P2CSTORE : P2P and Cloud File Storage for Blockchain Applications

Marcelo Filipe Regra da Silva
Instituto Superior Técnico, Universidade de Lisboa

***Abstract*—Blockchain is currently revolutionizing the world. It consists of a distributed ledger of transactions that can be used for several purposes, for payment processing, money transfers, 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 fact that it can only store, in an efficient way, data with small sizes because blockchain is replicated across all nodes. It is also worth mentioning that 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. The usage of blockchain to store files, for example, education certificates allow a better authenticity assessment of these files. We show the benefits of the approach with a blockchain application that manages education certificates. The application stores hashes of the certificates in the cloud and the certificates themselves in our storage system. In the final sections, we made a system evaluation that corroborates the fact that such a system with both P2P and cloud is relevant and it works. Therefore, a combination of both is possible and efficient, but it also improves availability without compromising system performance.**

I. 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. One example of such a system is a *blockchain* [1]. Blockchain is a promising new technology that is generating a vast interest world-wide, Portugal included [1].

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 enforce the authenticity of the stored data. 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.

As an example, project QualiChain (https://qualichain-project.eu/) is developing a blockchain-based system to enforce the authenticity of university certificates, [2]. 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* [3], 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.

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* [4]–[6], as many blockchains are also P2P systems [7], [8].

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 that will be described later in this document.

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 [9], this architecture is provided by many *cloud storage services* [10], [11], that ensure high-availability but charge for bytes stored and downloaded. 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 The 19th IEEE International Symposium on Network Computing and Applications (NCA 2020).

A. 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. We will also use a proof-of-storage (PoS) mechanism that allows a client to check if a server has a file without downloading that file.

B. Document Outline

The remainder of the document is structured as follows. In Section II we discuss related work that is relevant to our solution. Section III 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. Section IV evaluates the solution that we propose and discusses the associated results. Finally, Section V briefly concludes this document.

II. Related Work

In this section we briefly discuss the state-of-the-art related to our work in particular, P2P storage systems, cloud storage systems and storage for DApps. We also present related work on Proof-of-Storage systems and our use case, i.e. storage of education certificates in the blockchain.

A. Peer-to-Peer Storage Systems

P2P systems represent a distributed systems paradigm whereas data and computational resources are contributed by many volunteer hosts to provide a shared state. In this type of system, all nodes contribute to computational resources like disk storage and CPU power. There are many systems of this class, from old file-sharing services [12], to the much-cited Chord [13] and Kademlia that is the basis of the popular BitTorrent [14], among several others [15]. We look into more detail to IPFS, as it is commonly used in the context of blockchain [3]. There are other options in this space but arguably not as popular as IPFS: Storj [5], FileCoin [6], Ethereum SWARM [16], and Metadisk [17].

IPFS is a P2P distributed file system that leverages some of the best previous techniques and algorithms [4]. IPFS is content-addressable, i.e., when a file is stored it returns a multihash (a self-describing hash) that can be used by anyone connected to the IPFS network to retrieve this file, i.e., that expresses the context of the file and is used to retrieve it. IPFS is organized as a stack of sub-protocols. Each of the layers is responsible for different functionalities within the system. The layers are the identities layer, network layer, routing layer, exchange layer, Merkle DAG + Files layer, Naming layer, and finally the Application layer. 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. Files are encrypted and their integrity is checked with cryptographic hashes. It is arguably less available than a central storage system because nodes can leave and join the network at will and the download speed depends strongly on the bandwidth of the nodes storing the data.

B. Cloud Storage Systems

Cloud 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 clients [18]. There are many commercial cloud storage services such as Amazon S3 [10], and Google Cloud Storage [11]. In this work, we are particularly interested in the security and availability aspects of these services. Therefore, next, we discuss DepSky [] and Unidrive [] two systems focused precisely on improving these aspects.

DepSky is a cloud storage system that improves the integrity, confidentiality, and availability of files stored on commercial clouds [19]. Files are 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 stored and metadata, and vendor lock-in. DepSky tackles these issues by exploiting replication and diversity to store the data on several clouds (loss of availability). Using 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.

UniDrive attempts to solve some of the challenges of DepSky [20]. The authors point out what they consider is a drawback of DepSky: that it requires a global clock synchronization mechanism among clients and the existence of an extra lock/release process in each write operation. Also, DepSky uses a metadata file for each file/directory whereas on UniDrive there is a single metadata file that captures all of the metadata. UniDrive 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 strip user data into smaller chunks, perform erasure coding to add redundancy, like in DepSky. Afterward, schedule the distribution of these chunks to the multi-cloud to meet reliability and security requirements.

