University of Stavanger

BOHUA JIA
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Proving Redundancy In Decentralised Storage Networks

Master's Thesis - Computer Science - June 2022

I, **Bohua Jia**, declare that this thesis titled, “Proving Redundancy In Decentralised Storage Networks” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master’s degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

“Programming is a nice break from thinking.”

– Leslie Lamport

Abstract

Proof of Storage (PoS) is a scheme that proves the data is stored honestly. PoS is the general term for a collection of related proofs. Such as proof of retrievability and proof of data possession. In recent years, various PoS variants have been proposed, each with advantages. However, there is no protocol for auditing between peers without involving third-party auditors. This thesis proposes a protocol which allows auditing between peers without third-part involved. The protocol is fully implemented with Go coding language and deployed on the Swarm test network. The designed protocol is mainly based on proof of retrievability, proof of data possession and proof of replication. The proposed protocol uses a challenge-response protocol to send messages between nodes in a decentralised file storage system. Experiments show that our idea is feasible. After experimenting with different test nodes, the results show that the efficiency of our proposed protocol is related to the number of redundant data chunks owned by the challenger and prover in the same network. If either challenger or prover holds a relatively small amount of data chunks, our proposed protocol will have better performance.

Acknowledgements

I would like to thank my supervisors Hein Meling and Racin Nygaard for their enthusiasm and help with writing this thesis. Their comments and suggestions are always inspiring at the weekly meeting. I also want to thank Rodrigo Saramago for helping with properly making the citation on the Swarm documentation website and deploying bee test nodes using scripts.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

With the continuous development of the internet, the concept of digital life is gradually accepted by people, and people progressively begin to convert essential documents in study and work from paper documents to digital documents and store them locally. As a result, the demand for storing data increased. With the popularity of social media sites and software, recording and sharing life has become a fashion lifestyle for contemporary young people. This further increases the need for data storage. With the improvement of the camera quality of digital products, the resolution of photos and videos has risen from 1080 pixels to 4000 pixels, and at one time, the size of digital files generated by taking them is also increasing rapidly, and people need larger hard disk storage space to store these files.

With the explosive growth of the demand for hard disk storage, although the storage capacity of the computer's local hard disk is also increasing, it is not enough to meet the demand. Data outsourcing has become a convenient solution [1] when the demand for data storage is far greater than the actual local storage capacity. Users can achieve their data stored in the cloud anytime and anywhere by outsourcing data, reducing the local hard disk storage burden. Moreover, it can prevent the potential risk of local storage breakdown.

More and more people are getting familiar with the decentralised storage system

in recent years. An obvious benefit of deploying cloud storage services on a decentralised file system is that the cloud storage service will not enter a paralysed state if one node crashes or some node crashes. Users do not need to wait for nodes to restart and recover before using the service. In other words, unless an extensive range of nodes in the user's location is all offline, the user has a great chance will not be affected by the downtime of the nodes and can successfully access the files the user stored. Storing files in a decentralised file storage system is better than traditional centralised cloud storage services. Although there is almost no chance that users cannot obtain stored files in a decentralised network storage system, users with corresponding needs can ensure the availability of files by making replicas of the file. In other words, users can make redundancy to ensure the availability of files in order to overcome the problem.

Nevertheless, there are some problems [2]. How can users ensure that the redundancy data is actually saved, not just declared to be saved? How can users ensure their data integrity? The easiest way is to employ a third-party auditor. However, this solution has shortcomings. First of all, the user needs to trust the third-party audit unconditionally. That also raises concerns among users regarding data security and the accuracy of the third-party auditor's report. Furthermore, users and network storage service providers need to bear the cost of hiring third-party auditors. Therefore, we need a protocol to regularly perform automatic auditing between all nodes in the same decentralised storage system without hiring any third-party auditors to guarantee the integrity and availability of data. This paper will propose a protocol based on proof of storage (PoS) to achieve self-audit between nodes in a decentralised file system.

PoS are cryptographic techniques [3] that allow clients to check the integrity of data stored remotely efficiently. The client delivers an encoded version of its data to the storage service provider while maintaining a little piece of state in local storage to use a PoS. The client may then use a highly efficient challenge-response protocol with the storage service provider to verify the integrity of its data at any point in time. The challenge-response protocol involves two characters, namely challenger and prover. With the standard PoS scheme, the challenger sends a challenge to multiple provers and receives proofs back from the provers. There are many variants of the PoS scheme. There are differences in the data structure, using various algorithms and having additional roles involved. And each of them

has advantages and disadvantages.

1.2 Objectives

The goals of the study is given by the following list.

- Realising self-auditing between different peers in a decentralised file storage system.
- To be able to verify the message's authentication.
- To be able to identify the data chunk integrity.
- To be able to verify if the target chunk has been modified.
- To be able to find adversary misbehaviour.
- Reducing the cost of bandwidth when communicating with other peers.
- Keeping user data privacy as much as possible.
- Keeping the computational complexity as low as possible.
- Deploying the proposed protocol on the Swarm test network.

1.3 Approach and Contributions

Some subsections build up our entire approach. Firstly, we introduce the assumptions that our proposed protocol follows. After applying a hash function to the data, we assume that the hash function will only produce the same output if the input data is identical. There has no way to revert hashed value to get the original data. We assume recipients will finally receive the messages sent between peers in a decentralised file system. Moreover, we assume the prover will not delete any chunk during the auditing period. Secondly, we give an overview of the proposed protocol. Then, we describe the participants we used in our protocol. Although our protocol is designed for peers in a decentralised storage system, the same peer will play various roles in different situations. Next, we explain the tasks that different roles need to accomplish during auditing. There are two roles, challenger and prover. The challenger will create a challenge based on its locally stored data chunks. The prover iterates its locally stored chunks and tries to find

the chunks mentioned by the received challenge. The prover uses the result of found chunks to create proof and send it back to the challenger. The challenger starts validation when the proof is received. Finally, we present the algorithms we use to implement our protocol. The algorithms include creating challenges, proving data possession, verifying proof and some related functions.

The contributions we made to this thesis are given in the following list.

- The code of the proposed protocol has been implemented in the Go language.
- The protocol is implemented in a general case.
- The implementation can deploy on any decentralised storage system by adjusting the related functions.
- Generating unit tests for each part of the implementation.
- Generating unit tests for communication between Swarm test nodes.
- Generating experimental results into the system log file and converting the log into graphs.
- Analysing the experimental results.
- The idea of our proposed protocol has a positive effect on the financial field of a decentralised storage system.

1.4 Outline

The following list organizes the rest chapters of this thesis.

- Chapter 2 introduces the previous related research and the technology used in this thesis.
- Chapter 3 detailed the approach of our proposed protocol.
- Chapter 4 covers the test environment setup and gives the analysis based on the outcomes.
- Chapter 5 describes the insight from the outcome and introduces a possible future work.

- Chapter 6 presents the conclusion of this thesis.

Chapter 2

Related Work

This chapter will present the preliminaries of explanation for technical nouns, related schemes and the basic knowledge of Swarm.

2.1 Preliminaries

This section will give brief descriptions of some technical nouns, and we will mention them in the rest of this thesis.

2.1.1 Asymmetric Encryption

Asymmetric encryption is modern cryptography, and it aims to guarantee that sensitive information is secured. Asymmetric encryption is also known as public-key cryptography. Asymmetric encryption uses pairs of keys. Every pair include a public key and a private key [4]. For example, if user *A* wants to communicate with user *B* and encrypts their communication. User *A* will first share the public key with user *B*. Then user *B* uses the received public key to encrypt the message and send the ciphertext to user *A*. Finally, when user *A* receives the ciphertext from user *B*, user *A* will use the private key to decrypt the received message in order to get the plain text. Figure 2.1 illustrates a brief overview of asymmetric encryption message flow.

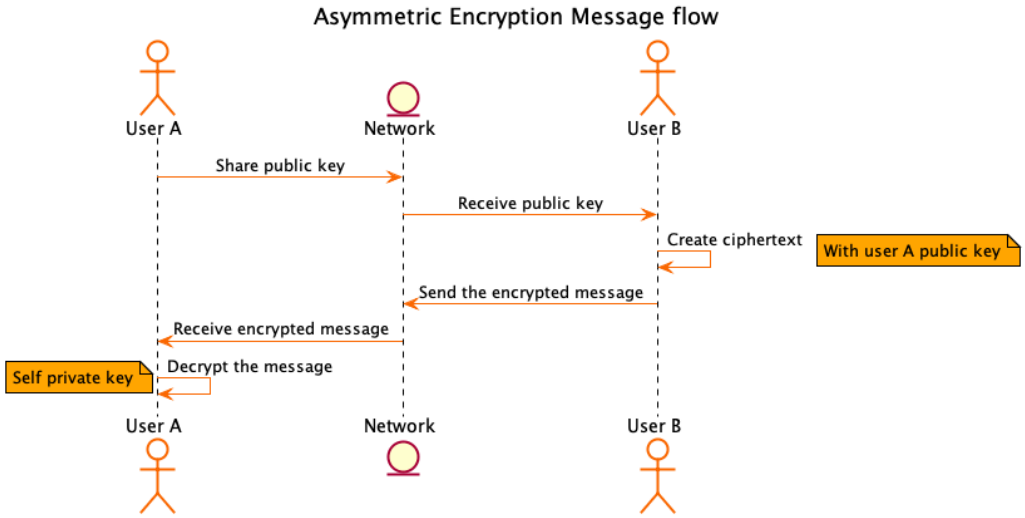

Figure 2.1: Asymmetric encryption message flow

Digital Signature

In addition to encrypted communication, asymmetric encryption algorithms can also perform digital signatures. A digital signature is a mathematical approach for proving data authenticity and integrity. Non-repudiation and uniqueness are essential characteristics of a digital signature. Figure 2.2 shows an example of a simple case where a digital signature is sent and identified without regard to encrypted data. Firstly, user *A* applies a hash function on plain text. Secondly, user *A* uses the private key with hashed data to create a digital signature. Furthermore, user *A* sends the plain text, digital signature and public key to user *B*. User *B* will apply the same hash function as user *A* used to the received plain text. Finally, user *B* uses calculated hashed data with received user *A* public key to validate the digital signature.

2.1.2 Gas

Gas is the unit of computing how many computational steps of code execution a transaction can use [5]. Gas price refers to the fee or price needed to pay to finish a transaction on blockchain.

