DISTRIBUTED STORAGE WITH VERIFICATION

IVAN LUCHEV, 202202879

MASTER'S THESIS

June 2024

Advisor: Niels Olof Bouvin

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

DISTRIBUTED STORAGE WITH VERIFICATION

Master's Thesis

Department of Computer Science
Faculty of Natural Sciences
Aarhus University

June 2024

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABSTRACT

Cloud storage is an essential aspect of modern computing, enabling users to store and access data globally. It is being used by individuals, businesses, and organizations for various applications, including file sharing, data backup, and collaboration.

Alternative to cloud storage, we have decentralized storage systems, which focus especially on privacy, security, censorship resistance, but not so much on data integrity. Cloud storage is already employing various techniques to monitor and ensure data integrity, such as checksums, error correction codes, and replication. Due to the decentralized nature of decentralized storage systems, these techniques may not be directly applicable, as they are more susceptible to attacks and undesirable behaviors from the network's participants.

This thesis aims to design and implement a secure and fault-tolerant decentralized storage system that can rival centralized solutions. With regards to the integrity of data stored in decentralized networks, a critical issue not fully resolved by existing systems. The proposed system will operate on peer-to-peer (P2P) networks, leveraging structured network topologies such as Kademlia, to ensure efficient routing and resource discovery.

The primary contribution of this work is the development of a storage system with an integrity verification mechanism that ensures data consistency and reliability, mitigating the risk of data corruption and loss. This involves evaluating proof of retrievability (PoR) techniques and a reputation system based on a ledger to combat malicious nodes. The system's scalability, performance, resilience, reliability, availability, and security are also considered to ensure comprehensive evaluation and testing.

NOMENCLATURE

In this section we provide a list of abbreviations and definitions used throughout this thesis.

1. **Decentralized network** — A network where no single entity has control over the network. While there are no minimum requirements for a network to be decentralized, when we talk about a decentralized network we will expect no single entity to have control of the majority of the nodes and for the network to have at least 10 nodes.
2. **Keeper** — A node, which stores files/data and provides them when requested.
3. **Verifier** — A node which performs audits on the files in the network in order to ensure their integrity.
4. **PoR** — Proof of Retrievability. A scheme that allows a client to check if a peer stores the data it claims to store efficiently using cryptography.

1 INTRODUCTION

Cloud storage is a key component of modern computing. It allows users to store and access their data from anywhere in the world. A lot of the popular cloud storage providers are centralized, meaning owned and operated by a single entity. On one hand, this is good for the user because the entity provides a friendly interface and customer support in exchange for a fee. On the other hand, the user has to trust the entity to keep their data safe and private and not misuse it. They also might prefer to pay with their storage or computation power instead of money, or they might not want to be locked into a single provider. Such cases are often when the user is a university, an NGO, or occasionally private users.

To address such use cases and the concerns with cloud storage, decentralized storage systems have been proposed.

We will focus on the most popular type of decentralized storage, which is decentralized hash tables (DHT). They are often used as the basis of NoSQL databases, such as Cassandra [3], which are used by companies such as Facebook, Twitter, and Netflix, in order to serve millions of users. While in the example above these databases are used within the premises of a cloud or a private datacenter, they are mostly used as decentralized storage solutions, therefore they focus on privacy, security, fault tolerance, and resistance to censorship.

The goal of this thesis is to design and implement a secure and fault-tolerant decentralized storage system that can compete with centralized storage systems, whose features we've listed in 2.2.3. Decentralized networks work on top of untrusted networks (unregulated networks with potential security risks), such as the Internet. In particular, we will focus on the integrity of the data stored on the network, as this is the main problem that is not addressed by existing decentralized storage systems. In the following sections, we will introduce decentralized networks, define what security and integrity means, discuss the motivation for this work, and propose a solution to the integrity problem in modern decentralized networks. We will then talk about the requirements for the system and how we will evaluate it.

1.1 OVERVIEW OF DECENTRALIZED NETWORKS

Decentralized storage is implemented on top of peer to peer (P2P) networks. They are a type of network where each node acts as both a client and a server. This means that each node can request and provide services to other nodes. There is also no central server, although there may be a central directory to help nodes find each other.