C. Proof-of-Storage

For a P2P storage system that stores data on peers storage space, it is important to have some kind of cooperation incentive mechanism to prevent the system's collapse.

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 [21], [22]. These mechanisms are homomorphic in the sense that they produce integrity control structures that have the same structure as the data to be checked. 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 [23],

[24]. 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 [25]. There are two main cooperation schemes; those based on reputation and those based on remuneration.

Reputation mechanisms have three stages [15], [26]. (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 [6]–[8]. (1) First, the negotiation process in which two peers may have to negotiate the interaction terms. (2) Second, the cooperation decision, meaning that during the negotiation and based on its outcome, a peer will determine whether to cooperate or not. (3) Third, there is the cooperation evaluation, in this phase, the requesting peer has to evaluate the provided service. (4) 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.

D. Education Certificates Storage on the Blockchain

Serratino et al. presented an ecosystem in which education certificates can be verified through the Ethereum blockchain [2]. 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 a HEI. There is one HSC per HEI of the consortium.

There are two typical workflows. (1) A student graduates in a 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 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.

III. P2CSTORE

In this section, we present the design and implementation of the P2CSTORE system. We start by explaining the system model. We analyze once again the problem definition and the relevant properties and assumptions. Finally, we describe the relevant algorithms.

A. System Requirements

The system has some requirements that are important to mention: *Data Freshness*: When a client requests a file from the system by send the URL of the file, the system must return the most recent version of the stored file. *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. *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.

B. 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 [14] to find which nodes are storing some specific content. Each node will exchange information through the lookup of other nodes. Each node has a node ID and the Kademlia algorithm uses the node ID to locate values on the network.

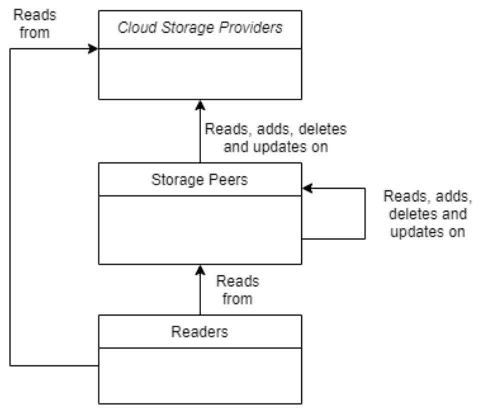
We assume the communication is reliable and secure. We do not present a specific solution for how to obtain this as there are several, e.g., using the TLS protocol [27] or the DTLS protocol [28].

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

C. 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.

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 1). *Cloud providers* are commercial public infrastructures that provide to their clients data storage and high levels of availability. *Storage peers* are 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 storage and due to that fact can also store content on the network/system,


Reads from
Cloud Storage Providers
Reads, adds, deletes and updates on
Storage Peers
Reads, adds, deletes and updates on
Reads from
Readers
Fig. 1: System Entities and their relationships

working as servers/storage providers as well. *Readers* are 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. 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.

D. 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 the next section.

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.

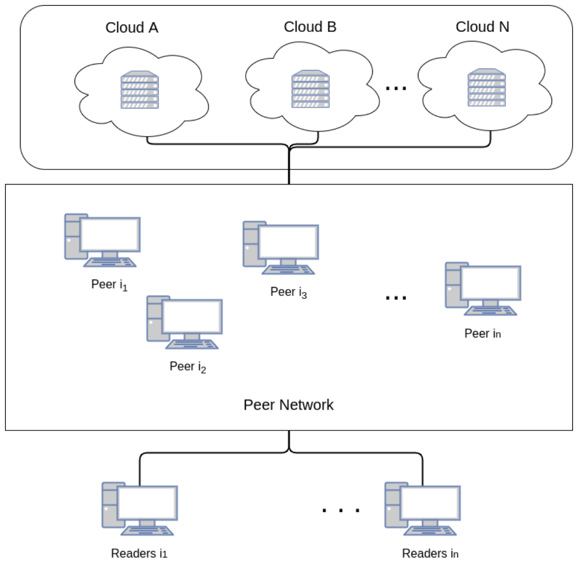
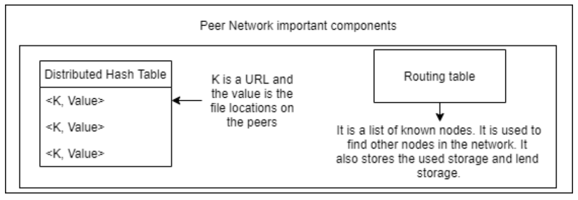


