$, according to the scheme in question.

Consider a node that wants to check if all its files are replicated in other nodes. First, for each file $f$ it has stored in the system, $V$ generates a random array of bytes that represent positions of the file; the size of the array can be configurable and the positions can be repeated on the array, meaning that we can have index 1 several times on the array. Afterward, $V$ generates a nonce (to prevent replay attacks). This nonce corresponds to a random string of configurable size. Then, $V$ creates a challenge object $c$ containing the byte array and the nonce, sends $c$ to the node(s) that is(are) storing $f$, and initiates a counter. A node that receives that request

plays the role of the prover. Once a prover $P$ receives the challenge $c$ it will reply with the bytes corresponding to the positions given by the list of bytes, and concatenates the result with the nonce in the challenge.

Afterward, $P$ executes a hash function on the resulting string. This hash is then sent to $V$. Once $V$ receives the hash it will verify if it was sent in the available time-frame, if yes, and if the response is correct then $V$ has proof that $P$ is storing file $f$ properly. If the response was not sent in the available time-frame the node down counter is incremented by 1. If this counter reaches the threshold $T_f$ (e.g., $T_f = 5$) the node is considered faulty. If the response is not correct then $V$ will handle this node as being faulty. If a node is considered faulty the system handles this case by marking locally (in a local file) the node as faulty. Afterward, $V$ removes the files that are storing that belong to $P$. Once this is done $V$ will need to update the DHT. It does so by adding the files again into the system while ignoring the faulty nodes. This way the files will be replicated among the number of nodes that they configured.

The time counter was added to protect the system against outsourcing attacks. Upon receiving challenge $c$ from a *Verifier* $V$, a malicious prover $M$ can quickly fetch the corresponding file $f$ from another storage provider $P$ and produces the proof, pretending that $A$ has been storing $f$ all along. By using the timer we can prevent such attacks by simply making attackers distinguishably slower than honest provers responding to challenges [BDG17].

On the Algorithm 1 we can see the four functions that implement the PoS algorithm on the *Verifier* side. Namely, we have the *storageProofRequest* (lines 1-21) which is the main function of the algorithm. Here we call the function *generateChallenge* (lines 34-45) that will generate the challenge as previously described. Then for each of the nodes that are storing the file we send the challenge, start the timer, and receive the response. Afterward, we verify whether the response arrived on the available time; if not then we increment the down counter by 1 and if this counter reaches a certain value $n$ we consider this node faulty and call the function *handleFaultyNode* (lines 23-26) which will handle this case. After this we call the function *checkReplicationUpdateDHT* (lines 28-32) that will update the DHT according to the nodes that are now considered faulty, ignoring the faulty ones. If the response arrived on time then we have to verify the correctness of it. If it is not correct that we call the *handleFaultyNode* (lines 23-26) and the *checkReplicationUpdateDHT* (lines 28-32) to mark the faulty nodes and update the DHT accordingly. Finally, if everything is all right and the verifications were successful we proceed to the next node.

On Algorithm 2 we can see the function that makes the PoS algorithm on the *Prover* side. This function is the *handleProofOfStorageChallenge* (lines 1-11). This function receives the

```
1  Function storageProofRequest(file):
2      call function generateChallenge
3      for each node that is storing the file do
4          send challenge to node
5          start timer
6          get response from node
7          if response time greater than time available then
8              /* Assume node temporarily unavailable                          */
9              increment node down counter by 1
10             if node down counter equals n then
11                 /* n is configurable                                        */
12                 call function handleFaultyNode
13             end
14             call function checkReplicationUpdateDHT
15         end
16         else if response is not valid then
17             call function handleFaultyNode
18             call function checkReplicationUpdateDHT
19         end
20         /* Otherwise everything ok, continue                                */
21     end
22
23 Function handleFaultyNode(nodeInfo):
24     mark locally node as faulty                  /* Local file stores faulty nodes */
25     remove files of faulty node
26     return
27
28 Function checkReplicationUpdateDHT(fileInfo):
29     /* This works like a new add, removes the previous information on the DHT and adds
          the file to the system ignoring the faulty nodes                     */
30     remove file from the system
31     add file to the system
32     return
33
34 Function generateChallenge(fileInfo):
35     create a list of bytes of size n
36     while list of bytes size equals 0 do
37         generate random(file size -1)
38         /* random can be from 0 to file size                                */
39         if random number is an odd number then
40             add i to list
41         end
42     end
43     generate a random nonce
44     generate the challenge with the list of bytes plus the nonce
45     return
46
```

**Algorithm 1:** Verifier Functions – PoS Algorithm

```
1  Function handleProofOfStorageChallenge(challenge, fileInfo):
2      get byte list from challenge
3      get nonce from challenge
4      get file from fileInfo
5      for each byte i in list do
6          get byte of position i of file
7          convert byte to character add character to response array
8      end
9      challenge response equals response array plus nonce
10     send the hash of the response to verifier
11     return
12
```

**Algorithm 2:** Prover Function – PoS Algorithm

challenge sent by the *Verifier* and obtains the positions list, the nonce, and the file. Next, it will get the respective position content from the file associated with the position on the list and make an array. Once all the positions of the challenge list are converted to characters of the file in a string we add the string plus the nonce creating the response or proof. Finally, we send the hash of the response to the *Verifier*.