Decentralized networks can be classified into two categories — unstructured and structured. Structured networks have a predefined topology and are more efficient in terms of routing and resource discovery. Often used topologies are circular and binary trees. The most popular structured network is Kademlia [12], which uses a binary tree topol-

ogy. It is also the foundation for most modern decentralized storage systems.

We discuss decentralized networks in more detail in 2.1.

1.2 SECURITY

Security is a major concern in decentralized networks because the network is open to anyone, and there is no central authority to enforce rules, in contrast to centralized storage systems. Peers can join and leave the network at any time, and they can lie about their identity and the data they store. We call such peers malicious.

We go into more detail what security means in 2.1.1. In this thesis we will focus on a subclass of attacks - storage attacks.

1.2.1 *Storage attacks*

Storage attacks are mostly ignored by the literature, because decentralized networks are often designed to drop old files based on some criteria, such as popularity or age. This is done to save space and to keep the network up to date. However, if we want durable storage, we need to address these attacks.

Storage attacks can be classified into two categories:

1. **Data availability attacks** — an attacker claims to store data, but does not.
2. **Data integrity attacks** — an attacker claims to store data, but stores different data.

Checking if a node stores the data it claims to store can be as simple as asking the node to return the data. However, this is a very inefficient and bandwidth-consuming method. Ideally, we would like to reduce the amount of traffic between nodes and still be able to check if the data is stored correctly.

1.3 PROOF OF RETRIEVABILITY

Proof of retrievability (PoR) refers to the ability of a prover to convince a verifier that it is storing a file. This could be as simple as sending the whole file, sending a hash of the file, or running a modern cryptographic PoR protocol. We are mainly interested in the last one, as it could be the most efficient one.

In this thesis we will explore the use of PoR protocols to solve the integrity problem and prevent storage attacks. We will look into how performant PoR protocols are and whether they are viable for decentralized networks.

We will dive deeper into PoR in 2.2.

1.4 REQUIREMENTS FOR THE SYSTEM

If we want to design a decentralized storage system that is similar and can compete with centralized systems, we need to provide the same guarantees and features:

1. **Scalability** — the system should be able to handle many users and data.
2. **Performance** — the system should be fast.
3. **Resilience** — the system should be able to recover from attacks.
4. **Reliability** — the system should be able to recover from failures.
5. **Availability** — data stored on the network should be accessible at all times.
6. **Integrity** — data stored on the network should not be possible to be tampered with.
7. **Security** — data stored on the network, and accessing the data should be secure.

Scalability is covered by most networks, regardless of being centralized or decentralized. Allowing many peers to join the network is a key feature of decentralized networks. **Performance** is also covered by most modern networks, as they provide $O(\log n)$ query time. **Resilience** is for the most part covered by the security additions to the networks, such as S/Kademlia **??**. An exception is the storage attacks, which are not addressed. **Reliability** is covered by rebalancing the network when a node leaves or joins. Most networks also provide some form of redundancy, which allows the network to recover from failures. **Availability** follows from reliability and resilience. **Security** can be solved by using encryption, cover traffic, onion routing, etc. **Integrity** is not the focus of existing networks, as discussed in 1.2.1.

These requirements have some overlap. In order to achieve resilience, we need to guarantee the integrity of the data. In order to achieve performance, we need optimized way to check the integrity of the data. And to achieve availability, we need the above two. We discuss how to solve the integrity problem in Section 4.2

If we want a system to compete with centralized storage systems, we need to be able to provide guarantees about the data stored on the network. A lot of the popular centralized storage systems provide data durability and availability of five (99.999%) or more nines.

1.5 HYPOTHESIS

In this work we want to answer the following questions:

1. How effective is PoR in addressing the integrity problem in decentralized networks?
2. Is a reputation system based on a ledger a viable measure against malicious nodes?
3. How does the validation system affect the performance of the network?
4. Does the validation system affect the other features of the system (performance, security, etc.)?

Of the requirements, Scalability, Performance, Resilience, Reliability, Availability, and Security are covered by other works. We need to evaluate the Integrity requirement, which will be done by evaluating the verification/auditing mechanism, the penalties and rewards for the nodes, and the reputation system. Implementing the validation system could make the performance degrade, so we have to evaluate if any of the requirements are affected by the validation system, and if so, how much. We will discuss the details and the results of the evaluation in 6.

We have implemented a version of the system, which is available at https://github.com/luchev/kiss.