Fig. 2: P2CSTORE System Architecture Overview

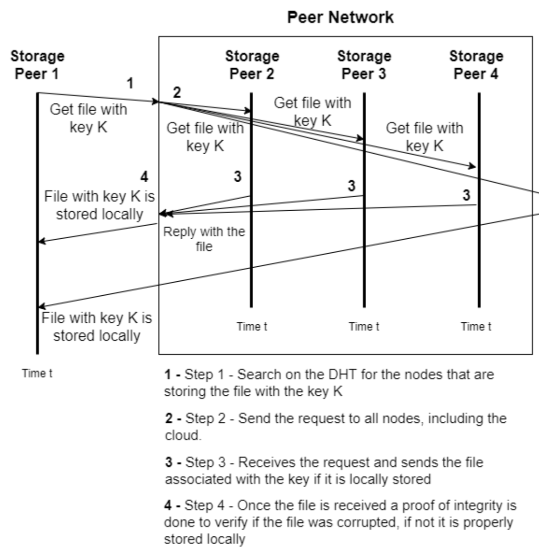

Peer Network important components
Distributed Hash Table
<K, Value>
<K, Value>
<K, Value>
K is a URL and the value is the file locations on the peers
Routing table
It is a list of known nodes. It is used to find other nodes in the network. It also stores the used storage and lend storage.
Fig. 3: P2CSTORE System Architecture Overview

E. Software Architecture

In Figure 2 and Figure 3 we can see two diagrams of the system architecture. In Figure 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 we have a more detailed image of the Peer Network, in particular its two major 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.

F. Basic Operations

In any storage system, some operations are straightforward. For instance, it must be possible to read data from the storage



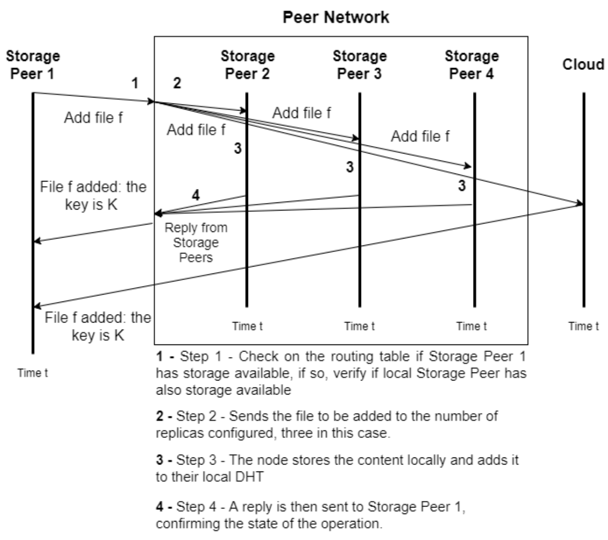
1 - Step 1 - Search on the DHT for the nodes that are storing the file with the key K

2 - Step 2 - Send the request to all nodes, including the cloud.

3 - Step 3 - Receives the request and sends the file associated with the key if it is locally stored

4 - Step 4 - Once the file is received a proof of integrity is done to verify if the file was corrupted, if not it is properly stored locally

Fig. 4: Get Operation Protocol Diagram



1 - Step 1 - Check on the routing table if Storage Peer 1 has storage available, if so, verify if local Storage Peer has also storage available

2 - Step 2 - Sends the file to be added to the number of replicas configured, three in this case.

3 - Step 3 - The node stores the content locally and adds it to their local DHT

4 - Step 4 - A reply is then sent to Storage Peer 1, confirming the state of the operation.

Fig. 5: Add Operation Protocol Diagram

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.

Get Operation: In Figure 4 we have a diagram of the protocol for the Get operation. The node Storage Peer 1 is attempting to get a file with a key K from the system, step 1. To be able to do that he searches on the DHT for the nodes that are storing the file with the key K, 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 upon file adding to the system. 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.