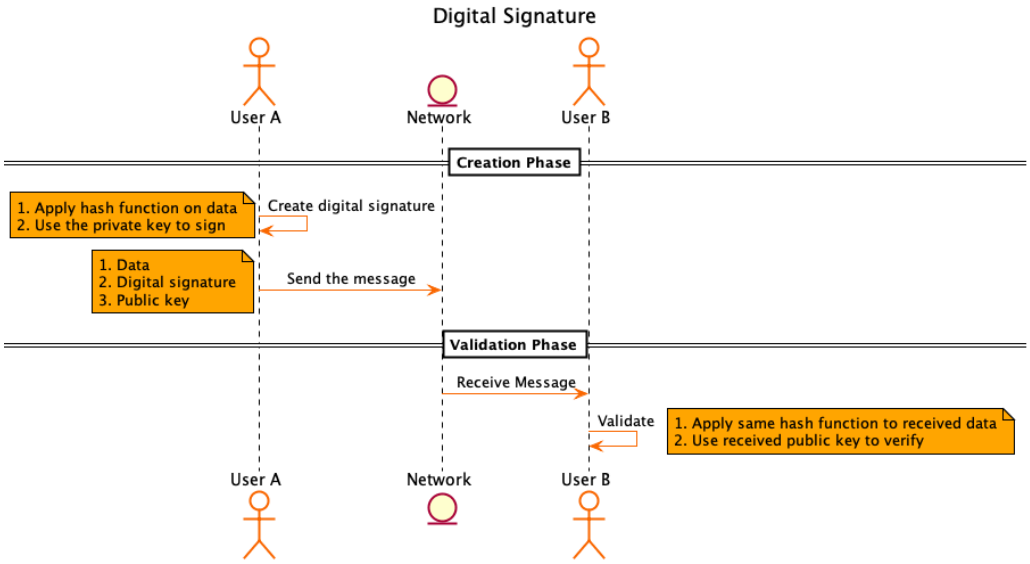

Figure 2.2: Create and validate digital signature message flow

2.1.3 Kademlia

The node relies on either the target ID or the routing table that the present node maintains to find the target in a P2P network. However, the nodes located on a P2P network are not stable in some cases. Sometimes, a node may go offline. Sometimes, a new node joins the network. A large-scale P2P network needs to build the mapping between the object and the network nodes, and when nodes change dynamically, objects are guaranteed to be accessible. Kademlia is a P2P distributed hash table that allows locating nodes near a particular ID and dynamically updating when nodes' status changes [6].

2.1.4 Public Verification

When users outsource their data to the cloud storage service provider, they lose some control of their data. So, they are always concerned about data integrity. Although the cloud storage services can provide a data integrity check, the report is unconvincing. Since the services provider may always produce a satisfying integrity report for keeping their good reputation even if part of the data is corrupted or missing [7], [8], [9] Since the verification may involve many calculations, users may not be able to afford it, so a public verification scheme has been

proposed to reduce the users' computational burden and prevent the server from cheating. The aim is to regularly hire an external, independent auditor to check the data integrity on behalf of users periodically. [10], [11]– [12] [13] [14] [15] [16].

2.1.5 xDAI

xDAI is a cryptocurrency pegged to the US Dollar [17]. The gas fee on xDAI is about $0.000021 per transaction. In addition, compared to Ethereum, xDAI is faster in performing a transaction. For example, a typical Ethereum transaction will take three to five minutes to finish a transaction. However, on the xDAI blockchain, the transaction will occur within five seconds to finish the transaction. These two advantages make the xDAI an ideal cryptocurrency for P2P transfers.

2.2 Related Schemes

This section will present the related schemes proposed by other researchers. Our proposed protocol is inspired by the fusion of some of the ideas from these schemes.

2.2.1 Proof of Storage (PoS)

PoS are cryptographic protocols that allow clients to quickly validate data integrity stored on a remote server [18]. Consider the following example, a user wants free local data storage and decides to outsource the data. After uploading the data to the cloud storage service provider, the user needs to ensure the cloud server stores the data honestly. However, the user may be unable to audit the cloud server frequently due to computational or bandwidth limitations. So, we need a third party that the user trusts to audit the server. Then the user will share the self public key and relate metadata from the uploaded file to the auditor. In this case, the cloud storage service provider's role as prover P and third-party auditor as verifier V. Common in PoS schemes is that V creates challenges based on data metadata, calculates the corresponding answer, and sends the challenge to the P. Then, the V will wait for P to produce the response for a reasonable time. On the P side, it needs to calculate the response based on the data that it stored previously and send it back to V. Suppose V received proof that does not match the answer that V calculated earlier or does not receive any response during the

waiting period. In that case, it marks *P* failure and alerts the user. The message flow is illustrated in Figure 2.3.

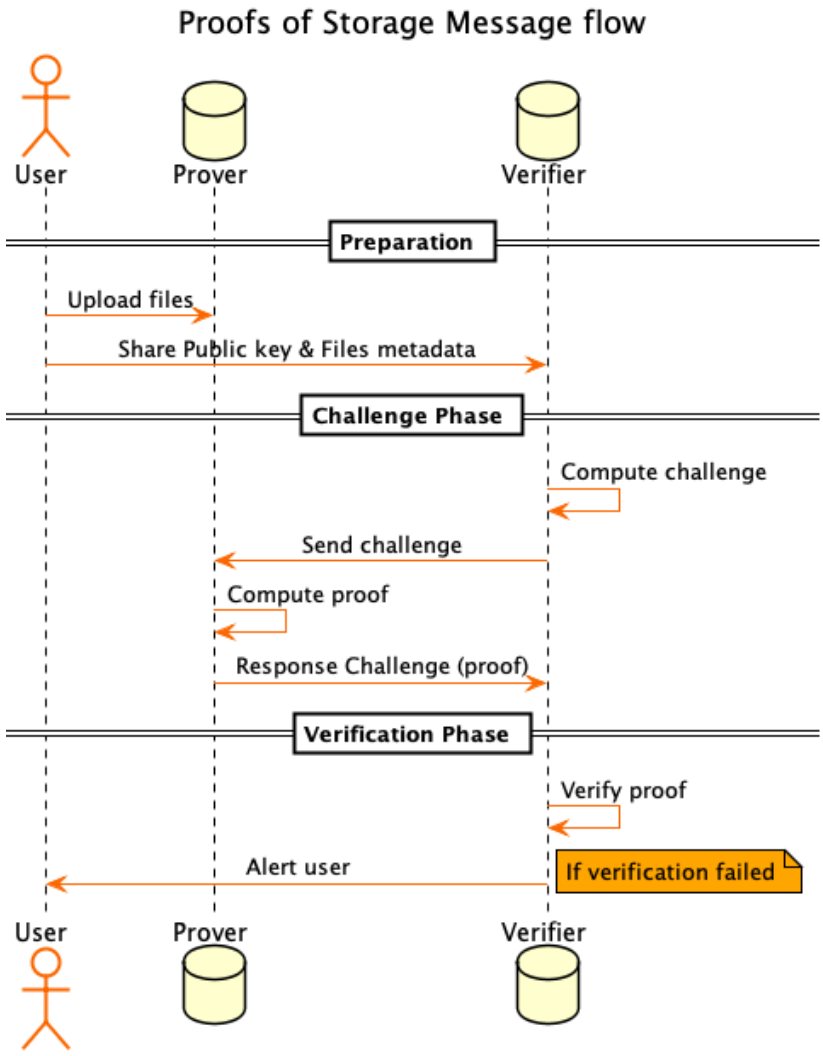

Figure 2.3: Proofs of Storage message flow

2.2.2 Proof of Retrievability (PoRet)

In a decentralised storage network, users store their data without recognising — or needing to know — where their data are stored on which devices or geographical regions with anonymous operators. Users need the scheme to ensure

they can retrieve their outsourced data back. The PoRet scheme had been proposed [14] [19]. It proves that the users' files can be completely recovered if the P is honest. An example, *P* provides *V* with evidence that the target file is integrated and can be fully retrieved. And *V* can verify the received proof. Since the data is split into chunks and using the hash function encrypts the chunks. So, even if a bit of a chunk is different, the result will be far from the correct answer.

2.2.3 Proof of Data Possession (PDP)

The *V* needs to verify the outsourced file's integrity on untrusted *P* frequently. The auditing action will cost most of the bandwidth to download the file if the outcome is only possible after retrieving the file. *V* needs a more efficient scheme to audit. The PDP has been proposed [9] [20]. PDP can verify the file's integrity without retrieving it. *V* only selects random chunks from the file to generate a challenge and sends it to *P*. Therefore, I/O costs are considerably reduced. Between *V* and *P* only send a constant small amount of data to each-other in the challenge and response protocol. Thus, it minimises network communication complexity. The PDP scheme is widely used in distributed storage systems for large files.

2.2.4 Comparison between PoRet & PDP

Table 2.1 indicates the comparison between the Proof of Retrievability and Proof of Data Possession.

Table 2.1: Performance comparison [21]

Scheme	Computation (Data Pre-process)		Communication bits		Storage Overhead (Server)	Computation (Verifier)			Computation (Prover)		
	exp.	mul.	Challenge	Response		exp.	mul.	pair.	exp.	mul.	pair.
PoRet	$\frac{\lvert F\rvert}{\lambda} + \frac{\lvert F\rvert}{m\lambda}$	$\frac{\lvert F\rvert}{\lambda}$	$\ell\lambda + \ell\log(\frac{\lvert F\rvert}{m\lambda})$	$(m+1)\lambda$	$\frac{\lvert F\rvert}{m}$	$\ell + m$	$\ell + m$	2	ℓ	$\ell + m$	0
PDP	$2\frac{\lvert F\rvert}{m\lambda}$	$\frac{\lvert F\rvert}{m\lambda}$	$\log \ell + 2\kappa$	2λ	$\frac{\lvert F\rvert}{m}$	ℓ	ℓ	0	ℓ	2ℓ	0

2.2.5 Proof of Replication (PoRep)

In some cases, the client demands to make replications of their outsourced file to avoid the file being unavailable for various reasons. For example, the stored

platform is offline. So, PoRep has been proposed [22]. PoRep is a novel type of PoS, demonstrating that the file has been duplicated to its unique physical storage. The *V* can ensure that the *P* is not replicating multiple copies of the file into the same storage area by enforcing distinct physical copies. In other words, *P* cannot pretend to store a file when it does not. Even though the PoRep has not been realised yet, the scheme aims to prevent p cheats. Mainly for preventing the Sybil attacks, outsourcing attacks and generation attacks. The following Figure 2.4 illustrates the message flow.

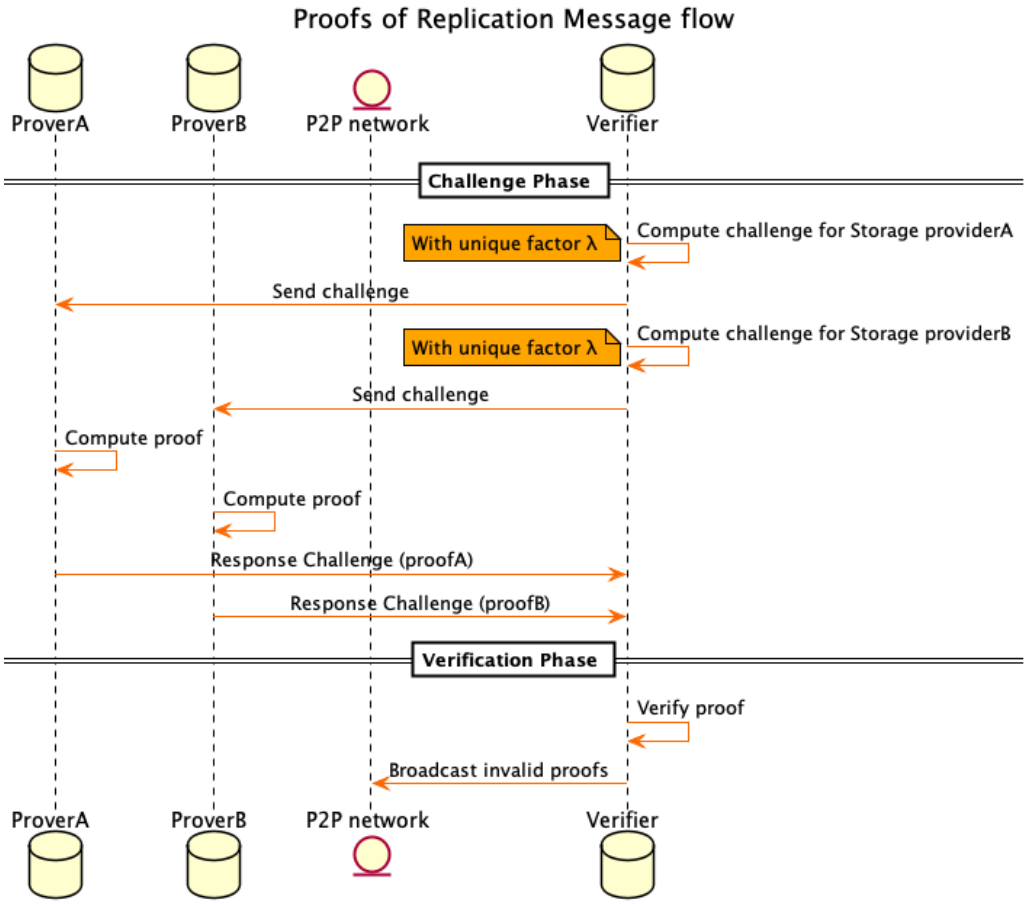

Figure 2.4: Proofs of Replication message flow

2.2.6 Proof of Space (PoSpace)

PoSpace is an alternative concept for proof of work, where a service requestor must dedicate a significant amount of disk space instead of computation. The PoSpace between V and P has two phases: the initialisation and proof execution phases. P is designed to store the file after the initialisation step, whereas V retains the file's metadata. V can start a proof execution phase later, and at the end, V outputs reject or accept. V is approvingly efficient in both stages. However, P is only efficient if the data is stored and accessible in the execution phase [23].