As presented, the scheme requires the *Verifier* node $V$ to keep its copy of the file. Although this makes sense in some cases, e.g., if $V$ is a university that stores its certificates in other nodes only for replication purposes, generically it is undesirable. To remove this limitation, before storing and deleting a file $f$, the $V$ has to generate a bag of challenges $B_f = \{c_1, c_2, ...c_m\}$ and use these challenges one by one when needed (each one only once). When no challenges are left, $V$ has to download $f$ and generate a new bag of challenges.

### 3.4.1 Attacks prevented by the PoS Algorithm

In this section, we briefly discuss how our PoS algorithm handles the two attacks that we are trying to prevent, namely the free-riding attack and the retention of information/cheating attack.

The first attack is the free-riding attack. As mentioned before it is of paramount importance that our algorithm can prevent such attacks. It does so by using the PoS algorithm. This algorithm incentives nodes to cooperate with the system. It does so by allowing our system to identify nodes that are doing free-riding attacks and that is not storing any data. Every time a malicious node fails to send the proof-of-storage to the verifier, the verifier will delete all the files from its storage that belong to the malicious node and stop interacting with it. This way, every node is incentivized to cooperate with the system, otherwise, it will eventually be ignored (considered faulty) by all correct nodes.

Another important attack that the system should be able to prevent is the outsourcing attack. In such an attack, the attacker $A$ receives a challenge $c$ from the verifier $V$ to assess whether $A$ is storing the file $f$. Upon receiving the challenge, $A$ immediately requests the $f$ to

a storage provider $P$ and generates the proof, sending it to $V$ attempting to convince it that it is indeed storing $f$. If this happens in our system, we assume that there is a maximum time for the node to solve the challenge, which is $T$ seconds, the attacker node will not be able to solve the challenge in time ($T$ seconds) and the system will consider the node as faulty. Also, the attacker node will eventually be considered faulty in all nodes and will not be able to store any files on them.

Therefore our algorithm attempts to enforce/incentivize cooperation between all participants attempting to prevent both attacks while ensuring the system sustainability through the storage by having a storage scheme in which a node can only use the same amount of storage that gives to the system.

## 3.5 Summary

Throughout this chapter the P2CSTORE's design was at focus, an explanation was done on how each major component works in the system and how each of the major operations protocols was implemented.

The system satisfies our requirements, in particular, the requirement of ensuring data integrity is ensured by performing an integrity check upon getting the data from the system. The requirement of data freshness is ensured because only the newest version of the file is kept in the system. Lastly, the data availability requirement is ensured depending on the configurations of the user, but our system allows for both P2P and Cloud storage providers. By using both architectures we offer a wider range of options and more storage redundancy reducing the probability of the system becoming unavailable.

# Chapter 4

# Evaluation

In this section, we evaluate P2Cstore. The goal of our evaluation is to answer three main questions:

1. Which is faster to use the P2P storage system or Cloud-based storage system?
2. What are the costs of using multiple clouds to store data versus P2P systems or the Ethereum blockchain itself?
3. What is the cost of the PoS mechanism?

We evaluated the P2Cstore system regarding the time it took to perform each operation and the PoS algorithm.

This evaluation does not include education certificates. Our system functions parallel to blockchain applications. Therefore our evaluation scenario will focus solely on the P2P + Cloud storage system. Our solution is designed to have files be easily referenced through for example some kind of smart contract of the Ethereum blockchain. If for instance, someone wanted to use our solution to store Education Certificates, as previously described, it could simply reference the files and access them through the system or directly through the blockchain by performing a call on a smart contract. But these examples are external to our solution and could be for any type of blockchain application which is why they were not tested.

## 4.1 Experimental Evaluation

In the experimental evaluation, we chose to split the test scenarios between P2P, P2P with Cloud, and P2P with PoS. We divided the testing into three categories because they compose the important aspects that will allow us to answer the previously asked questions. These test scenarios will give a clear view of the performance of the system's major components separately.

It is worth explaining that the previously asked questions focus mainly on the performance differences between the P2P storage system and cloud-based storage system, the costs associated when compared with the benefits, and finally, the necessity of having a PoS viewed from a performance standpoint which is why the test scenarios are going to helps answer these three questions.

For the clouds, we used S3 [Amaa] and GCP [Gooa] because they are the most common and also because they are included in the library JClouds [jcl].

We performed all the operation tests with sets of 100 operations, namely the add, get, delete operation tests. For the PoS algorithm tests, we did sets of 100, 150, and 200 operations for file sizes of 5Bs, 100KBs, 1MBs, and 10MBs. We choose these values because they represent the usual file size of an education certificate, or even of a PDF file, from a few KBs to a few MBs. In all the experiments below, we report the average time taken for these operations to complete. We decided not to test the update operation because it simply is the combination of a delete operation followed by an add operation. For the evaluation environment, we used the PlanetLab Europe[pla] test network. We chose this test network because it has a set of nodes across Europe which allowed us to test the system in a realistic deployment.