Add Operation: In Figure 5 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 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: In Figure 6 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 his 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

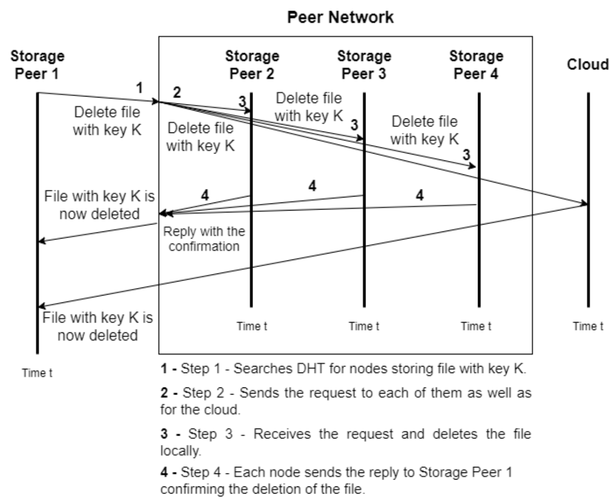

Peer Network
Storage Peer 1
Storage Peer 2
Storage Peer 3
Storage Peer 4
Cloud
1
2
3
Delete file with key K
Delete file with key K
Delete file with key K
Delete file with key K
4
File with key K is now deleted
Reply with the confirmation
File with key K is now deleted
Time t
1 - Step 1 - Searches DHT for nodes storing file with key K.
2 - Step 2 - Sends the request to each of them as well as for the cloud.
3 - Step 3 - Receives the request and deletes the file locally.
4 - Step 4 - Each node sends the reply to Storage Peer 1 confirming the deletion of the file.

Fig. 6: Delete Operation Protocol Diagram

the DHT. Upon receiving the request each of the Storage Peers deletes the file locally, step 3. Afterward, each node sends the reply to the 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 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.

***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. We do not describe the protocol for the update operation because it consists simply of a delete operation followed by an add operation.

G. 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.

H. 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).

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 π, 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 than V has a 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 than 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, the 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.

On the Algorithm 1 we can see the four functions that make 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 than 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 functions is the *handleProofOfStorageChallenge* (lines 1-11). This function receives the challenge sent by the *Verifier* and obtains the positions list, the nonce, and the file. Next, it will get the respective character on the file associated with the position

```
Function storageProofRequest (file):
    call function generateChallenge
    for each node that is storing the file do
        send challenge to node
        start timer
        get response from node
        if response time greater than time available then
            /* Assume node temporarily
               unavailable                              */
            increment node down counter by 1
            if node down counter equals $n$ then
                /* $n$ is configurable                  */
                call function handleFaultyNode
            end
            call function checkReplicationUpdateDHT
        end
        else if response is not valid then
            call function handleFaultyNode
            call function checkReplicationUpdateDHT
        end
        /* Otherwise everything ok, continue            */
    end

Function handleFaultyNode (nodeInfo):
    mark locally node as faulty      /* Local file stores
      faulty nodes */
    remove files of faulty node
    return

Function checkReplicationUpdateDHT (fileInfo):
    /* This works like a new add, removes the
       previous information on the DHT and adds
       the file to the system ignoring the
       faulty nodes                                     */
    remove file from the system
    add file to the system
    return

Function generateChallenge (fileInfo):
    create a list of bytes of size $n$
    while list of bytes size equals 0 do
        generate random(file size -1)
        /* random can be from 0 to file size            */
        if random number is an odd number then
            add i to list
        end
    end
    generate a random nonce
    generate the challenge with the list of bytes plus the nonce
    return
```

Algorithm 1: Verifier Functions – PoS Algorithm

```
Function handleProofOfStorageChallenge (challenge,
  fileInfo):
    get byte list from challenge
    get nonce from challenge
    get file from fileInfo
    for each byte i in list do
        get byte of position i of file
        convert byte to character add character to response array
    end
    challenge response equals response array plus nonce
    send the hash of the response to verifier
    return
```

Algorithm 2: Prover Function – PoS Algorithm

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*.

IV. Evaluation

A. Experimental Evaluation

The goal of our evaluation is to answer three main questions: Which is faster to use the P2P storage system or Cloud-based storage system? What are the costs of using multiple clouds to store data versus P2P systems or the Ethereum blockchain itself? What is the cost of the PoS mechanism?

1) Methodology: 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. For the clouds, we used S3 [10] and GCP [11] because they are the most common and also because they are included in the library JClouds [29].

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 [30] 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. Next, we will discuss the obtained results, draw our conclusions, and attempt to answer each of the above questions.

B. Operations Latency Evaluation

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.

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 10.

On the obtained graphics the Xs axis represents the results of the operations for each of the test scenarios of the operation. The Ys axis represents the time spent in seconds.

