P2CSTORE: P2P and Cloud File Storage for Blockchain Applications

Marcelo Silva Miguel Matos Miguel Correia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{marcelo.filipe.regra.da.silva, miguel.marques.matos, miguel.p.correia}@tecnico.ulisboa.pt

***Abstract*—We live in an era where storage systems are of major importance. In this work, we focus on storing files for use in blockchain applications. A blockchain is a distributed, replicated system, that contains a ledger of operations and executes programs known as smart contracts. Although a blockchain may be considered a data storage system, public blockchains like Ethereum charge the users per each byte stored making them expensive to store large files. Therefore, applications based on blockchains often use an external file storage system, usually peer-to-peer (P2P). 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, depending on aspects such as the trust they place in peers and clouds. 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.**

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], [2]. A blockchain can enforce the integrity and availability 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 security properties like integrity, authenticity, and non-repudiation. The fact that data is replicated in many nodes additionally ensures availability.

As an example, project QualiChain (https://qualichain-project.eu/) is developing a blockchain-based system to enforce the authenticity of university certificates, [3], [4], [5], [6]. 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* [7], 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: P2P and centralized storage. DApps often use as external file storage system a *peer-to-peer (P2P) file system* [8], [9], [10], [11], [12], [13], as many blockchains are also P2P systems [14], [15]. In a P2P system, there is no centralized server, instead some nodes share resources to store files. If a node wants to get a particular file, it can request it from the network. A file system is said to provide availability if it can ensure that a certain file will be ready to be requested at any given time. This is hard to guarantee in P2P systems since nodes can leave and join the network at will, making the files they store available and unavailable as they join and leave respectively. These systems are typically based on volunteer nodes and therefore can be used free of charge.

An alternative is a 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 [16], this architecture is provided by many *cloud storage services* [17], [18], [19], [20], that ensure high-availability but charge for bytes stored and downloaded.

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 administrators to select among a wide range of configurations with different trust, cost, and availability trade-offs. To guarantee high availability and integrity of the files we will leverage replication techniques and a proof-of-storage mechanism. 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.

II. THE P2CSTORE SYSTEM

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

A. 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 the content of the files it stores.

Nodes communicate through the Kademlia DHT [21]. Each node will exchange information through the lookup of other nodes. Each node as a node ID and the Kademlia algorithm uses the node ID to locate values on the network.

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.

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 [22] or the DTLS protocol [23].

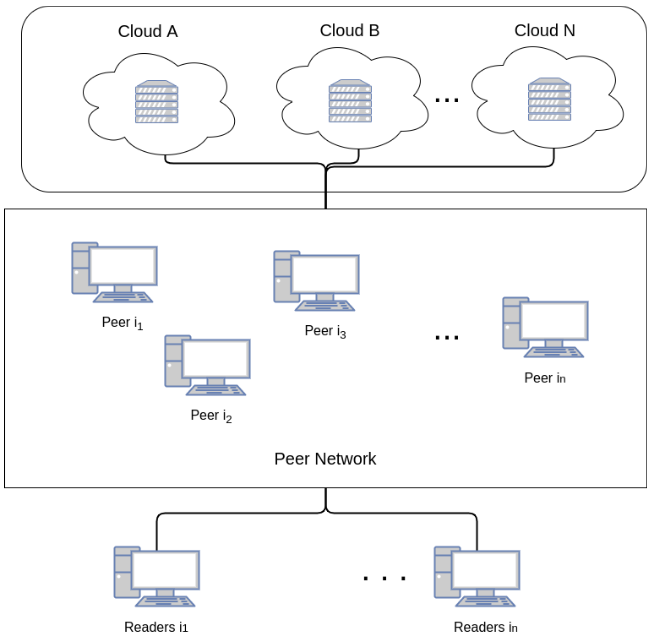

Fig. 1. P2CSTORE System Overview.