2.3 The Swarm

This section will describe the Swarm in general with some key components we will use in the following chapter.

The Swarm is a peer-to-peer (P2P) network of nodes cooperating to offer decentralised storage and communication [24]. The built-in incentive structure enforced by smart contracts on the Ethereum blockchain and fueled by the BZZ token makes this system economically self-sustaining. Swarm aims to build a global computer that can act as an operating system and deployment environment for decentralised applications by extending the blockchain with peer-to-peer storage and communication.

2.3.1 Swarm address

The address consists of a peer's physical address, topology address and signature, represented as a byte array [25].

2.3.2 Chunk size

When users upload their data into Swarm, each file is split into four-kilobyte chunks and assigned to a unique swarm address [25].

Chapter 3

Approach

This chapter introduces an audit protocol based on blockchain technology. Inspired by PoS based on a decentralised system, we design and implement a protocol for mutual audit between cloud storage service providers without third-party participation. Although the protocol deploys on the Swarm, its design prototype applies to any blockchain platform. We firstly present the assumptions we used during the project design. Secondly, we describe an overview of the structure of the protocol. Then we introduce the participants in our protocol and their roles. Finally, we detail the different protocol modules and the helper functions.

Our proposed protocol is based on PoS, PoRet, PDP and PoRep schemes because we made a combination of those related principles to construct our protocol. Although, our protocol only involved two participants, namely challenger and prover. They remain the similar functionalities of the V and P, respectively. For a comprehensive introduction to our proposed protocol, we decided to present it by choosing one network storage service provider as a challenger and giving a detailed message flow between the challenger and different provers.

3.1 Assumptions

This section will introduce the assumptions we used during the design. Assumptions involve different levels, including code-level, communication-level and design-level.

3.1.1 Hash function

A hash function can take the arbitrary length of the input value and map it to a fixed size of the result [26]. The identical input value with the same hash function will always generate the identical output. Anti-collision and anti-tampering ability are essential characteristics of a hash function. The anti-collision property ensures that it is almost impossible for two different pieces of data to have the same hash value. In other words, finding two different pieces of data with the same hash value is hard. The tamper-proof ability ensures that even if only one bit in the data is changed, the hash value output by the hash function will substantially change. A hash function is a one-way function, which means it is almost impossible to invert the hash function to retrieve the original data.

The formula 3.1 denotes applying a hash function on an arbitrary length of input value x and getting the hashed value output y.

$$y = H(x) \tag{3.1}$$

We assume that:

- Given identical input values will always output the same result.
- Applying a reversed hash function cannot retrieve the original input x from known y.

The formula 3.2 indicates applying a hash function to the proof components and obtaining the corresponding hashed value.

$$Proof = H(K|Id|chunk) \tag{3.2}$$

We assume that a new *Proof* cannot be generated by only changing the *Id* without knowing *K* and *chunk*.

3.1.2 Network communication

The ability to receive messages

We assume that during the interaction between different suppliers, as long as the challenger successfully creates the challenge, other suppliers will successfully re-

ceive the challenge. Likewise, responses from other providers will also be received by the challenger. With this assumption, our design no longer considers the troubles caused by response time or P2P network connection errors, which means we have removed all factors if time needs to be considered. Although a related article proposes construction that does not require time assumptions [27], we ultimately did not adopt this approach because its implementation was impractical.

No collaboration

We assume no collaboration between the various network storage service providers. Therefore, when a supplier is audited, it will not get help from other suppliers. In other words, the supplier cannot download the corresponding chunks from different suppliers to pretend that the supplier actually stores the chunks and successfully respond to the challenge by creating valid proof.

3.1.3 Chunks status

We assume that the peer will not remove any chunk during the auditing. Since a prover computes a proof based on challenger chunks' addresses, if one or some of the chunks has been removed on the challenger side, it may cause a potential failure when the challenger tries to validate the proof.

3.2 Default hash function

This section will introduce some of the fashion hash functions. Then we make the comparison between mentioned hash functions. Finally, present the default hash function we chose and used during the implementation.

3.2.1 MD5

Md5 is a cryptographic hash function that can generate a 128-bit hash value for an arbitrary input length [28]. It is used to ensure complete and consistent information transmission. The typical application of md5 is to generate a message digest for a piece of information to prevent tampering.

3.2.2 Secure Hash Algorithms

The National Institute of Standards and Technology (NIST) published the Secure Hash Algorithm (SHA) as a series of cryptographic hash functions [29]. SHA 2 is a branch of the SHA family, of which SHA256 and SHA512 are the two main variants, respectively, the 32- and 64-bit versions of the same hash algorithm [30].

3.2.3 Comparison and Decision

MD5 performs better than SHA-256 [31] in running time and complexity fields. Although the complexity of both algorithms is the same, i.e., O(N), in time consumption, MD5 takes less time than SHA-256. However, MD5 has been shown to be insecure [32]. SHA-256 and SHA-512 have strong collision resistance, which means the message can not be decrypted by brute force. However, the advantages of SHA-256 over SHA-512 are that SHA-256 requires less computing power and uses less memory. So, we decide to choose SHA-256 as our default hashing function.

SHA-256 Performance

Figure 3.1 shows the running time for applying the SHA-256 hash function on the corresponding information during the generating proof stage. The data is based on 70 rounds of simulation. The result shows that although the running time for hashing can vary, the amount of time used every time keeps minuscule.

Figure 3.2 shows the first-byte value with hexadecimal after applying SHA-256 to the corresponding information. The result shows that the first byte of the hashed value is uniformly distributed. That means every outcome has an equal chance to occur. Moreover, that will prevent an adversary from learning the outcome and making predictions.

3.3 Overview of the proposed design

This section will firstly introduce the motivation for peers to perform the auditing. After that we discusses the participants. Then we describe the different roles of the participants. Finally, we remove some technical definitions to present a brief

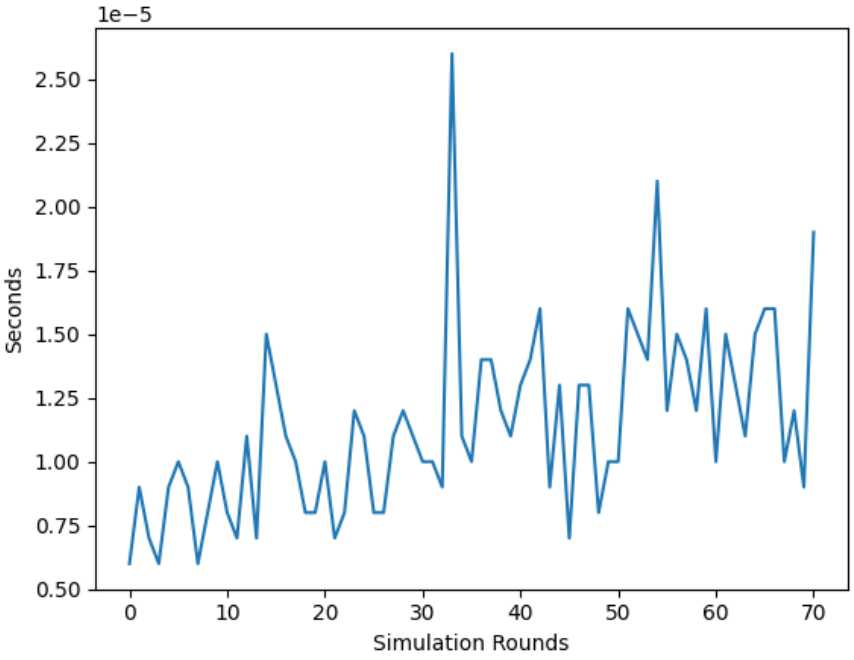

Figure 3.1: Hashing running time

overview of the message flow for our designed protocol, but we will elaborate in the following section.

3.3.1 Motivation

Although we omit the third-party auditor in the proposed protocol, every transaction on the blockchain still needs to pay the gas fee. Nevertheless, these expenditures pay off. After periodically auditing other nodes, the evaluation of honest peers will be maintained relatively high. Therefore, honest service providers can attract customers and bring higher profits than deceitful peers.

3.3.2 Participants

The participants in our designed protocol are the nodes in a decentralised storage file system. Every node is a network storage service provider with the same functionalities, which are also called peers. These peers store the data from users' uploads. In Swarm, data has been split into chunks and distributed to peers ge-

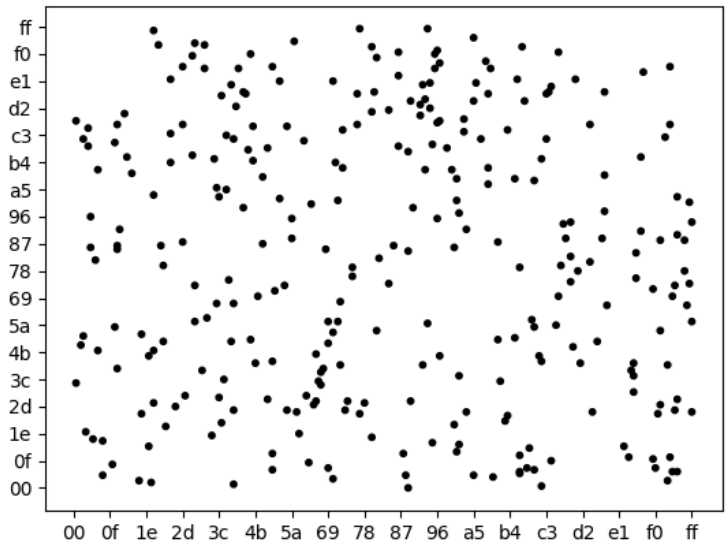

Figure 3.2: The distribution of the first byte hashed value

ographically close to the user for privacy protection. The user may want to make replicas for their data to ensure accessibility. Due to the user demand, the peer may store the same chunks as others. Peers need to prove that they honestly store those user-uploaded data.

3.3.3 Roles

We design two roles to finish a round of auditing: challenger and prover. There has no restriction on which role a peer has to act. Every peer can be a challenger or a prover. Most of the time, every peer acts as a challenger and a prover simultaneously.