The nodes that we used were in different countries to increase geographic diversity and hence obtain results closer to a real deployment. Client nodes were located in France and the Netherlands, and server nodes were located in France, Italy, Canada, Portugal, Ireland, and Sweden.

In the following subsections, we will discuss the obtained results, draw our conclusions, and attempt to answer each of the above questions.

### 4.1.1 Operations Latency

In this section, we measure the latency of each operation, i.e. the time it takes since a client issues a request until it sees that the operation is completed successfully. We varied the following parameters: file size (5Bs, 100KBs, 1MBs, and 10MBs), and a different set of nodes, meaning that we used servers in different countries like Germany, Italy, Portugal, Netherlands, Canada, and Switzerland. We only used a set of 100 operations as mentioned above.

The color scheme of all the four graphic legends indicates the file size variations between tests and helps on the analysis and interpretation of the graphic.

Due to the different magnitude of the results, all graphics below are on a logarithmic scale except the one in Figure 4.4.

### Add operation

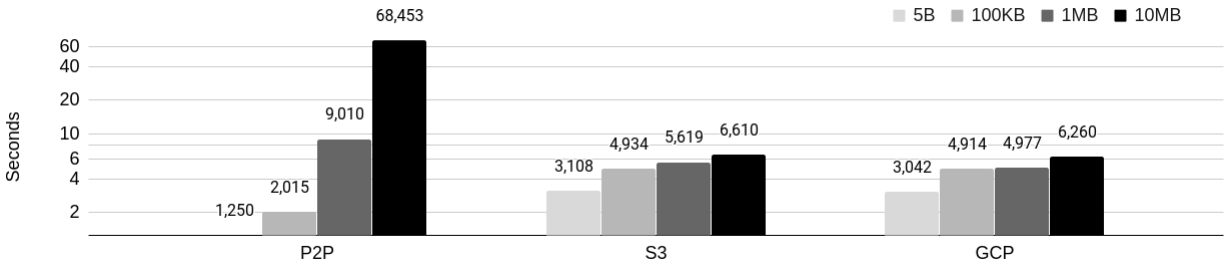


Figure 4.1: Results for 100 add operations for different file sizes.

The X-axis represents the results of the operations for each of the test scenarios of the add operation. For instance to find the results of the test P2P for files with 10MBs we go to the P2P section on the X-axis and find the black bar. The Y-axis represents the time spent in seconds.

As we can see in Figure 4.1, for larger files the time it takes to write both in the P2P and clouds increases. The increase is very huge on the 10MB file for the P2P because the code was not written concurrently for this operation. This is the main reason that we are seeing such a huge difference when compared with the smaller sizes. Because for each of the 5 nodes used for storage purposes the data is sent by the writer node then it waits, and only when everything is done it will write on the next node and so on. As we can see on the clouds the increment is much smaller because we are writing to the cloud once, instead of the P2P in which we are writing 5 times due to the number of nodes we want to store the data into. This is the reason behind the fact that we can see a great increase in the P2P and a smaller one on the clouds.

### Get operation

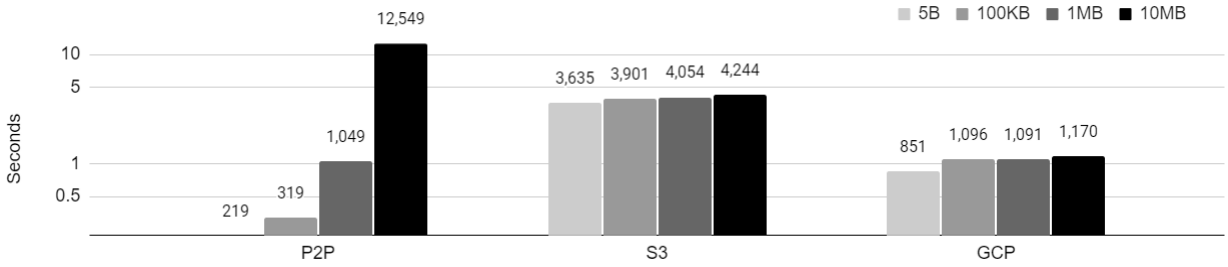


Figure 4.2: Results for 100 get operations for different file sizes.

As we can see in Figure 4.2, for larger files the time it takes to write in the P2P and on both clouds increases. The increase is greater on the 10MB files for the P2P because our implementation does not exploit concurrency. In fact, for each of the 5 nodes used for storage purposes

the writer node to send the data, wait for the operation to complete, and only afterward start writing on the next node. This is the main reason that we are seeing such a significant difference when compared with the previous one.

### Delete operation

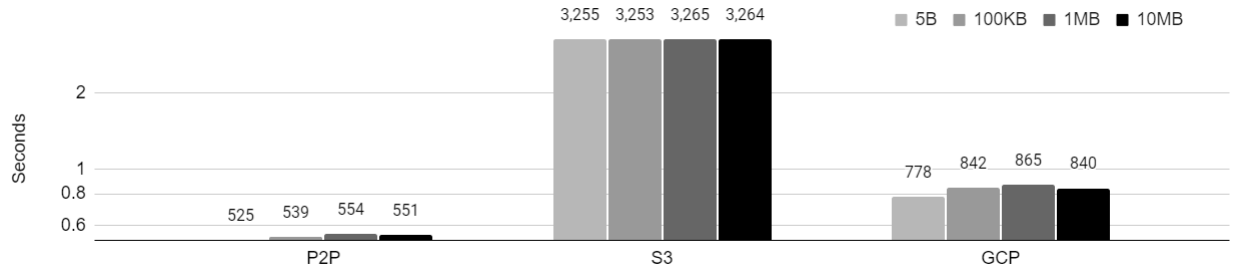


Figure 4.3: Results for 100 delete operations for different file sizes.