2 RELATED WORK

2.1 DISTRIBUTED STORAGE SYSTEMS

Distributed storage comes in two flavors — centralized and decentralized. Centralized systems are mostly proprietary and are either used by companies or are offered as a service. Decentralized systems are mostly open-source and are used by academia, individuals, and occasionally companies.

There are many decentralized storage systems that have been developed over the years. While there exist decentralized networks, which are focusing on decentralized computing, most of the decentralized networks are used for storage or some form of data sharing. There are different kinds of decentralized storage:

1. **Unstructured networks** — nodes are connected to each other without any specific topology.
2. **Decentralized Hash Tables** — key-value store where the keys are distributed among the nodes.
3. **Blockchain based storage** — data is stored in a blockchain, or cryptocurrency is rewarded to users who share their available storage.

We will mainly focus on decentralized hash tables, as they are the most popular and are used by most of the decentralized storage systems. Unstructured networks are used for general purpose storage, but are not as efficient as Blockchain based storage is niche and is not used for general purpose storage.

2.1.1 *Security*

When we talk about security in decentralized systems we assume that the underlying network provides no security guarantees. Attackers can eavesdrop on the communication between nodes, modify the messages, and even drop them. They can also spoof IP addresses and there is no authentication of data packets in the underlying network. This is a reasonable assumption as most decentralized networks are built on top of the Internet, which is inherently insecure.

S/Kademlia [19] summarizes the main attacks on decentralized networks:

1. **Eclipse attacks** — an attacker isolates a node from the rest of the network.
2. **Sybil attacks** — a single entity pretends to be multiple entities.
3. **Churn attacks** — an attacker joins and leaves the network repeatedly.

4. **Adversarial routing** — an attacker returns adversarial routing information.

5. **Denial-of-service attacks** — an attacker floods the network with requests.

6. **Storage attacks** — an attacker manipulates the data stored on a node.

Kademlia and S/Kademlia cover most of these attacks and are mostly secure against them. The one exception is storage attacks. No decentralized network has a solution for storage attacks.

2.1.2 *Categorizing Networks*

There are many ways to categorize networks, and we will cover some of them here.

1. **Structured/Unstructured** — Networks can have a predefined topology, or they can be constructed at random. A downside of unstructured networks is often the discovery of resources. Since a resource can be at any node, typically such networks broadcast queries through the whole network. This is both inefficient and can also cause traffic congestion. In modern structured networks the query time is $O(\log n)$, where n is the number of nodes in the network.

2. **Keyspace** — the size of the keyspace can vary between different networks. Most networks use a keyspace of the form 2^b, where b is the number of bits. Each node in the network is assigned a peer ID, which is a random number in the keyspace. A good distribution of the peer IDs is important for the performance of the network, therefore the peer IDs are usually generated using a cryptographic hash function such as SHA-256.

3. **Naming Scheme** — how the files are named in the network. When storing files, an ID is generated for the file, which determines its location in the network. Usually the name of the file or the contents of the file are hashed to produce the key. More rarely, the key of the file is chosen by the user. While this allows for human-friendly names, it has a downside that it could be subject to dictionary attacks.

4. **Replication** — if and how the data is replicated in the network. Replication usually works by storing the file on the k closest nodes to the key of the file. More rarely replication is only achieved by caching the file at different nodes.

5. **Anonymity** — some networks are designed to provide anonymity to the users. Anonymity means that the identity of the user, the data they are storing or requesting, the queries, and the location of the data are hidden. Networks usually achieve this by using onion routing and cover traffic. Additionally, the network caches the files aggressively, so the original storage node is hard to trace. Anonymity usually means that the network cannot provide any

guarantees about the integrity of the data, as it should not be possible to be traced.

6. **Security** — what security guarantees the network provides Security is a major concern in decentralized networks because the network is open to anyone, and there is no central authority to enforce rules, in contrast to centralized storage systems. Peers can join and leave the network at any time, and they can lie about their identity and the data they store. We call such peers malicious.

   When we talk about security we assume that the underlying physical connection provides no security guarantees. Attackers can eavesdrop on the communication between nodes, modify the messages, and even drop them. They can also spoof IP addresses and there is no authentication of data packets in the underlying network. This is a reasonable assumption as most decentralized networks are built on top of the Internet, which is inherently insecure.