Challenger

The challenger role has interested in sending challenges to audit other service providers. The challenger will create a challenge during the challenge phase based on the chunks stored in its storage. After computing the challenge, the challenger will send the challenge to all nodes that list on the Kademlia routing table. An

overview is illustrated in Figure 3.3. Then the challenger starts waiting for re-

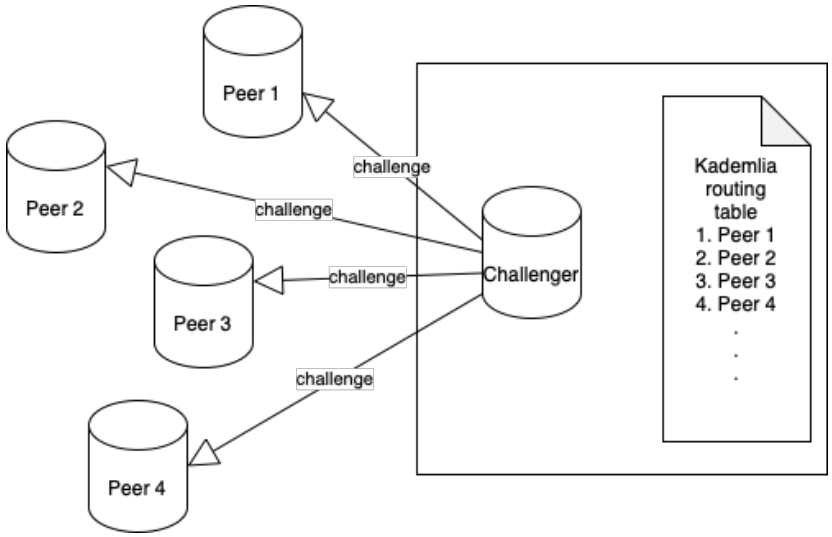

Figure 3.3: A high-level overview of a challenger in the challenge phase

sponses from other peers. The challenger finishes the challenge phase, and when it receives any response, the challenger starts the verification phase. When receiving proof from other peers, the challenger computes the validation. The challenger will store the result in its local storage and broadcast it to other nodes that the challenger can connect by the Kademlia routing table.

Prover

The prover role is interested in responding to the challenges sent by challengers in order to finish auditing. Since a positive validation will provide advantages when the service provider competes with others, the prover role will most likely honestly calculate the proofs. When the prover receives a challenge, it will start generating proof by the metadata of its stored files. The prover will send the proof back to where the challenge comes from when the proof is produced. An overview is illustrated in Figure 3.4. In this case, the prover's task is completed.

3.3.4 Message flow

The message flow is illustrated in Figure 3.5. It depicts a situation where a challenger sends the challenge to two different provers. In this case, when provers

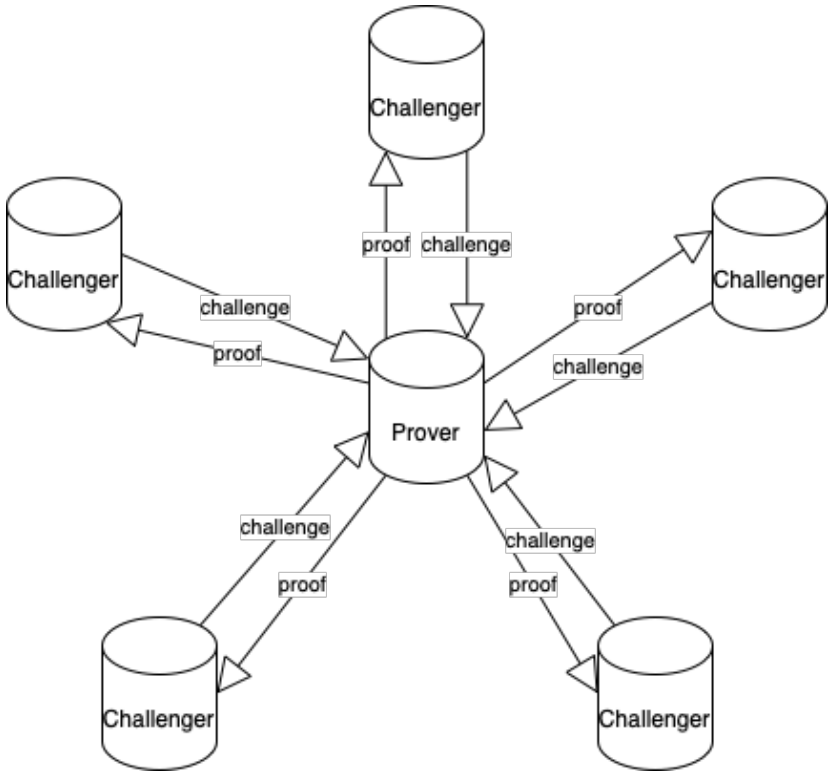

Figure 3.4: A high-level overview of a prover in the challenge phase

receive the challenge from the challenger, each prover generates a proof based on their stored chunks and responds to the challenger respectively. When the challenger receives the proof from a prover, the challenger will validate the received proof and broadcast the result to its neighbours.

3.4 Algorithms

This section will describe the related algorithms in detail. The Create Challenge function is designed for the challenger to generate a challenge based on locally stored chunks. The Prove Data Possession function is designed for the prover to calculate a proof based on the received challenge. The Verify Proof function is designed for the challenger to validate the received proof. The helper functions are designed for the nodes to get relative information.

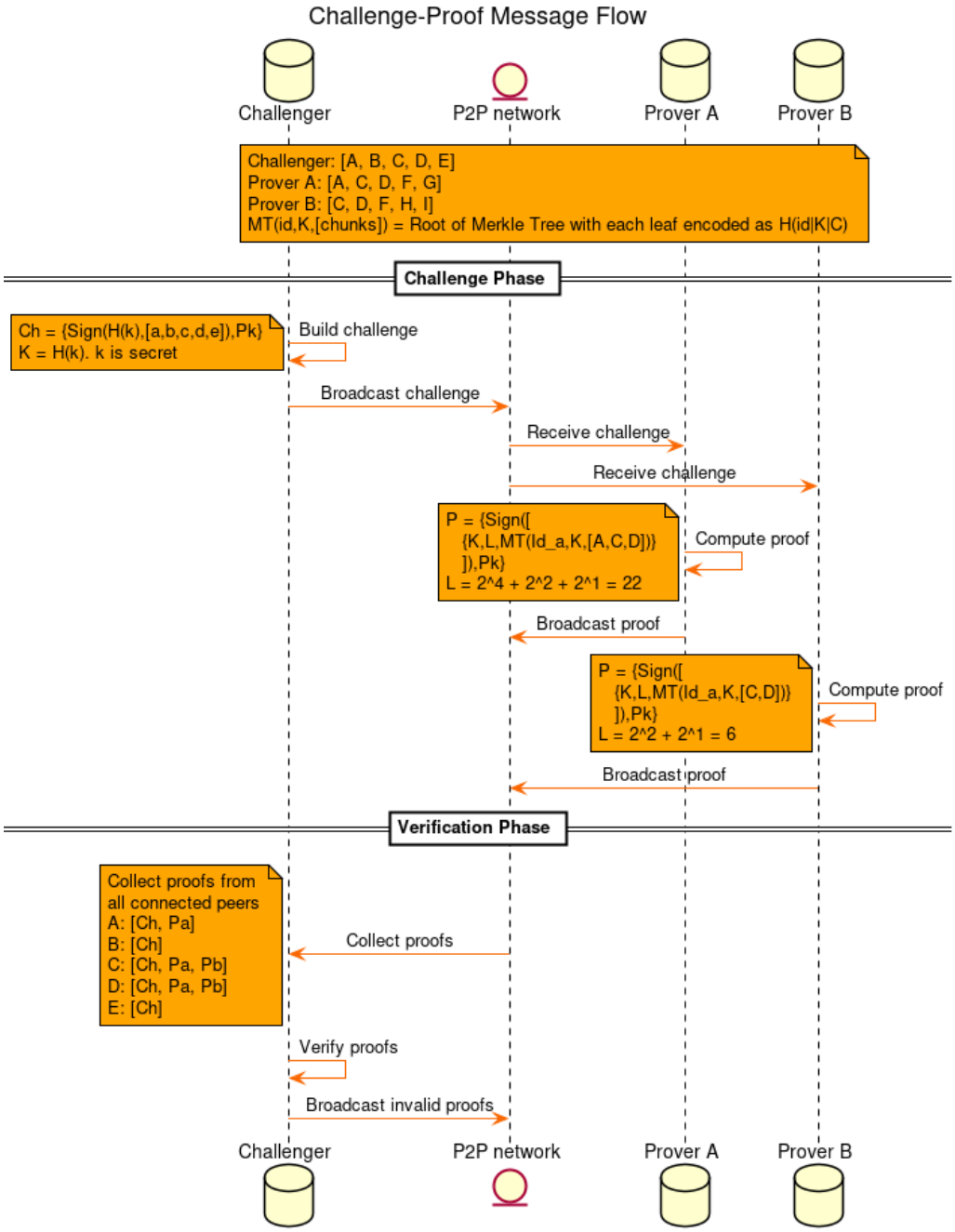

Figure 3.5: Challenge and Proof message flow [33]

3.4.1 Create Challenge

Algorithm 1 Create Challenge

```
1: func (v * VERIFIER) CreateChallenge(PKC, ChunkIDs)
2:     k ← GenerateRandomKey()
3:     K ← H(k)                          ▷ Commit to the nonce K.
4:     C ← [K, ChunkIDs]
5:     v.challenges[K] ← ChunkIDs
6:     return [C, Sign(H(C), PKC.priv), PKC.pub]
```

Figure 3.6: Pseudo Code for Create Challenge algorithm [33]

Create Challenge algorithm, in Figure 3.6 describes a challenger producing a challenge. A challenge has three components. They are chunks' metadata, digital signature and challenger's public key.

Metadata

Letter C indicates metadata for the locally stored chunks. The metadata is the combination of hashed nonce and chunks' addresses, the two most critical components of a challenge. To make a unique challenge, we need a nonce that is an abbreviation for 'number only used once'. We use the current timestamp as a seed for the pseudo-random number generator to ensure that the "random" numbers generated by the generator are unique even after rebooting the current node. Another critical context of a challenge is the local stored chunks' addresses. We get the chunks' addresses directly by reading the related information from the local hard drive in an offline test environment. With Swarm, we have different processing methods related to the helper functions, which will be explained together with the other related functions. We store hashed nonce and chunks' addresses as a key-value pair locally. In other words, the nonce mapping to the chunks' addresses.

Digital signature

A challenger uses its private key, encoded metadata, and applies the default hash function to produce the digital signature. That is important when the prover receives a challenge from a challenger. The digital signature can prove the challenge's integrity and be identified by a prover. If a prover fails to verify the signature by using the public key provided by the challenger, it proves that the challenge has been modified during network communication. Then the received challenge will be ignored.

Public key

The challenger needs to provide its public key for the recipient to identify the challenge's integrity and authenticity.

Finally, all three components are packaged into a challenge and sent to various peers.

3.4.2 Prove Data Possession

Prove data possession algorithm describes a prover calculating a proof. Before a prover calculates the proof, it must validate the received challenge. In order to identify the challenge, the prover first needs to encode the metadata, which is the first part of the challenge. Then the prover will apply the default hash function to the encoded metadata. Finally, the prover validates the challenge using the public key provided by the challenge.