As we can see in Figure 4.3 for larger files the time it takes to delete from the P2P and both clouds remain roughly the same. The increase is residual although the code was not written concurrently for this operation. This is the case because the cost of deleting a single file is very small, even for larger files, therefore the increment associated with not using concurrency is residual, but if it was used the results could still improve. Interestingly, S3 takes a substantially large time to perform the delete operation when compared with GCP, and this time is roughly constant regardless of the file size. This is because in S3 a delete corresponds to the insertion of a *delete marker* [S3D], and in fact, one can observe that the deletion time is similar to the insert time for small files (Figure 4.1).

### PoS Algorithm


Figure 4.4: Cost of the PoS for different file sizes and total number of files (configuration with P2P storage only).

This test is different from the previous ones because it only applies to the P2P system. After

all, it is used to ensure that the peers are storing the content that they promise to, as previously explained. Also in Figure 4.4 the graphic is not on a logarithmic scale like the others. This test was not done for the clouds because they do not execute any code, therefore there is no way for us to send a challenge to the clouds, they would not be able to solve the challenge without executing code.

On the obtained graphic the X-axis represents the results of the operations for each of the test scenarios of the PoS algorithm test. For instance to find the results of the test for 100 Files with 10MBs we go to the 100 Files section on the X-axis and find the black bar. The Y-axis represents the time spent in seconds.

As we can see in Figure 4.4 for larger files the time it takes to perform the PoS algorithm, and respective reply to the verifier, increases but not by a lot because the file size also increases, and on the challenge generation the file size is used to generate the random file. The challenge size consists of a list of bytes with 16 positions in which each one points to a respective character in the file. This way even for larger files the time it takes to generate the challenge will not increase significantly because the list size is 16. However, it is worth reminding that this value is easily configurable.

### 4.1.2 Discussion

In this section, we will answer the previously asked questions and try to draw a few conclusions.

***Which is faster to use P2P storage system or Cloud-based storage system?***
After analyzing the results we can conclude that for smaller files the P2P is faster, however, this depends on the nodes that compose the system. For different nodes, either with faster CPUs or with fast network connectivity or that are close to each other the obtained results could be different. It is also important to point that for the clouds the scenarios are similar. If the cloud server that we are accessing is closer or further away the time it takes to communicate with it also decreases or increases, respectively. But it is also important to point out that the code to write, get and delete was not done concurrently on the P2P level, therefore the times could be easily reduced by making the code operate concurrently. This improvement would alter our conclusions because for larger files the time it takes to add and get files from the P2P would reduce significantly, making the P2P a better solution overall.

***What are the costs of using multiple clouds to store data versus P2P systems or the blockchain Ethereum itself?***
This question is very interesting namely because the cost differences are huge between the three

scenarios. The cost of storing a 256bit on the Ethereum network is 20000 gas. The cost of gas at the time of writing this document is 105 Gwei and each Gwei is 0.000000001 ETH [Woo14, etha]. We can easily calculate the cost of storing a file with 1KB [ethb]. Considering that 256bit costs 20000 gas then 1KB costs 80000 gas which gives us a transaction fee (Fiat) of $3.00352. The cost of storing data on Amazon S3 depends on the region and it also varies depending on the space used. It is paid monthly and it depends on the usage and amount of data stored [s3P]. We experimented to attempt to find out the cost of storing certain values in the S3 storage bucket, and we discovered that using the Price Calculator [pri] for 100GBs a month, 10000 requests of either PUT, COPY, POST, LIST, 10000 requests of either GET, SELECT and considering a data returned of around 50GBs and 50GBs of scanned data it would cost around 2.49 USD for the US East Ohio region for the S3 Standard cost, as represented in the Figure 4.5. For GCP, the cost is similar to S3, meaning that there is also a monthly cost depending on the usage and amount of data stored [gcp]. Finally, the cost of having a P2P node machine connected has an associated electricity cost. This value depends on the country and region as well as the machine that is being used. In conclusion, the cheapest solution seems to be the P2P storage system because the only associated costs are the ones related to the fact that the storage machines must be connected for the system to operate. But the price of S3 and GCP includes the costs that both AWS and Google have with their always-connected servers. That and for larger files, the cost of electricity would not vary, but both on the clouds and the Ethereum network the costs will increase as the file size increases as well.