   Security can be defined as the ability of the network to detect and prevent attacks. S/Kademlia [19] summarizes the main attacks on decentralized networks:

   a) **Eclipse attacks** — an attacker isolates a node from the rest of the network.
   b) **Sybil attacks** — a single entity pretends to be multiple entities.
   c) **Churn attacks** — an attacker joins and leaves the network repeatedly.
   d) **Adversarial routing** — an attacker returns adversarial routing information.
   e) **Denial-of-service attacks** — an attacker floods the network with requests.
   f) **Storage attacks** — an attacker manipulates the data stored on a node.

7. **Persistence** — how long the data is stored in the network. One of the possible approaches is to let a file be available as long as there is interest in it, i.e., it is being requested. This can be expanded in a way that rare files or rare pieces are more valuable and networks that employ a bartering system can use this to incentivize peers to store them. Another approach is to republish the file every X amount of time, e.g., every 1 hour, to ensure that the file is still available. Networks that use this mechanism also use it to ensure that the data is stored on the correct nodes, in case a node goes offline, or a new, closer node, joins the network. Additionally, some networks split the files into chunks and replicate them separately. The most reliable networks make use of erasure codes to ensure that the file is still available even if some chunks are lost.

8. **Reputation system** — some networks have a reputation system to incentivize nodes to store data. In networks where the files are

split into pieces or blocks, storing rare pieces can increase a peer's reputation. Another approach is to use a ledger to keep track of the reputation (currency) of the peers, or have each node keep track of the reputation of its neighbors.

9. **Scalability** — how well the network scales with the number of nodes. Modern networks scale well with tens of thousands of nodes, the comparison mainly comes down to optimizing the constant factors. Networks often utilize locality or proximity to optimize the routing and speed.

10. **Fault Tolerance** — how well the network handles nodes failing or behaving maliciously. Fault tolerance is not addressed in many networks, as it is assumed that most nodes are honest. Common ways to deal with nodes failing are to use a republishing mechanism to ensure that the data remains available.

11. **Data Integrity** — if and how the network ensures that the data is stored correctly. Networks usually create a cryptographic hash of the file, which is used to verify the integrity upon retrieval. Most networks do not address data integrity before retrieval.

12. **Data Retrieval Efficiency** — how efficient the DHT is in retrieving data.

We will list some of the most popular decentralized storage systems and discuss how they fit into these categories. As most of the existing solutions focus only on a subset of these categories or make use of similar techniques, we will omit some of them for each system and focus only on the important and unique features that each system provides.

2.1.3 *DHTs*

Chord

Chord [5] is one of the first DHTs. It uses a ring-based topology and was used as the basis for many other DHTs, before Kademlia [12] became popular. Chord uses a 2^{160} keyspace, which is navigated using a modular arithmetic metric. Replication is optional and is achieved by storing the file on the k closest nodes to the key of the file. The keys are cryptographic hashes of the data. Upon joining the network, a node becomes responsible for the keyspace between itself and its predecessor. When a node leaves the network, its keyspace is assigned to its successor. The performance of Chord is $O(\log n)$ hops to reach the correct node (Figure 1), where n is the number of nodes in the network. The hops form a path, which essentially models the exponential search algorithm.

Pastry

Pastry [16] is a DHT that uses a ring topology. It provides only the routing algorithm, and implements no application-level functionality. It uses a 2^{128} keyspace, and nodes' IDs are either a cryptographic hash of their IP address or their public key. Pastry uses a more complex routing algorithm than Chord, by making use of a table with neighbors that