B. P2Cstore Overview

The problem that we solve, as described above, is to create a storage system to store generic files for DApps, e.g., education certificates, on the Ethereum blockchain.

To solve this problem we created P2CSTORE which consists of a peer-to-peer (P2P) network for nodes/clients to interact with, with access to cloud storage services. The clients decide how they want their data to be stored: on peers, on clouds, or both. Next, we describe in more detail each component of the P2CSTORE architecture.

The P2CSTORE system has two types of participants. First there are *storage peers* (or nodes). These participants provide storage space and act as both readers and writers of files. Second there are *readers*. These are not full participants, but simply allowed to read files. (not write or delete). This second type of participants was included to allow 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, e.g., HR 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 on anything. 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.

An overview of the P2CSTORE is presented in Figure 1. As described above, it is composed of peers of the peer network that also provide storage, by readers, and by cloud storage providers. The readers do not participate in the peer network, they can only read content shared by storage peers. Therefore we have the cloud storage peer layer, the peer network layer, and finally the reader's layer.

Each peer can leave and rejoin at will and the system will still function as long as other nodes are online providing the content hosted by the node that left. If a node changes the content of data without permission the system will catch this and consider the content corrupt and reject it. If a certain node fails to prove that it is storing the file the system will also catch this fault.

P2CSTORE is based on a fairness condition: each peer can only use the same amount of P2P storage that it gives to the system, although it can use more storage in the cloud with the associated 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 (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 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 writer node finds a destiny node to store the file it (the writer node) will verify on the routing table if the destiny node has available space for that content. To clarify we called this node writer to distinguish it from the destiny node, in reality, this is an ordinary peer node that is attempting to perform an add operation. The second is a Proof-of-Storage algorithm, explained in the next section.

C. *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 Algorithm 1 we can see the four functions of the PoS algorithm *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. Afterwards, 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.

```
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
```

Algorithm 1: Verifier Functions – PoS Algorithm

On Algorithm 2 we can see the PoS algorithm *Prover* side. There is a single function, *handleProofOfStorageChallenge* (lines 1-11). This function receives the challenge sent by the

```
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

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 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.

III. Conclusion

In this paper, we presented P2CStore, a P2P and Cloud storage system for Blockchain applications. Our approach includes a P2P network in which nodes locate each other through lookup calls using the Kademlia DHT. We also included two cloud storage providers to our solution, namely AWS S3 buckets and GCP Storage buckets. To prevent free-riding attacks we designed a PoS algorithm in which nodes have to prove that they are storing the files. We also created a mechanism that only allows nodes to use the storage space that they provide to the system. This way we prevent the collapse of P2CStore by incentivizing cooperation and sharing of resources. This way nodes will opt for being correct and work with the system instead of working against it. This is important to ensure the system's sustainability.

The design and implementation of P2CStore can be extended and improved in several different ways. It could be done a smart contract on Ethereum that would be responsible to reference the hash of Ethereum transactions to the respective node on P2CStore. 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 for the file location on our storage system. Another natural improvement is to make the processing of the operations concurrent on the client-side to reduce latency.

Acknowledgements This research was supported by the European Commission under grant agreement number 822404 (QualiChain) and by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID).

References

[1] S. Underwood, "Blockchain beyond Bitcoin," *Communications of the ACM*, vol. 59, no. 11, pp. 15–17, 2016.

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

[3] 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.

[4] A. Mikroyannidis, A. Third, and J. Domingue, "Decentralising online education using blockchain technology," in *The Online, Open and Flexible Higher Education Conference: Blended and online education within European university networks*, Oct. 2019.

[5] 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.

[6] I. R. Keck, M.-E. Vidal, and L. 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.

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

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

[9] D. Vorick and L. Champine, "Sia: Simple decentralized storage," *Nebulous Inc*, 2014.

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

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

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

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

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

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

[16] 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*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

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

[18] 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*, 2011, pp. 143–157.

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

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

[21] 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.

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

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