***What is the cost of the PoS mechanism?***

Overall we believe it is worth having the PoS mechanism implemented in the system. The advantages are greater than the possible drawbacks. Namely the fact that with this mechanism we can ensure that each node is only using the amount of storage it gives to the system. And also we can ensure that at the PoS time if a node is not storing the content as he was supposed to it will be marked as faulty and discard by the verifier node for future interactions. This mechanism prevents the system from collapse by incentivizing cooperation among nodes and sharing resources while punishing those that do not comply with this as previously described. The one thing that is worth discussing further is the frequency that the PoS mechanism will be executed. Well in our system this is configurable, therefore for different applications, different values could apply better than others. It is worth mentioning that making PoS requests consumes system resources, therefore the frequency of PoS requests should be properly considered according to the type of application that is being used. One example is the Education Certificates. In this

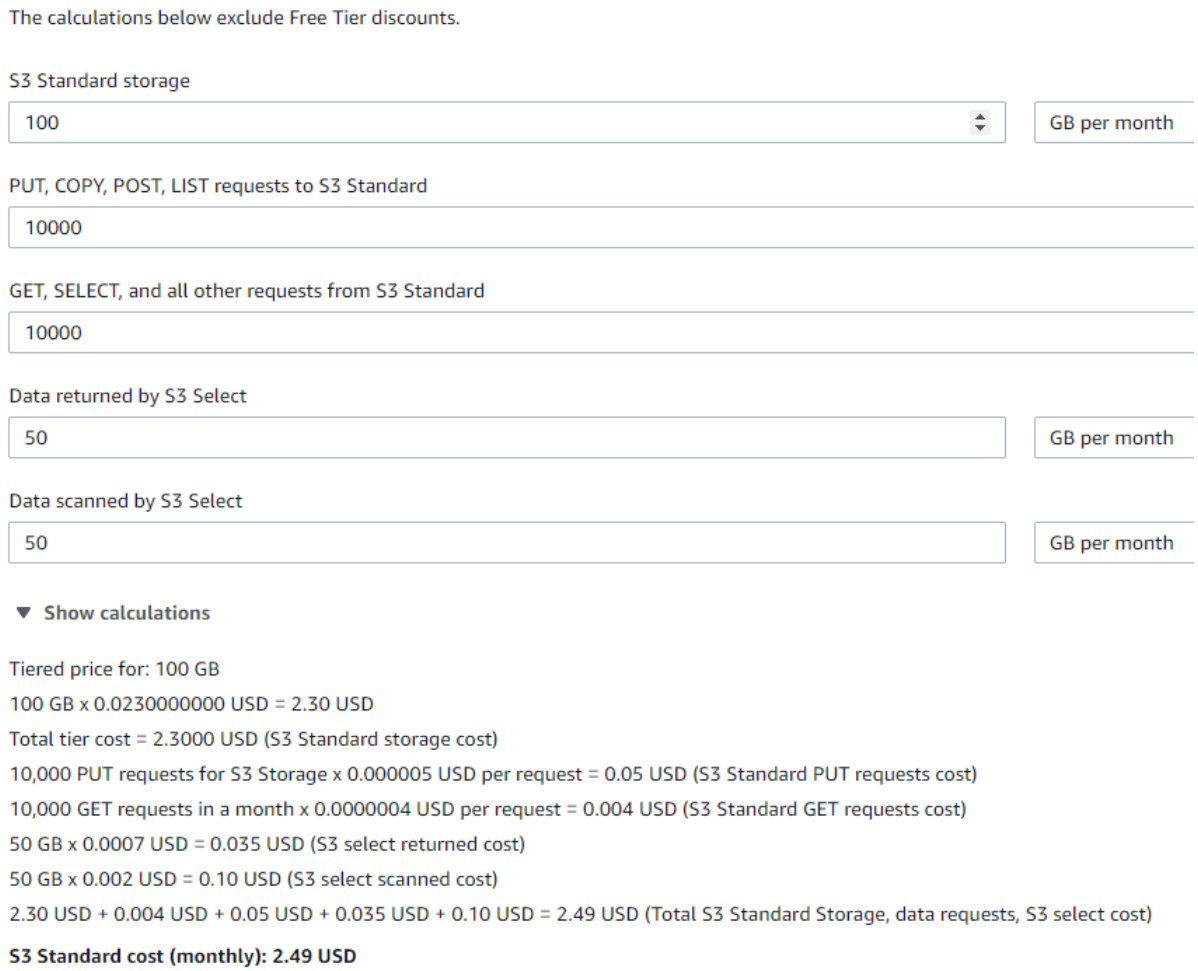


Figure 4.5: Cost of the S3 storage bucket. [pri]

case, it is not necessary to do several PoS requests, one a day should suffice. However, if some education certificate is being requested more often, maybe more PoS requests could be useful for that particular file, and the same for the opposite case, if a certain certificate is not requested for a long time, maybe performing fewer PoS requests is a good idea to save up resources. In conclusion, this mechanism helps to ensure the system's sustainability and it is an important component of our system, however, it is up to the developers/administrators to configure how often is worth it to perform this operation for each file/node.

Lastly, it is worth mentioning that using this solution to work in parallel to a blockchain application would not interfere with the previously obtained results. Our solution is designed to have files be easily referenced through for example some kind of smart contract of the Ethereum blockchain. If for instance, someone wanted to use our solution to store Education Certificates, as previously described, it could simply reference the files and access them through the system or directly through the blockchain by performing a call on a smart contract. But these examples are external to our solution and could be for any type of blockchain application which is why

they were not tested.