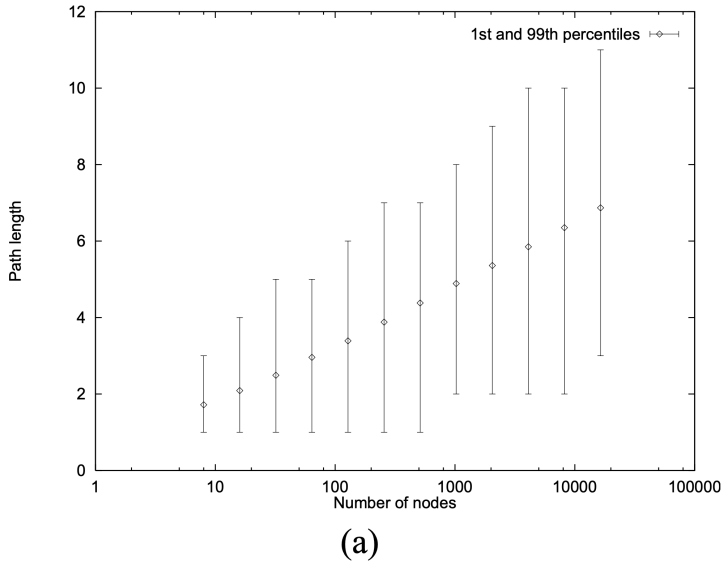

Figure 1: Chord performance [5], Figure 10 (a)

share a common prefix with the node. The routing algorithm utilizes prefix routing using the table. It also uses locality based on IP routing hops in order to optimize the routing and speed. A difference from Chord is how Pastry deals with node failures lazily. This means that when a node fails, the network will remove it from the routing tables of other nodes only when they try to access it. The paper explores a network with 5000 nodes where 10% (500) of the nodes fail. To repair such a failure 200,000 queries were required. Pastry does not address replication and recovery of data.

PAST

PAST [15] is a storage solution implemented on top of Pastry. A file is stored under a unique fieldID (key), which is generated at the time of insertion into the network. PAST does not support deletion, it provides a reclaim operation, which marks a file as "can be deleted", but does not force its deletion. In terms of security PAST assumes most nodes are honest and does not provide much protection against malicious nodes. It does address data integrity by using a cryptographic hash of the file, which is used to verify the integrity upon retrieval. The paper also briefly mentions random audits to see if the data is still available, but provides no details on how this would be implemented or what the effectiveness is. The integrity checks are done by utilizing a new concept that the paper introduces — smart cards. Smart cards have 2 parts — a private key, used to sign data and contracts and a balance section. The balance indicates how much data this peer is storing in the network versus how much it is storing locally. This balance is used to disallow freeloaders and deal with malicious nodes trying to flood the network with fake data.

Kademlia

Kademlia [12] is one of the most popular structured DHTs. It uses topology based on a binary tree, that organizes nodes based on their XOR distance. It is used as the basis for many other DHTs. Kademlia

uses a 2^{160} keyspace, which is navigated using a XOR metric. Replication is optional and is achieved by storing the file on the k closest nodes to the key of the file. The keys are usually hashes of the file contents. This enables a basic form of data integrity, as the file can be verified by comparing its hash to the key. However, this does not allow users to choose the key of the file. Kademlia does not provide anonymity to the users, as the network operates based on node IDs, which are public and can potentially be traced back to individual users. Stored data in the network needs to be republished every 1 hour to ensure that the file is still available, as records expire after 24 hours. The base Kademlia does not employ any security mechanisms, unless accidental such. For example, Eclipse attacks and Churn attacks are mitigated by the fact that the network favors longer-lived nodes over new ones. The network uses the republishing mechanism to ensure that the data remains available when nodes leave or join the network. The fault tolerance relies on the above mechanism, as well as the fact that the network favors longer-lived nodes over new ones. Queries in Kademlia take $O(\log n)$ hops to reach the correct node, where n is the number of nodes in the network.

S/Kademlia

S/Kademlia [19] is a secure version of Kademlia. It is the security extension of Kademlia and addresses the main attacks on decentralized networks.

To address Sybil attacks S/Kademlia employs a proof-of-work mechanism — a cryptographic puzzle that the joining node must solve. It also proposes a solution for adversarial routing by sending multiple queries to different parts of the network and comparing the results. The routing algorithm is also improved to be more resistant to adversarial routing by running multiple disjoint queries in parallel and comparing the results (Figure 2). The paper does not cover data storage attacks. DDoS attacks are not covered by the paper. These are usually attempted to be solved by rate limiting, throttling, and traffic filtering, however there are no guarantees that these will work against a sufficiently large attack.