1) Add operation: As we can see in Figure 7, 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

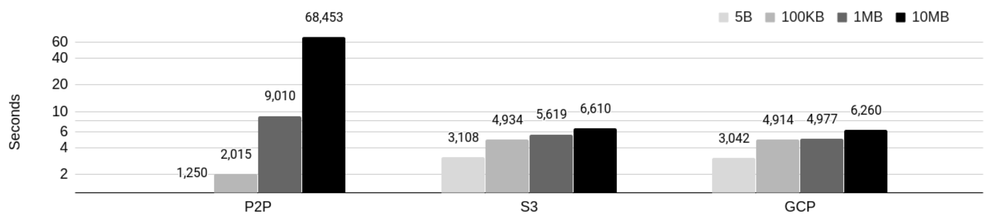


Fig. 7: Results for 100 add operations for different file sizes.

operation. This is the main reason that we are seeing such a huge difference when compared with the smaller sizes. 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.

2) Get operation: As we can see in Figure 8, 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.

3) Delete operation: As we can see in Figure 9 for larger files the time it takes to delete from the P2P and both clouds remain roughly the same. The increase is residual. 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* [31], and in fact, one can observe that the deletion time is similar to the insert time for small files (Figure 7).

4) PoS Algorithm: 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 10 the graphic is not on a logarithmic. This test was not done for the clouds because they do not execute any code.

As we can see in Figure 10 for larger files the time it takes to perform the PoS algorithm, and respective reply to the verifier, increases but not by much 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.

C. 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 [8], [32]. We can easily calculate the cost of storing a file with 1KB [33]. 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 [34]. We did an experiment to attempt to find out the cost of storing certain values in the S3 storage bucket, and we discovered that using the Price Calculator [35] 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. 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 [36]. Finally, the cost of having a P2P node machine connected has an associated electricity cost. This value depends on the country and region

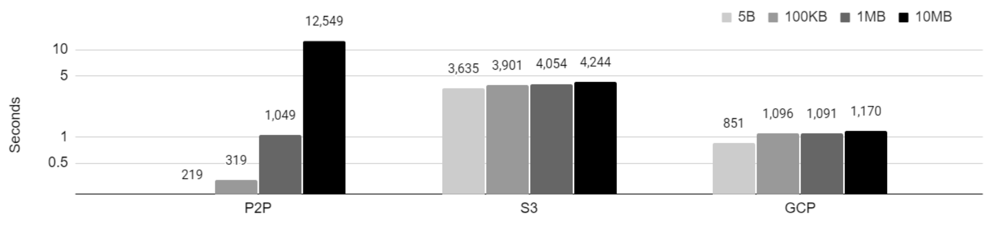


Fig. 8: Results for 100 get operations for different file sizes.

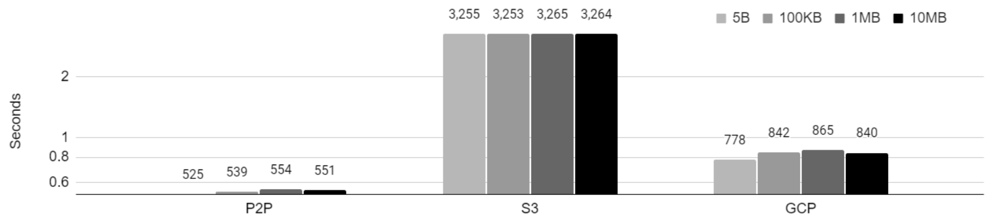


Fig. 9: Results for 100 delete operations for different file sizes.

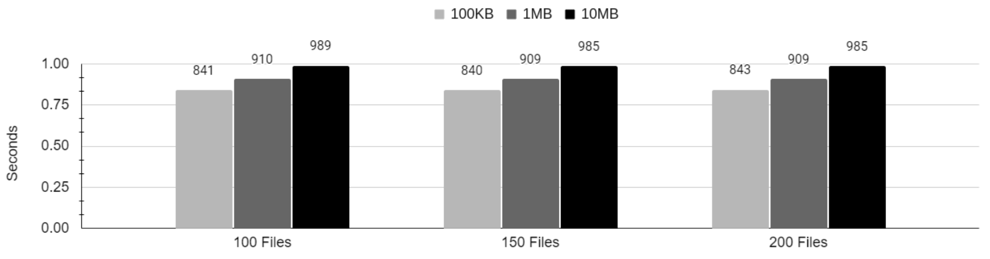


Fig. 10: Cost of the PoS for different file sizes and total number of files (configuration with P2P storage only).

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 it 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. In our system this is configurable, therefore for different applications, different values could apply better than others. Performing 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. 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 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.