If the validation fails, the prover will discard the received challenge. Otherwise, the prover starts to initialise variables. They are a counter and a byte array. The counter will perform a bit operation to indicate the matched chunk. The byte array has a fixed capacity depending on the proposed protocol's chosen hash function. For the default, when using SHA-256 as the default hash function, the capacity of the byte array is 32. The capacity of the byte array will automatically change if the default hash function is changed. After these two variables are initialised, the prover starts computing the proof.

Proof has three segments. They are corresponding information, digital signature and prover's public key.

Algorithm 2 Prove Data Possession

```
1:  func ProvePossession(PKC, Store, C)
2:      if ¬VerifySign(C) then
3:          return nil
4:      proof ← [0]^H.length
5:      L ← 0
6:      for i ← 0 to |C.ChunkIDs| do
7:          data ← Store.Read(C.ChunkIDs[i])
8:          if data ≠ nil then
9:              proof ← XOR(proof, H(data + C.K))
10:             L ← L & 1 << i                    ▷ Add 2^i to L
11:     proof ← H(PKC.pub + proof)
12:     P ← [C.K, L, proof]
13:     return [P, Sign(H(P), PKC.priv), PKC.pub]
```

Figure 3.7: Pseudo Code for Prove Data Possession algorithm [33]

Corresponding information

Letter P indicates corresponding information. The corresponding information combines the nonce from the received challenge, the counter and the byte array, the three most vital components of a proof. Since the same chunk will have the same address, if a chunk address matches one of the chunks' addresses from the challenge, the prover will use the matched address with helper functions to get the chunk data. Then the prover concatenates the chunk data and the nonce from the challenge. Then apply the default hash function to the concatenated data. Furthermore, the prover will apply the logic XOR between hashed value and the local variable byte array and use the result to overwrite the byte array. At the same time, apply shift one bit left, then apply logic AND to the local variable counter. From this point, whenever the local chunk's address matches the chunk address from the received challenge, the XOR logic will apply to the overwritten byte array and overwrite it again. Moreover, the counter will increase one bit in the binary format. Until the prover finishes checking all chunks' addresses from

the challenge, the prover concatenates the self public key and the final byte array. Then the combination will be encoded, and the prover will apply the default hash function to the encoded combination. Finally, the nonce from the challenge, the counter and the hashed result build the corresponding information.

Digital signature

A prover uses its private key, encoded corresponding information, and applies the default hash function to produce the digital signature. That is important when the challenger receives the proof as a response from the prover. The digital signature can prove the prover's integrity and be identified by the challenger. If the challenger fails to verify the signature using the prover's public key, it indicates that the proof has been modified during network communication. Then the received proof will be ignored.

Public key

The prover must provide its public key for the recipient to identify the proof's integrity and authenticity.

Finally, corresponding information, digital signature and prover's public key will be packed as proof and sent back as a response to the challenger.

3.4.3 Verify Proof

Verify proof algorithm, in Figure 3.8 describes a challenger validating a proof. The algorithm will output a boolean type result. True for accepted and false for rejected. The verify proof algorithm has three steps. They are preparation, calculation and comparison.

Preparation

During the preparation stage, the challenger must verify the received proof's authentication, initialise a byte array and find the related chunks' addresses.

In order to verify the digital signature, the challenger first needs to encode the corresponding information, which is the first part of the proof. Then the challenger applies the default hash function to the previous encoded output. The

Algorithm 3 Verify Proof

```
1:  func (v *VERIFIER) VerifyProof(PKC, Store, P)
2:      if ¬VerifySign(P) then
3:          return false
4:      MyProof ← [0]^{H.length}
5:      ChunkIDs ← v.challenges[K]
6:      for i ← L to 0 do
7:          if i & 1 = 1 then          ▷ Chunk is included in proof
8:              cP ← v.GetChunkProof(Store, ChunkIDs[i], P.K)
9:              if cP = nil then
10:                 return Error('chunk not found')
11:             MyProof ← XOR(MyProof, cP)
12:         i = i ≫ 1                  ▷ Bitwise right-shift
13:     MyProof ← H(P.pub + MyProof)
14:     return MyProof = P.proof
```

Figure 3.8: Pseudo Code for Verify Proof algorithm [33]

challenger uses the hashed value as a parameter and the prover's public key to validate the authentication of the proof. If the signature is invalid, the challenger will ignore the received proof. Otherwise, the challenger starts to initialise a byte array. The way to initialise a byte array is identical to how a prover initialises a byte array. The capacity of the byte array is 32 based on the default hash function. Then the challenger will use the nonce from the received proof to map the related local chunks' addresses. Based on the assumption, if the challenger deletes one or some of the chunks that have been used to generate the challenge, then the proposed protocol will not work. In other words, the challenger can only verify a received proof if the challenger has all the chunks of the proof.

Calculation

In the calculation stage, the challenger needs to calculate the proof for mapped chunks' addresses using the same strategy as a prover computes a proof.

The challenger needs the counter from the received proof to determine which chunk is included. The challenger will first transform the counter into binary format. If the current bit is one, that indicates the chunk has been stored by both the challenger and the prover. Then the challenger will use the helper function to get the hashed value for the combination of the target chunk address and nonce. Furthermore, the challenger will apply the logic XOR between hashed chunk data and the initialised byte array and use the result to overwrite the byte array. Moreover, the challenger will iterate all bits from the counter. If the current bit is one, the challenger will go through the same proof steps again. Until all bits in the counter have been looped, the calculation part is done and will pass the final byte array to the next step.

Comparison

The challenger needs to concatenate the prover's public key with the byte array from the last calculation step. Then apply the default hash function on the combination. Finally, make a comparison between hashed combination with the hashed chunks' proof from the received proof.

3.4.4 Other related functions

Helper functions are designed for different algorithms to achieve their goal. The helper functions for different test environments have other ways to implement them. For example, before deploying the proposed protocol on the Swarm test environment, all helper functions are designed for the local hard drive. Without using any application programming interface (API), we can get all related chunk information by standard package. The following will describe the Swarm test environment.

Read

The Read function aims to get the target chunk data. The target chunk data can be achieved by calling the *Get* function from the Swarm API. The *Get* function needs the target chunk Swarm address, and the Swarm address is one of the most significant parameters used to find the target chunk from the database. In this case, the Read function will only return the chunk data from all related chunk information.

The prover must use the Read function to determine whether a chunk mentioned by the received challenge is also stored locally. If the Read function returns a non-empty value that indicates there exists a matched chunk. The returned matched chunk data will participate in the rest operations.

The challenger needs to use the Read function to provide the currently matched chunk data in order to get the matched chunk proof when the challenger validates the received proof.

Has

The Has function indicates if the target chunk exists in the local storage. We get target chunk status by using the Swarm API and pass the chunk Swarm address as one of the parameters. The result is a boolean value indicating whether the target chunk is stored in this peer.

List Stored Keys

The List Stored Keys function aims to get all stored chunks addresses. We obtain the Swarm addresses of all chunks stored by the current peer by calling the *List-StoredChunks* method from the swarm API and convert the result to a suitable format.

Get Chunk Proof

The Get Chunk Proof function in Figure 3.9 aims to get the hashed value of the combination of target chunk data and nonce.

The algorithm takes three parameters, namely, store, id and nonce. In this case, the store is the challenger, the id is the target chunk address, and the nonce gets

Algorithm 4 Get Chunk Proof

```
1: func (v *VERIFIER) GetChunkProof(Store, id, K)
2:   if v.cP[K + id] = nil then   ▷ v.cP is a key-value store.
3:     if ¬Store.Has(id) then
4:       return nil
5:     data ← Store.Read(id)
6:     v.cP[K + id] ← H(data + K)
7:   return v.cP[K + id]
```

Figure 3.9: Pseudo Code for Get Chunk Proof algorithm [33]

from received proof. Firstly, the challenger needs to concatenate the nonce and chunk address and use the output to check whether the relative information has been stored previously. If no data is found, the challenger uses the target chunk address as the parameter to the Has function. If the Has function returns false, which indicates no chunk's information was found, the algorithm will return an empty value with an error message to the challenger. If the scenario occurs, the challenger cannot use the output of the Verify Proof algorithm to determine whether the received proof is valid because one of our protocol's assumptions no longer holds. Otherwise, the challenger uses the target chunk address as the parameter to call the Read function. The Read function will return the target data. Then the algorithm will concatenate the chunk's data and the nonce. Furthermore, applying the default hash function on the previous output. The concatenation of the nonce and chunk address will be set as the key that maps to the hashed value, and the key-value pair will be stored locally on the challenger. Finally, the algorithm will return the final hashed value to the challenger and use it to participate in subsequent operations.

Chapter 4

Experimental Evaluation

This chapter consists of two parts. The first part describes the test environment setup, and the second part represents the test results. The test setup is to experiment with our proposed protocol. Due to the time constraints of writing the thesis, we did not have time to deploy the protocol to the test cluster. Therefore, this test result cannot be evaluated as an actual deployment result. However, the protocol has already been deployed in the Swarm test net. Therefore, this test setup helps to achieve a mock deployment locally. The test aims to simulate the interactions between the various peers. Furthermore, get the time consumption for each step of every peer in the Swarm network by using our proposed protocol. Since the test runs locally, the second part of this chapter represents the test results without considering the impact of network latency and bandwidth effect on the protocol.

4.1 Experimental Setup and Data Set

This section presents the test environment setup, including configuring the Swarm test node, uploading data and the simulation strategy.

4.1.1 Preparation

Deploying a Swarm node needs sufficient xDAI for the Gas fee and at least one BZZ. Since xDAI and BZZ cost real currency, thus instead of using xDAI and BZZ to deploy a Swarm node, we alternatively use the Swarm test node.

The Swarm test node needs testnet tokens to deploy. Testnet Ethereum faucet is a tool for developers to get the testnet Ether. Many different providers can offer test tokens, and we choose Goerli Faucet [34]. The advantages of Goerli are that it can immediately send test tokens into the wallet account and provide a Goerli test network. So, we can use test Ether instead of xDAI and deploy the Swarm test nodes on the Goerli test network instead of deploying nodes on the Ethereum mainnet. Then we use Swarm test token gBZZ to replace BZZ for the Swarm test node.

4.1.2 Setup test node

We set up eight test nodes. They have similar configurations. Since we have a local test environment, thus the only difference between each node in the configurations is the port numbers. The figure 4.1 presents the configuration of a test node in the general case. A test node setup is complete after sending sufficient

```
{
   "password": "1",
   "api-addr": "1633",
   "debug-api-enable": true,
   "debug-api-addr": "localhost:1635",
   "data-dir": "bee_data_root/bee_1",
   "verbosity": 5,
   "bootnode": "",
   "full-node": true,
   "network-id": 10,
   "mainnet": false,
   "warmup-time": 0,
   "clef-signer-enable": false,
   "swap-initial-deposit": "1000000000000000",
   "p2p-addr": 1634,
   "swap-endpoint": "https://cloudflare-eth.com",
   "cors-allowed-origins": "*"
}
```

Figure 4.1: General test node configuration

test Ethereum tokens and one gBZZ to the given address. Finally, we set up the other seven test nodes with the corresponding ports.

4.1.3 Test data

The proposed protocol aims to audit redundancy between different peers. To achieve that, we need serval test files. The idea was to use a terminal command line to produce a given-size file. However, files of any given size generated using the command line contain only the same characters. However, we consider that although the file name and file size are not the same between different files, the file content is identical. Though Swarm will split different files into chunks and give each chunk a unique chunk address, it may cause a potential impact. In order to avoid any possible potential interference factors, we finally abandoned the idea of creating test data using a terminal command line.