Freenet

Freenet [9] is a DHT that focuses heavily on anonymity. Another classic example of a network that provides no guarantees about the longevity of the data. It uses replication by aggressive caching during retrieval to hide where the data is stored originally. It also anonymizes the queries and the responses by using onion routing-style techniques. However, this means that the network heavily relies on peers being honest. The keyspace is 2^{160}. The node ID is the XOR of the seeds of a number of nodes. Which means that the ID will be truly random, however it might allow a form of DoS attack where malicious nodes join the network in bulk, flooding it with traffic. Files are stored under hashes of the public key of a user-chosen string (description). The private key is used to sign the file as a basic form of data integrity. This enables dictionary attacks and key-squatting, which is a natural downside of a system that focuses on anonymity. Interesting new concept is the support of

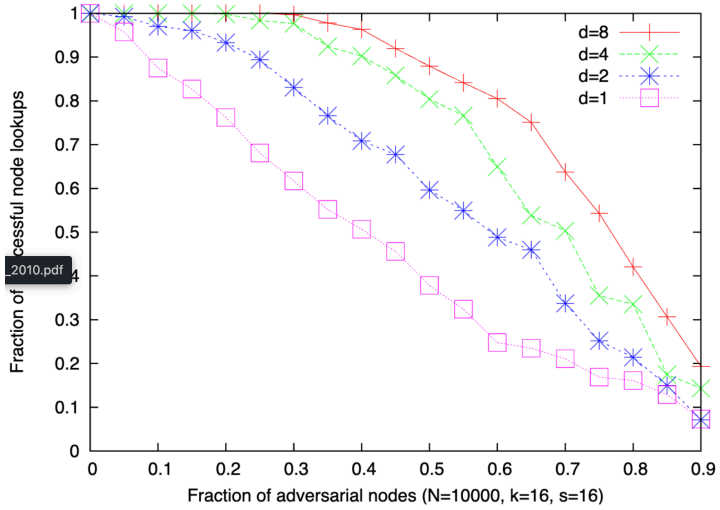

Figure 2: S/Kademlia node lookup success with d disjoint paths and k neighbor bucket size [19], Figure 4 (a)

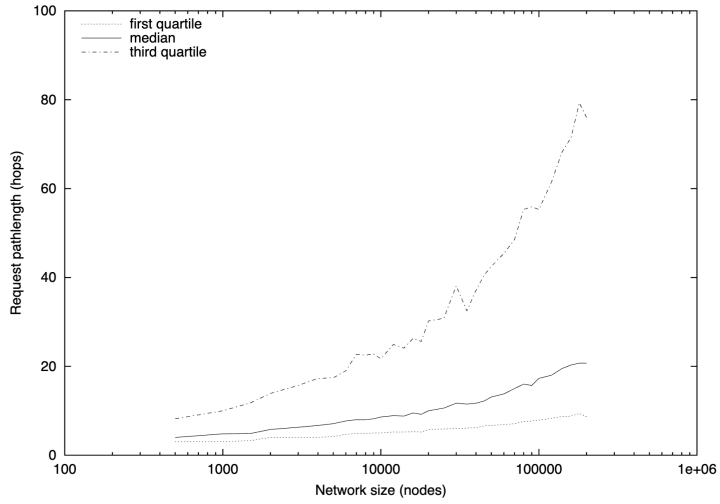

Figure 3: Freenet uses a hybrid routing algorithm that is between structured and unstructured, which results in higher number of hops, and hence lower performance [9], Figure 3

pointers, which allow the user to store a file under a key, and then store other keys that simple point to the same file. It uses a custom routing algorithm, whose performance can be seen in Figure 3.

2.1.4 *Large scale storage networks*

IPFS (InterPlanetary File System)

IPFS is a DHT that provides human-friendly names for files. However, if a user wants their files to be verifiable, they can create a personal namespace (a subdirectory), which is their public key, and store files under it. In this case the name is no longer human-friendly, but the files are verifiable. Human-friendly names have a downside that they could be subject to a dictionary attack and squatting — where a malicious user could register a name similar to a popular one and serve malicious content. IPFS makes use of Merkle DAGs in order to uniquely identify files and to ensure tamper resistance, as files can be verified by comparing their hash to its checksum. The system doesn't provide

persistence guarantees. It relies on popular files being requested often enough, so they can be cached by different nodes.

IPFS [10] uses the BitSwap protocol, which allows peers to exchange partial blocks of data, similarly to BitTorrent. Peers can barter for the blocks they need, or choose to store rare pieces of data to increase their reputation. The authors hint at using a ledger to keep track of the reputation (currency) of the peers, but leave it as future work. The BitSwap protocol borrows a lot from BitTorrent, and acknowledges that there are exploits such as BitTyrant [2] and BitThief [2] that can be used to exploit the protocol. Preventing these exploits however are left as future work.