D. Functional Evaluation - System Requirements

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.

As described in Section III 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 ensured by performing an integrity check upon getting the data from the system. When a new file is added 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 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: Our system is prepared to have both cloud storage providers as well as a configurable number of peers. These values are configurable by each user to increase availability. Therefore the availability is related with the number of clouds + peers configured by each user.

V. Conclusion

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.

The proposed solution presents an architecture of a P2P and Cloud Storage System that uses the best of both. It uses both to improve the availability of the system as well as the cost of usage, giving to the developer/administrator the flexibility to manage the trade-offs of using P2P and Cloud Storage Systems.

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, a description of the Implementation was presented. Afterward, the evaluation of the system was described and the obtained results.

References

[1] M. E. Peck, "Blockchains: How they work and why they'll change the world," *IEEE Spectrum*, vol. 54, no. 10, pp. 26–35, 2017.

[2] D. Serranito, A. Vasconcelos, S. Guerreiro, and M. Correia, "Blockchain ecosystem for verifiable qualifications," in *Proceedings of the 2nd IEEE Conference on Blockchain Research & Applications for Innovative Networks and Services*, September 2020.

[3] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: building smart contracts and dApps*. O'Reilly Media, 2018.

[4] J. Benet, "IPFS-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[5] S. Wilkinson, T. Boshevski, J. Brandoff, and V. Buterin, "Storj a peer-to-peer cloud storage network," 2014.

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

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[8] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[9] 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," vol. 53, no. 4, pp. 50–58, Apr. 2010.

[10] "Amazon S3," https://aws.amazon.com/s3/.

[11] "Google Cloud Storage," https://cloud.google.com/storage/.

[12] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "Measurement study of peer-to-peer file sharing systems," in *Multimedia Computing and Networking*, vol. 4673, 2001, pp. 156–170.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM SIGCOMM*, 2001, pp. 149–160.

[14] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*, 2002, pp. 53–65.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, R. G. 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*, Nov. 2000.

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

[17] S. Wilkinson, J. Lowry, and T. Boshevski, "Metadisk a blockchain-based decentralized file storage application," *Tech. Rep.*, 2014.

[18] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. Pearson Education, 2005.

[19] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," *ACM Transactions on Storage (TOS)*, vol. 9, no. 4, p. 12, 2013.

[20] H. Tang, F. Liu, G. Shen, Y. Jin, and C. Guo, "Unidrive: Synergize multiple consumer cloud storage services," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 137–148.

[21] B. Wang, B. Li, and H. Li, "Panda: public auditing for shared data with efficient user revocation in the cloud," *IEEE Transactions on Services Computing*, vol. 8, no. 1, pp. 92–106, 2015.

[22] N. T. F. de Carvalho, "A practical validation of homomorphic message authentication schemes," Master's thesis, University of Minho, 2014.

[23] J. Benet, D. Dalrymple, and N. Greco, "Proof of replication," *Protocol Labs, July*, vol. 27, p. 20, 2017.

[24] B. Fisch, "Tight proofs of space and replication," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2019, pp. 324–348.

[25] L. Strigini *et al.*, "Resilience-building technologies: State of knowledge – ReSIST NoE deliverable D12," 2007.

[26] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings 18th ACM Symposium on Operating Systems Principles*, 2001, pp. 202–215.

[27] T. Dierks and C. Allen, "The TLS protocol version 1.0 (RFC 2246)," IETF Request For Comments, Jan. 1999.

[28] E. Rescorla and N. Modadugu, "Datagram transport layer security version 1.2 (RFC 6347)," IETF Request For Comments, 2012.

[29] "Apache jclouds," https://jclouds.apache.org/, accessed: 2020-09-19.

[30] "Planetlab europe," https://www.planet-lab.eu/, accessed: 2020-09-19.

[31] "Amazon s3 - developer guide," https://docs.aws.amazon.com/AmazonS3/latest/dev/delete-or-empty-bucket.html, accessed: 2020-09-19.

[32] "Eth gas station," https://ethgasstation.info/, accessed: 2020-09-13.

[33] "Eth gas station calculator," https://ethgasstation.info/calculatorTxV.php, accessed: 2020-09-13.

[34] "S3 pricing," https://aws.amazon.com/pt/s3/pricing/, accessed: 2020-09-13.

[35] "Aws s3 price calculator," https://calculator.aws/#/createCalculator, accessed: 2020-12-03.

[36] "GCP pricing," https://cloud.google.com/storage/pricing, accessed: 2020-09-13.