The alternative solution is to generate text files with the specified size consisting of random characters. Using randomised characters for every text file minimises the potential for inaccurate test results. We have two types of test files, image type and text type. The individual image file size is less than one megabit. Text files vary in size, from hundreds of megabits to one gigabit. We use six images in jpg format and eight text files in txt format.

When the test files have been generated, we start uploading test files into test nodes. Since the proposed protocol aims to audit the redundancy in the decentralised storage network, We need each test node to contain some redundant data from the current test environment. The strategy of uploading files is to try to make every test node have some files identical to others and to differentiate the file size for each test node. We named the eight test nodes following the numerical order from node zero to node seven. Furthermore, we tried to keep node zero, having the least amount of chunks when uploading data, gradually increasing the number of chunks owned by each node. Finally, node seven has the most significant amount of chunks.

The capacity for stored data chunks of each test node is given in the Figure 4.2

4.1.4 Simulation

Due to the time constraints of this thesis, we decided to simulate interactions between different peers locally instead of testing the proposed protocol in a Swarm

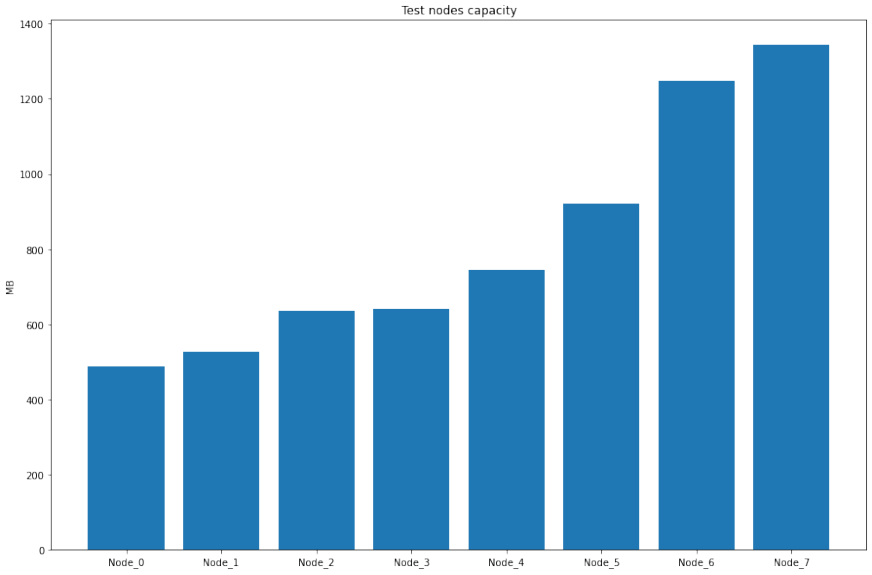

Figure 4.2: Node Capacity

cluster. Since the interaction between different peers through sending network requests and responses, thus the first thing is to select two different peers. Every test node uses a unique port number, the most obvious way to differentiate swarm test nodes. So, we manually put all possible port numbers into a set and randomly selected two of them. One will act as a challenger, and another will act as a prover. Then, these two peers will start to communicate with each other. A round of simulation finished when the challenger finished the verification stage. The more rounds of simulation we have, every test node is more likely to have even opportunity to interact with others. In other words, the simulation is closer to the situation of real P2P communication in a topology network.

4.2 Experimental Results

This section presents the result after running 420 rounds of simulations. In addition, we will briefly introduce the used statistical tools, and the result evaluation will be given.

4.2.1 Overview

The result of every round simulation is recorded into a log file. During the pre-processing, we keep the different protocol stages' running times records for every round of the simulations and format the record from a log file into a CSV file. In order to present the overview of the simulations, we use the stacked bar graph to indicate the performance for different stages.

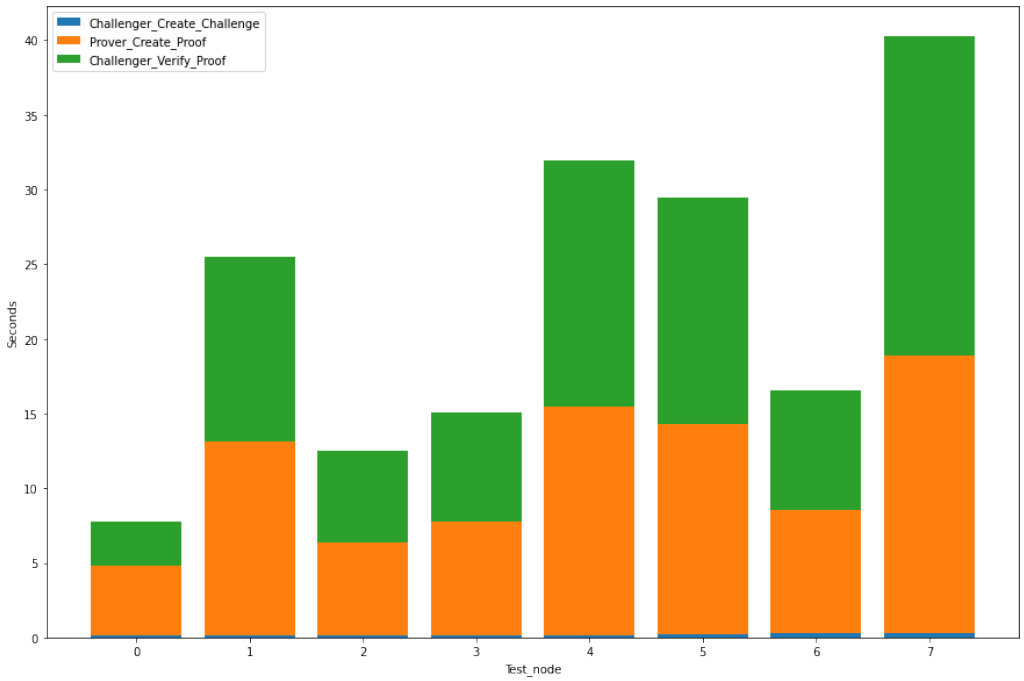

Figure 4.3: Time spent by each test node in one round of simulation

The figure4.3 is a stacked bar graph showing the average time spent for every test node during the auditing stages. Since the stored chunks for test nodes are not symmetrically distributed, the data used to generate the graph is the median value of time consumption for every auditing stage of every test node. The test nodes have one thing in common except node zero, which is that the verifying proof stage is roughly the same as the proving proof stage. The reason node zero uses less time in the verifying stage than the time it uses to create a challenge is caused by the strategy we used to upload test files. Therefore, after randomly selecting the test files, although node zero has the least amount of chunks, it contains unique test data, and this unique data accounts for a considerable propor-

tion of the test data of node zero. In other words, node zero has fewer redundant chunks than other test nodes. So, when other test nodes compute a proof, it will cost less time since they do have not many identical data chunks with node zero. A node containing a large amount of unique data chunks also explains why some of the nodes have a larger size of stored chunks but the time used to finish a round of simulation is less than the node that stores fewer data chunks. For example, nodes two and three compare with node one, and node six compares with nodes four and five. There is one thing in common that is no matter how many data chunks a node stores, the time to create a challenge has a tiny proportion compared to other stages.

4.2.2 Time usage analysis

In this section, we will give a brief introduction to the box plot first, then, using box plots, present the analysis of time consumption for three stages of the proposed protocol.

Box plot

A box plot is a statistical graph used to display information about the dispersion of a set of data. The box plot reflects the characteristics of the original data distribution. A box plot in Figure4.4 describes the discreteness of given data with the minimum value, maximum value, lower quartile value, median value and upper quartile value.

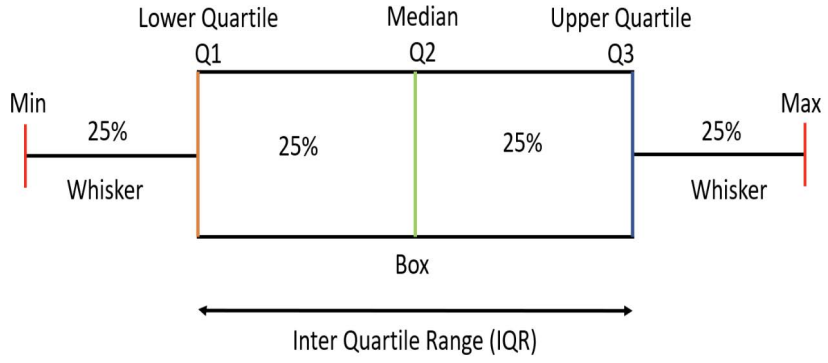

Figure 4.4: Box plot [35]

After sorting the data based on the interest column and dividing it into four equal parts, the numbers at the three dividing points are the quartiles; the three are the lower, the second, and the upper. So, the lower quartile indicates the value from the data at the point of 25%. The median is also known as the second quartile, which is the value from the data at the point of 50% The upper quartile shows the value at the point of 75% from the data.

Creating Challenge

Figure 4.5 indicates a challenger's time consumption to generate a challenge for all test nodes during the 420 rounds simulation. From the result, the maximum value of generating a challenge for a challenger is around 0.375 seconds, and the minimum value is less than 0.1 seconds. That is reasonable since test node eight contains roughly three times the total amount of data chunks as test node one. The maximum value is also roughly three times the minimum. So, the number of node stored chunks impacts the duration of generating a challenge. In this test environment, the average time consumption is the green triangle presented in Figure 4.5, which is calculated by the formula 4.1, and the value is around 0.2 seconds. The median value of creating a challenge is around 0.16 seconds, less than the average time. Since the distribution for the number of chunks for every test node is skewed and when there are two outliers, the median value better explains the "center" of the time consumption for creating a challenge. There have outliers which have greater value than the maximum value. We believe that is caused by the laptop running out of memory and leading to reducing computational power.

$$\overline{x} = \frac{\sum_1^n xi}{n} \tag{4.1}$$

Generating Proof

Figure 4.6 indicates a prover's time consumption to create a proof for all test nodes during the 420 rounds simulation. The result shows that the maximum value of creating a proof for a prover is around 32 seconds, and the minimum value is less than 0.5 seconds. If the challenger is the test node one which only contains 0.12 million chunks, and the produced challenge is based on the test

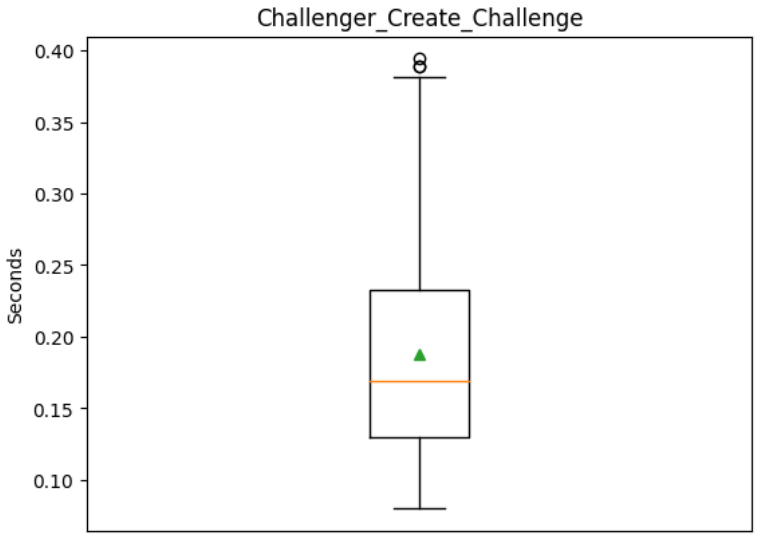

Figure 4.5: Box plot for generating a challenge

node one locally stored chunks. Any other test nodes only iterate 0.12 million times in order to produce proof. So, it is reasonable to have a relatively small minimum value. The maximum value is roughly 32 seconds. Two factors can affect the maximum value. First, the scenario is the opposite of having a relatively small minimum number of values. The challenger and prover are most like be the test node seven and test node eight. That means the challenge contains a relatively large amount of data chunks, and the prover also has a large set of data chunks. Although the data structure for the data chunk is key-value pairs in the Swarm, the prover iterates all chunks from the received challenge cost time and using the iterated chunk address as a key to map locally stored chunk also need time. Second, due to the hardware limitation, the computation speed will also drop when nearly running out of the laptop memory. The average time consumption is around 12.5 seconds. The median value is almost the same as the average time. There also have outliers caused by the hardware limitation.