Overall the system operates at the expected speed. However, it is possible to improve a few aspects, namely, make the add, get, and delete operations concurrently. This way we can easily improve the overall performance of the system.

## 4.2 Correctness Argument

In this section, we will perform a qualitative functional evaluation of the P2Cstore. We will assess if the P2Cstore fulfills the requirements previously described. If not, why and if yes, how does the system P2Cstore makes it possible.

### 4.2.1 System Requirements

As described in Chapter 3 there are a set of requirements that the system is supposed to ensure. Considering that it is not practical to test them quantitatively we will do it qualitatively. We will explain how we ensure each of the system requirements, in particular:

#### Data Freshness

This requirement is ensured because whenever a file is added to the system, it is created a new entry for it on the DHT. Every time a new operation to either delete or update the content of the file takes place the old file will be deleted, and for the update, a new file will be added. These changes occur both on the physical storage as well as on the DHT, this way we ensure that the data presented to the user in case of a get operation happens is always correct and up to date.

#### Data Integrity

This requirement is of paramount importance to our system. Considering that the major motivation for our work is the storage of Education Certificates it is essential to ensure that the certificate was not corrupted in any way. We do this by performing an integrity check upon getting the data from the system. When we add a new file to the system the hash of the file's contents is generated and that becomes the file key to access the file. Once someone performs the get operation to get the file and read its contents the system verifies if the hash of the contents of the file retrieved from the system is the same as the file key that is on the DHT. If the values match, then the integrity of the file is good. Otherwise, the file is considered corrupted and is rejected by the system. This way if anyone attempts to change the file contents or manipulate data, the system will be able to spot it and reject those changes.

#### Data Availability

This requirement is more relative to each situation. Our system is prepared to have both cloud storage providers as well as a configurable number of peers decided by each user to increase availability but, if a user for example intends to simply use two storage peers to store its data and no clouds it is easy to say that the system could easily become unavailable. However, if a user uses five storage peers and two cloud storage providers to store its data it would be much more unlikely that this user finds the system unavailable because it has seven different sources to retrieve its data and access the system.

In general, the availability of the system depends mostly on the user configurations and whether or not the storage peers that were used to store the data become offline often or not. But even if for instance a node is the majority of the time offline and not providing the service our PoS algorithm can mark it faulty and relocate the data stored on that node to another. With this measure, we also increase availability.

## 4.3 Summary

The generic goals of our work were that we should be able to ensure all system requirements, be able to implement the solution as described, and finally that the system operations performance was reasonably good. As described above it is safe to consider that the system requirements were fulfilled. While developing this work we manage to implement some extra features that make the system more interesting and more robust overall.

One such example is our PoS Algorithm. As previously explained this algorithm prevents free-riding attacks and improves the system sustainability. Due to the PoS Algorithm, any node is incentivized to work with the system and participate fairly than to work against the system and attack it in some way.

Another example is our storage space exchange mechanism. This mechanism operates alongside the PoS Algorithm. With this mechanism, any node that wants to use the system has to provide the same storage it intends to use. This way we can maintain the available storage always guaranteed.

These features help the system to be more robust, sustainable, and resilient to attacks.

# Chapter 5

# Conclusions

Blockchains can only store small amounts of data. To store larger files it is necessary to have some kind of storage system, parallel to the blockchain. In this document, we proposed a Storage System for Blockchain.

This work presents the P2CSTORE system, a storage system based on P2P and Cloud Storage that uses the best of both. It uses both to improve the availability of the system as well as the cost, giving to the developer/administrator the flexibility to manage the trade-offs of using P2P and Cloud Storage Systems. This system attempts to solve the need introduced in Section 1 which is the fact that blockchains can not store large amounts of data, only hashes.

The document starts by presenting several related works that will help with the development of the project. We analyzed several systems as well as the blockchain in particular Ethereum. In the next section, an architecture was proposed to try to solve the defined problem. Afterward, a strategy was delineated on how to evaluate the proposed architecture once implemented.

## 5.1 Achievements

This dissertation reached two important achievements. The first achievement is the development of the P2CSTORE system, which can leverage both cloud and P2P technologies to create a storage system for DApps. The second achievement was the fact that we were able to ensure all of our initial system requirements, data freshness, data integrity, and data availability. The third achievement worth mentioning is the fact that we were able to improve upon our initial goals, in particular, we created a PoS Algorithm and a Storage exchange mechanism to improve the system robustness and sustainability.

We also consider an achievement the fact that, as mentioned in the Chapter 1, a version of this work was published as *P2CSTORE: P2P and Cloud File Storage for Blockchain Applications*

for the 19th IEEE International Symposium on Network Computing and Applications (NCA 2020).

## 5.2 Future Work

The design and implementation of P2Cstore can be extended and improved in several different ways. A smart contract that would be responsible to reference the hash of transactions to the respective node on P2Cstore could be done on the Ethereum blockchain. This could be used for the education certificates, meaning that when some university issued a certificate it could store the file in our system and make an Ethereum transaction holding a smart contract that would point to the file location on our storage system. This suggestion would allow for the usage of our storage system to store education certificates parallel to the Ethereum blockchain. This is interesting because it allows for the verification of the certificate authenticity on the blockchain, like proposed by Serranito et al. [SVGC20], as described in Section , and for the storage of the actual certificate in our storage system.