Validating Proof

Figure 4.7 indicates a challenger time consumption to verify a proof for all test nodes during the 420 rounds simulation. The result shows that the maximum value of verifying proof for a challenger is around 34 seconds, and the minimum

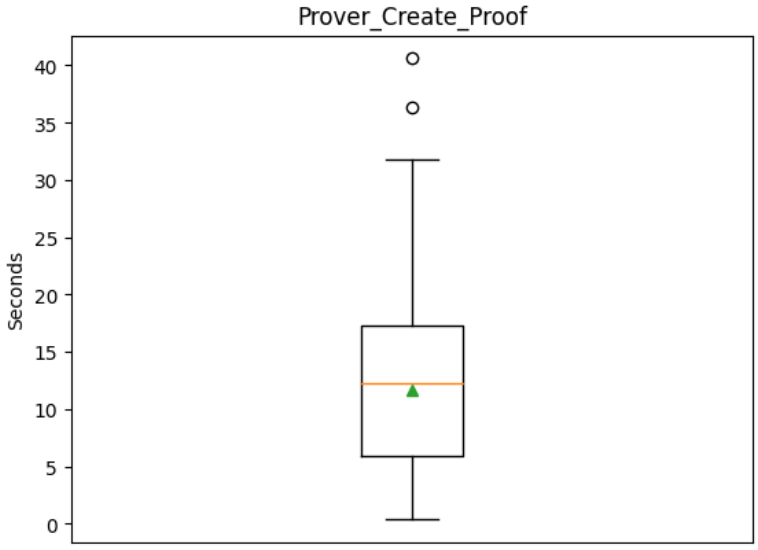

Figure 4.6: Box plot for generating a proof

value is less than 1 second. Same as the scenario of creating a proof. If the prover is the test node one, less than 0.12 million chunks generate the proof. So, it is reasonable to have a relatively small minimum value. The maximum value is roughly 34 seconds. Similar to the time consumption of generating a proof, two previous factors can also affect the maximum value when verifying a proof. The average time consumption is around 12.5 seconds. The median value is almost the same as the average time. There also have outliers caused by the hardware limitation.

Comparing the three stages

Figure4.8 shows the comparison between three stages. We deliberately deleted the outliers that resulted in both creating proof and verifying proof stages. The purpose is to reduce the scale of the overall image. So, the box plot for creating challenge stage appears slightly more prominent. Comparing the time-consuming of creating a challenge stage with other stages, we can intuitively find a significant gap. Obtaining the main parameters required to generate the challenge only by traversing the locally stored data chunks is significant because the time required is much less than the other two stages. Next, we will compare the

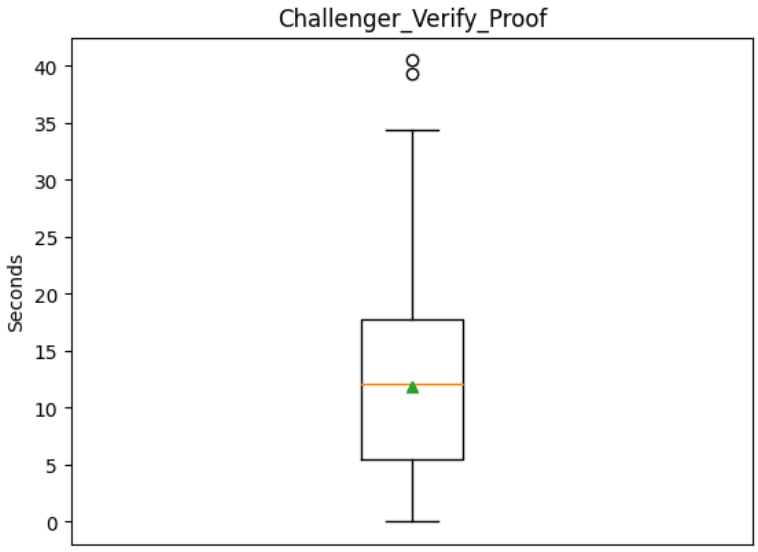

Figure 4.7: Box plot for verifying a proof

two stages of creating proof and verifying proof. First, the two stages overall have similar-sized interquartile ranges. The range and maximum value between the quartiles of the stage of verifying the proof are slightly more extensive than the stage of creating the proof. That also shows the greater data dispersion in the verifying proof stage. So, compared with the challenged prover, the time consumed by the challenger to verify the proof depends more on the number of chunks involved in the received proof. The higher the number of chunks, the longer it takes to identify the proof.

4.2.3 Estimated Running Time

This section will present the estimated time consumption by using a point plot graph. We describe the uncertainty shown on the graph first, and then we explain the reasons that cause the uncertainty.

Figure 4.9 is a point plot graph. It indicates the estimated time consumption and uncertainty for every auditing stage for each test node. The estimated time for generating a challenge for all test nodes is less than a second and has no uncertainty. That means the provided data can clearly describe the time consumption for creating a challenge. However, the other two stages have uncertainties and

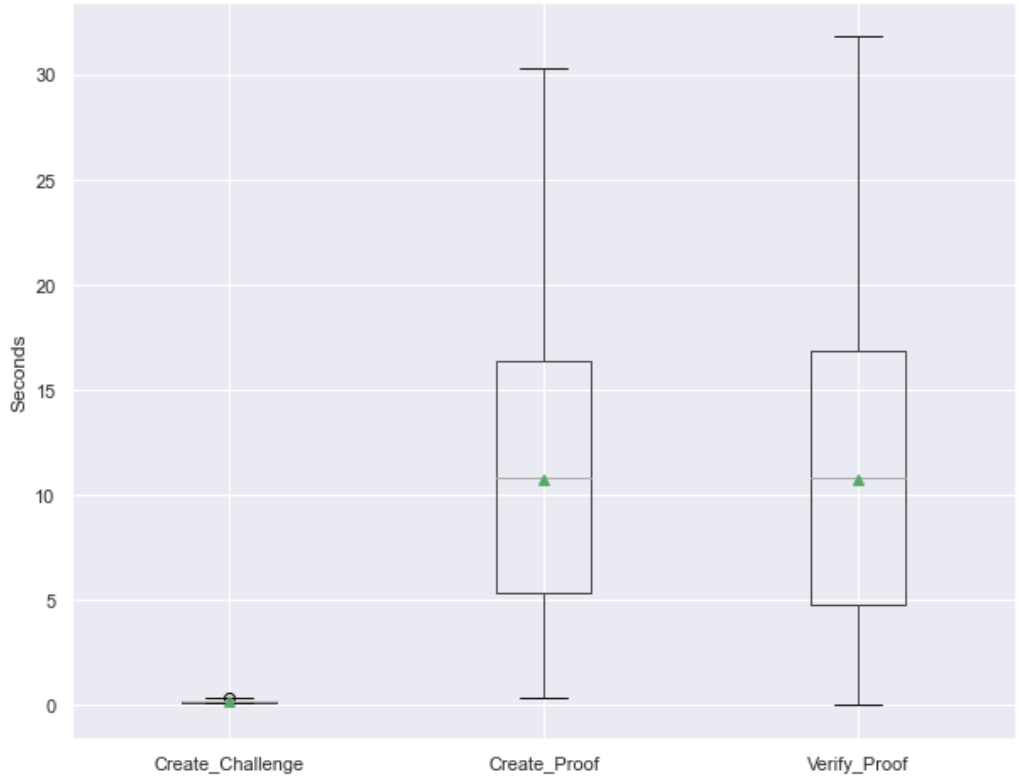

Figure 4.8: Box plot for verifying a proof

are caused by two reasons, one caused by general limitation and another caused by redundancy data distributed. First, the uncertainty is caused by the limitation of the number of simulation rounds. If we have 4200 rounds rather than 420, the estimated result will have less uncertainty since the data has been better explained. When the data has been well explained, the uncertainty will reduce when estimating the time-consuming. Second, a test node may contain almost all replicated data chunks with another test node, or it may only contain a small proportional amount of data chunks with another due to the redundancy distribution. However, this kind of uncertainty is what the protocol will face if it can finally be deployed on the Swarm network. Since in the Swarm network, we cannot control the stored chunks of a node. So, if the distribution of the redundant data chunks causes uncertainty, it proves that the simulation succeeded in simulating the actual scenario. In conclusion, if we have no hardware limitation to run the simulations, then the uncertainty highly depends on the distribution of

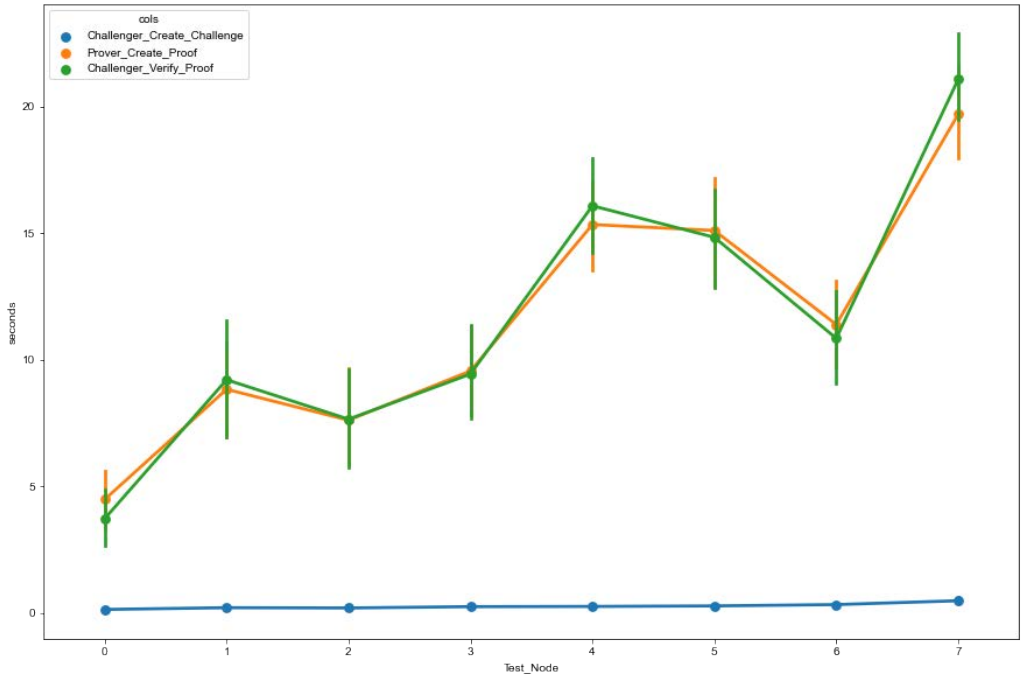

Figure 4.9: Estimated time for each stage for every test node

the redundant data chunks.

Chapter 5

Discussion

This chapter will discuss the outcomes presented in chapter four and their insight by describing the limitations and strengths. In the end, we also provide future work to improve our protocol.

5.1 Outcomes and Insight

The performances for all three stages in our proposed protocol have outliers. Since we only have the local test environment, the outliers are likely caused by the local machine memory running out. The limitation of the local test environment is evident. Firstly, we have limitations on hardware. The simulation was highly dependent on the CPU and the available memory. The performance is rapidly reduced when running out of the available hardware resources, producing outliers. Secondly, we ignore the network traffic. During a local test simulation, the recipients will immediately receive the message. Moreover, we do not set any period for a challenger to create a challenge and broadcast it to all its neighbours. We cannot measure whether frequently sending challenges will cause exceptional network overhead. If the scenario costs the network overhead, outliers will occur as well. We cannot say that the scenario will never occur. However, our protocol has the advantage when a prover responds to a challenger. So, the network traffic from a prover side is reduced. As the proof size is fixed, reducing the requirement for network bandwidth.

The outcomes from chapter four also indicate that if a Swarm node has high re-

dundancy compared to others, the performance is inversely proportional to the redundancy ratio. In other words, the more redundant file chunks, the lower the efficiency.

Choosing a proper hash function is significant. If a node contains a large number of redundant chunks, although the outcome shows the hashing only costs a little time to finish, the number of times call the hash function will bring it to be significant. In our protocol, two factors are the most time-consuming. The first one is the hash function, and the second one is the calculation. We are using SHA-256 as the hash function, ensuring the result is under security conditions and only takes up relatively few hardware resources compared to other hashing methods. We are using bit-operation to handle the calculation; for a computer, using the basic level for calculation will take the least time compared to other data types.

5.2 Future work

This section will present the extensions that could make our protocol more suitable and profitable for deploying on the Swarm.

Sharing Keys

After a Swarm node setup, secret key pairs are assigned to the Swarm address. We can use those key pairs as the role's private and public keys to reducing unnecessary redundancy. For the generality, the proposed protocol can also deploy on other decentralised storage systems. Based on the idea of decoupling programming, a better solution is that the protocol will auto-check whether the storage system has been assigned secret key pairs when initialising the protocol. If they are already existing, then we can reuse them. We believe, for most cases, the node for a decentralised storage system will have secret key pairs, but finding a common way to get them for all possible systems will be another challenge.

Motivating Nodes

The way we handle the invalid proofs can be approved from broadcasting the invalid proofs to broadcasting both valid and invalid proofs. Although that will generate some overheads in storage and network transitions, if the node only

keeps the valid proofs locally, the adversary may complain that he has not received anything from the challenger. Since the protocol does not involve any time assumption, the protocol cannot determine whether a prover has failed if we do not receive any response from one prover. And that is why we had the related assumption described in section 3.1.2. The protocol can be better if it can motivate those provers who can show they are an honest service provider. Since making a transition on blockchain will cost gas fees. If we can implement a reputation system and distribute fewer chunks to those who only have a low reputation rate, that will bring more profit to those honest nodes. Based on the reputation system, all nodes will be potentially affected and motivated to perform clients' requests honestly.

Chapter 6

Conclusions

This thesis proposes a protocol for auditing between peers in a P2P network without employing any third-party auditor. It fills the vacancy of being out of third-party auditor audits that are impossible with existing PoS variants. The proposed protocol can ensure that the audit can be completed without reading the original content of the file to maximize the protection of user privacy. The prover is more inclined to store user data honestly because all false claims and unauthorized modifications of user data will lead to credit loss and waste gas generated by creating transactions. An added benefit of using our proposed protocol is omitting the cost of hiring a third party. Although each transaction on the blockchain must pay a certain amount of gas fee, if the audit period is determined in an appropriate range, the gas fee problem caused by sending challenges and responses will be controlled to the greatest extent.

Our proposed protocol builds on the challenge-response protocol. The challenger creates a challenge based on local storage chunks and sends it to all peers in the routing table maintained locally by the challenger. After the peer receives the challenge, it compares the local storage chunks with the received challenge, generates a response, and replies to the challenger. The challenger finally confirms whether the prover is honestly storing the user's data by identifying the validation of the received response.

We set up the test environment locally for the proposed protocol. These include the configuration of swarm test nodes, uploading data of different sizes and types,

and local simulation testing after deploying the proposed protocol on the Swarm. The experimental results show that two factors determine the performance of the proposed protocol. The first one is the number of data chunks owned by both parties. The second one is the proportion of two nodes that contain identical redundant file chunks. At the same time, when using the SHA-256 hash function to hash a unit data chunk, it takes almost negligible time. Experiments show that our proposed protocol is feasible.

List of Figures

List of Tables

Appendix A

Instructions to Compile and Run Protocol

1. Go version request: 1.17.10
2. Using the URL `git@github.com:snarlorg/bee-snarl.git` clone the repository.
3. Finding the **.env** file and replace the **BEE_SWAP_ENDPOINT** variable with your own **infura** URL.
4. If you do not have a suitable infura URL, go to `https://infura.io` and register. When you have your valid account, click the **CREATE NEW PROJECT** button at the top right corner. Then, select **Ethereum** from the **PRODUCT** drop box and give your project a name, then click **CREATE** button. Then, click the **SETTINGS** button on the right side of your project at the infura user dashboard page. You will find the URL under the **ENDPOINTS** label and choose the URL start with **wss**.
5. Open a terminal and run the command make binary && sudo cp dist/bee /usr/local /go/bin/bee && ./run_node.sh -r 1 at your cloned folder path to build the first Swarm test node. (You may need to adjust the paths by editing **run_node.sh** file.)
6. In order to activate a Swarm test node, you need to send sufficient test tokens to the address shown on the terminal. You can find more related infor-

mation here `https://docs.ethswarm.org/docs/introduction/terminology#xbzz-token`.

7. Now, you can upload your test files. You can find more related information at https:// docs.ethswarm.org/docs/access-the-swarm/upload-and-download
8. Keep the test node running while opening another terminal and run the following command **curl -X GET http://localhost:1633/pos/create** if the terminal prints out relative information, then you are successfully set up a test node.
9. If you want to run the local simulations, then you need to set up another seven test nodes by repeating the steps from four to six. For the command shown in step four, you need to change the number after **-r** , which indicates the number of the test node.
10. When you finish set up all eight Swarm test node, then you can run the **TestGetResult** test that located at **bee-snarl/pkg/api/pos_test.go** while keep all the test nodes running.

Bibliography

[1] Brian Mackenzie. Cloud storage software statistics and trends, Jun 2020. https://www.trustradius.com/vendor-blog/cloud-storage-statistics-and-trends#behavior.

[2] Mehul A Shah, Mary Baker, Jeffrey C Mogul, Ram Swaminathan, et al. Auditing to keep online storage services honest. In *HotOS*, 2007.

[3] Seny Kamara. Proofs of storage: Theory, constructions and applications. In *International Conference on Algebraic Informatics*, pages 7–8. Springer, 2013.

[4] Robert Shirey. Internet security glossary, version 2. 2007.

[5] Vitalik Buterin. Ethereum whitepaper. https://ethereum.org/en/whitepaper/, 2013.

[6] 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. Springer, 2002.

[7] Mehdi Sookhak, Abdullah Gani, Hamid Talebian, Adnan Akhunzada, Samee U Khan, Rajkumar Buyya, and Albert Y Zomaya. Remote data auditing in cloud computing environments: a survey, taxonomy, and open issues. *ACM Computing Surveys (CSUR)*, 47(4):1–34, 2015.

[8] Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597, 2007.

[9] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609, 2007.

[10] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *European symposium on research in computer security*, pages 355–370. Springer, 2009.

[11] 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. Springer, 2008.

[12] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *2010 proceedings ieee infocom*, pages 1–9. Ieee, 2010.

[13] Cong Wang, Sherman SM Chow, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE transactions on computers*, 62(2):362–375, 2011.

[14] Hovav Shacham and Brent Waters. Compact proofs of retrievability. *Journal of cryptology*, 26(3):442–483, 2013.

[15] Yuan Zhang, Chunxiang Xu, Hongwei Li, and Xiaohui Liang. Cryptographic public verification of data integrity for cloud storage systems. *IEEE Cloud Computing*, 3(5):44–52, 2016.

[16] Yuan Zhang, Chunxiang Xu, Xiaohui Liang, Hongwei Li, Yi Mu, and Xiaojun Zhang. Efficient public verification of data integrity for cloud storage systems from indistinguishability obfuscation. *IEEE Transactions on Information Forensics and Security*, 12(3):676–688, 2016.

[17] Jared Stauffer. What is xdai? how do i use xdai? a simple explanation. `https://jaredstauffer.medium.com/what-is-xdai-how-do-i-use-xdai-a-simple-explanation-7440cbaf1df6`, 05 2020.

[18] J Benet. Filecoin: A decentralized storage network. protocol labs, 2017.

[19] Kevin D Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 43–54, 2009.

[20] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Osama Khan, Lea Kissner, Zachary Peterson, and Dawn Song. Remote data checking using provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):1–34, 2011.

[21] Anjia Yang, Jia Xu, Jian Weng, Jianying Zhou, and Duncan S. Wong. Lightweight and privacy-preserving delegatable proofs of storage with data dynamics in cloud storage. *IEEE Transactions on Cloud Computing*, 9(1):212–225, 2021.

[22] Juan Benet, David Dalrymple, and Nicola Greco. Proof of replication, 2017.

[23] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Annual Cryptology Conference*, pages 585–605. Springer, 2015.

[24] Swarm team. Swarm, storage and communication infrastructure for a self-sovereign digital society. Technical report, 2021.

[25] Swarm bee documentation. `https://docs.ethswarm.org/docs/`. Accessed: 02-06-2022.

[26] Patrick Schueffel, Nikolaj Groeneweg, and Rico Baldegger. The crypto encyclopedia. Technical report, Growth publisher, 2019.

[27] Ivan Damgård, Chaya Ganesh, and Claudio Orlandi. Proofs of replicated storage without timing assumptions. In *Annual International Cryptology Conference*, pages 355–380. Springer, 2019.

[28] Ronald Rivest. The md5 message-digest algorithm. Technical report, 1992.

[29] Chu-Hsing Lin, Yi-Shiung Yeh, Shih-Pei Chien, Chen-Yu Lee, and Hung-Sheng Chien. Generalized secure hash algorithm: Sha-x. In *2011 IEEE EUROCON-International Conference on Computer as a Tool*, pages 1–4. IEEE, 2011.

[30] Raffaele Martino and Alessandro Cilardo. Sha-2 acceleration meeting the needs of emerging applications: A comparative survey. *IEEE Access*, 8:28415–28436, 2020.

[31] Dian Rachmawati, JT Tarigan, and ABC Ginting. A comparative study of message digest 5 (md5) and sha256 algorithm. In *Journal of Physics: Conference Series*, volume 978, page 012116. IOP Publishing, 2018.

[32] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35. Springer, 2005.

[33] Racin Nygaard and Hein Meling. Storage guarantees in decentralized storage systems. *Internal work to be submitted for publication*, 2022.

[34] Goerli faucet. `https://goerlifaucet.com/`. Accessed:04-06-2022.

[35] Box plot. `https://www.geeksforgeeks.org/box-plot/`. Accessed: 12-06-2022.

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no