Another natural improvement is to make the processing of the operations concurrent on the client-side to reduce latency.

On the PoS Algorithm, a requester has to have the file to generate and verify the challenges is a drawback, this is a problem because the requester can lose the file and if it does so it has no way to request a PoS from the storage peers, instead, it will have to trust them, get the file and then perform the PoS. To tackle this issue we propose for future work that on the add operation of the file to the storage system the verifier, which in this case is the node adding the file, generates a number of challenges and uses them for the proofs. To prevent the repetition of proofs the verifier node, periodically, should request the file to the system and regenerate a new set of proofs and discard the used ones.

Another possible improvement to P2Cstore could be the addition of TLS to enhance the confidentiality of communications. More cloud storage providers could be added to the system.

For the update operation, it is interesting to add some sort of locking mechanism to prevent readers from seeing the old version as the newer version is written into the system.

Another relevant improvement is regarding security, for instance, if a malicious storage peer $M$ attempts to deceive the system by performing a wrong used storage update he can do so. This is possible because the update of that value on the routing table is based on trusting the writer node, which is the node that performed the operation. The prevention of this attack is possible by, for example, having a verification done by the nodes storing the file contents when an update is done on the routing table. This is possible if the nodes know how many

replicas there are in the system for that particular file of the node $M$, if so they can perform the necessary calculation and verify if the result is correct, if not they can update it with the proper value after some kind of consensus is achieved among the storage nodes that are storing the file.

The other important improvement is on the node registration. In the present version, we have a local file from which we read if a node is allowed or not to join the network. However, it would be more interesting to move this file to the BootStrap Nodes. This way, only these nodes can register users into the network and verify if they are allowed to join or not.

# Bibliography


[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[AKK09] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. Proofs of storage from homomorphic identification protocols. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 319–333, 2009.

[Amaa] Amazon S3. `https://aws.amazon.com/s3/`.

[Amab] Amazon s3 description. `https://aws.amazon.com/s3/?nc2=type_a`. Accessed: 2020-12-13.

[ANSF16] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 181–194, 2016.

[AW18] Andreas M Antonopoulos and Gavin Wood. *Mastering Ethereum: building smart contracts and dApps*. O'Reilly Media, 2018.

[BCQ+13] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[BDG17] Juan Benet, David Dalrymple, and Nicola Greco. Proof of replication. *Protocol Labs, July*, 27:20, 2017.

[Ben14] Juan Benet. IPFS-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[BG18] J Benet and N Greco. Filecoin: A decentralized storage network. *Protoc. Labs*, 2018.

[BJO09] Kevin D Bowers, Ari Juels, and Alina Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198. ACM, 2009.

[BM07] Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure key-based routing. In *2007 International Conference on Parallel and Distributed Systems*, pages 1–8. IEEE, 2007.

[But13] Vitalik Buterin. Mastercoin: A second-generation protocol on the bitcoin blockchain. *Bitcoin Magazine (4 November 2013)*, 2013.

[C+11] B. Calder et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[CDK05] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Pearson Education, 2005.

[DA99] Tim Dierks and Christopher Allen. The TLS protocol version 1.0 (RFC 2246). IETF Request For Comments, January 1999.

[dC14] Nuno Tiago Ferreira de Carvalho. A practical validation of homomorphic message authentication schemes. Master's thesis, University of Minho, 2014.

[DKK+01] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings 18th ACM Symposium on Operating Systems Principles*, pages 202–215, 2001.

[droa] Dropbox. `https://www.dropbox.com/?_hp=c&landing=dbv2`. Accessed: 2020-10-31.

[Drob] MagicBox, DropBox new storage infrastructure. `https://blogs.dropbox.com/tech/2016/05/inside-the-magic-pocket/`.

[etha] Eth gas station. `https://ethgasstation.info/`. Accessed: 2020-09-13.

[ethb] Eth gas station calculator. `https://ethgasstation.info/calculatorTxV.php`. Accessed: 2020-09-13.

[FFM04] Michael J Freedman, Eric Freudenthal, and David Mazieres. Democratizing content publication with coral. In *NSDI*, volume 4, pages 18–18, 2004.

[Fis19] Ben Fisch. Tight proofs of space and replication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 324–348, 2019.

[gcp] GCP pricing. `https://cloud.google.com/storage/pricing`. Accessed: 2020-09-13.

[Gooa] Google Cloud Storage. `https://cloud.google.com/storage/`.

[goob] Google drive. `https://www.google.com/drive/`. Accessed: 2020-10-31.

[Her19] Maurice Herlihy. Blockchains from a distributed computing perspective. *Commun. ACM*, 62(2):78–85, 2019.

[jcl] Apache jclouds. `https://jclouds.apache.org/`. Accessed: 2020-09-19.

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, R. Gummadi D. Geels, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[KMK+19] C. Kontzinos, O. Markaki, P. Kokkinakos, V. Karakolis, S. Skalidakis, and J. Psarras. University process optimisation through smart curriculum design and blockchain-based student accreditation. In *Proceedings of 18th International Conference on WWW/Internet*, 2019.

[KVH20] Ingo R Keck, Maria-Esther Vidal, and Lambert Heller. Digital transformation of education credential processes and life cycles: A structured overview on main challenges and research questions. In *12th International Conference on Mobile, Hybrid, and On-line Learning (eLmL 2020)*, 2020.

[lus] Lustre. introduction to lustre architecture. `http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf`. 2017.

[MM02] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65, 2002.

[MTD19] Alexander Mikroyannidis, Allan Third, and John Domingue. Decentralising online education using blockchain technology. In *The Online, Open and Flexible Higher*

*Education Conference: Blended and online education within European university networks*, October 2019.

[mul] Multihash. `https://multiformats.io/multihash/`. Accessed: 2020-12-12.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[one] Onedrive. `https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage`. Accessed: 2020-10-31.

[Pec17] Morgen E Peck. Blockchains: How they work and why they'll change the world. *IEEE Spectrum*, 54(10):26–35, 2017.

[pla] Planetlab europe. `https://www.planet-lab.eu/`. Accessed: 2020-09-19.

[pri] Aws s3 price calculator. `https://calculator.aws/#/createCalculator`. Accessed: 2020-12-03.

[RD01] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 188–201, 2001.

[ree] Systematic codes. `https://en.wikipedia.org/wiki/Systematic_code`. Accessed: 2020-10-31.

[RM12] Eric Rescorla and Nagendra Modadugu. Datagram transport layer security version 1.2 (RFC 6347). IETF Request For Comments, 2012.

[S+07] L Strigini et al. Resilience-building technologies: State of knowledge – ReSIST NoE deliverable D12. 2007.

[S3D] Amazon s3 - developer guide. `https://docs.aws.amazon.com/AmazonS3/latest/dev/delete-or-empty-bucket.html`. Accessed: 2020-09-19.

[s3P] S3 pricing. `https://aws.amazon.com/pt/s3/pricing/`. Accessed: 2020-09-13.

[SHI+12] Sea Shalunov, Greg Hazel, Janardhan Iyengar, Mirja Kuehlewind, et al. Low extra delay background transport (ledbat). *draft-ietf-ledbat-congestion-04. txt*, 2012.

[sol] Solidity programming language ethereum. `https://docs.soliditylang.org/en/stable/`. Accessed: 2020-12-02.

[SVGC20] Diogo Serranito, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. Blockchain ecosystem for verifiable qualifications. In *Proceedings of the 2nd IEEE Conference on Blockchain Research & Applications for Innovative Networks and Services*, September 2020.

[SW08] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107, 2008.

[Swa19] Swarm. SWARM: Storage and communication for a sovereign digital society. `https://ethersphere.github.io/swarm-home/`, 2019.

[TLS+15] Haowen Tang, Fangming Liu, Guobin Shen, Yuchen Jin, and Chuanxiong Guo. Unidrive: Synergize multiple consumer cloud storage services. In *Proceedings of the 16th Annual Middleware Conference*, pages 137–148. ACM, 2015.

[Und16] Sarah Underwood. Blockchain beyond Bitcoin. *Communications of the ACM*, 59(11):15–17, 2016.

[VC14] David Vorick and Luke Champine. Sia: Simple decentralized storage. *Nebulous Inc*, 2014.

[vyp] Vyper programming language ethereum. `https://vyper.readthedocs.io/en/stable/`. Accessed: 2020-12-02.

[WBBB14] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peer-to-peer cloud storage network. 2014.

[WLB14] Shawn Wilkinson, Jim Lowry, and Tome Boshevski. Metadisk a blockchain-based decentralized file storage application. *Tech. Rep.*, 2014.

[WLL15] Boyang Wang, Baochun Li, and Hui Li. Panda: public auditing for shared data with efficient user revocation in the cloud. *IEEE Transactions on Services Computing*, 8(1):92–106, 2015.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[WRLL10] Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Toward publicly auditable secure cloud data storage services. *IEEE Network*, 24(4):19–24, 2010.

[WWRL10] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *Proceedings of the IEEE INFOCOM*, pages 1–9, 2010.

# Appendix A

# User Interface - Operations

The user interface allows for several commands, in the document we described the add command, in this appendix we will describe the other commands that correspond the the available operations. It is worth mentioning that it is possible that the order of the system messages can be different if we are using clouds or not, because as the requests are done parallel to clouds and DHT in some cases the DHT can respond faster and in others the clouds might respond faster. This is also the case if we use multiple clouds.

### Get Command User Interface

```
bss-terminal: get
File key: FILE_KEY
File donwloaded from the DHT
bss-terminal: ...
```

The file can also be download from the S3 cloud or the GCP cloud, that depends whether or not the user has the clouds enabled or not.

### Delete Command User Interface

```
bss-terminal: delete
File key: FILE_KEY
File deleted: FILE_KEY
bss-terminal: ...
```

### Update Command User Interface

```
bss-terminal: update
File key: FILE_KEY
What is the path to the new file version?: NEW_FILE_PATH
File deleted: FILE_KEY
File stored: NEW_FILE_KEY
bss-terminal: ...
```