OceanStore

OceanStore [14] another storage network aiming at scaling to serve trillions of files. To achieve this, the paper proposes a two-tiered query algorithm. First, a probabilistic fast query is used, based on bloom filters, and if it fails, a slower hierarchy based query is issued. In terms of security the paper discusses only DDoS attacks. It proposes hashing the file ID with multiple salts to generate different keys, which are then used to store the file on different nodes. This distributes the file across the network differently than sequential replication. To ensure data integrity OceanStore splits the files into chunks and stores them on different nodes. It also uses erasure codes so that the file can be restored from any k out of n chunks. Another network that is very similar and dives deeper into erasure codes is Tahoe [23].

2.1.5 *Blockchain-based networks*

Sia

Sia [20] is a blockchain-based storage network. The network stores storage contracts in a blockchain, while the data itself is stored by the peers. Sia aknowledges the need for auditing and that it is better for the network itself to audit the files, instead of requiring the clients to do so. In terms of replication, the network has no in-built such — it proposes the users to upload the same file multiple times. When creating a contract the client specifies how often the audits happen, as well as the price paid out for a successful audit. Contracts also have an upper limit of failed audits, after which the contract is terminated. Sia utilizes merkle trees to perform the audits, which results are then sent to the blockckain so they can be verified by anyone. The paper also proposes clients to use erasure codes, but they do not come built-in. To prevent Sybil attacks, the network requires peers to lock in coins when advertising themselves as available to store data. This way, it becomes expensive for peers to join en masse and it requires them to make a commitment to the network. Balancing rewards and punishments is left to the client to decide, as they get to propse the contract as well as choose which peer to store it at. While the paper proposes a lot of innovative ideas, it does not fully explore most of them and it doesn't evaluate them under what paramemters they make sense.

Filecoin

Filecoin [8] is another blockchain-based storage network, which is a direct upgrade of Sia and works on top of IPFS. Filecoin is described as a decentralized storage network that turns cloud storage into an algorithmic market. It also introduces its own cryptocurrency, which powers the transactions in the market. Coins are earned by storage nodes, that store the data, and by retrieval nodes, that serve the data, and they are paid by the clients. The way tokens are awarded to the nodes is in micropayments done by the client after a successful audit/retreival. The paper defines Data Integrity as the ability to prove that retrieved data is untampered with, and Retrievability as the ability to prove that the data is still available. There is a catch in the definition of Retrievability — it works under the assumption that the peers follow the protocol. It does not address the case where the peers are malicious and stop serving the data. This is the first instance where Proof of Retrievability is mentioned in the context of decentralized storage, however the paper does not use the modern PoR protocols, and assumes PoR is probabilistic and proves that a server is storing some part of the data, but not necessarily all of it. For the audits Filecoin uses a custom protocol named Proof of Replication, which is a variant of PoR. In simple words, Proof of Replication works by storing each replica of a file encrypted with a different key, which makes each file different, it then issues cryptographic challenges as audits. This prevents Sybil attacks, where the malicious peers create multiple identities but store the same data in 1 place. To build on top of it also makes use of Proof of Spacetime, which is a protocol that proves that a node has been storing data for a certain amount of time.

2.1.6 *Other networks*

ARA

ARA is not a storage network, but proposes ways to deal with free-riding in peer-to-peer networks. The paper assumes a specific kind of malicious peer, that is selfish, i.e., does not want to contribute their resources to the system. The paper proposes a decentralized economy system, similar to performing micro-paymets to a server. The introduction of an economy means the possible subset of attacks is mainly related to disrupting said economy. To deal with these attacks the paper proposes audits on the transactions and the credits of each peer. The audits rely on cryptographic signatures and at least one non-malicious peer that can perform the audit. The audit is essentially checking the claim of the malicious peer for how much data they store versus how much has been recorded that they should store in the system's ledger. The paper does not cover content cheating (file integrity) as they argue it is not feasible without human intervention.

2.1.7 *Summary*

Decentralized networks have evolved over the years with each new network building on top of the previous ones, by adding a new fea-

ture such as anonymity, security, a new routing algorithm, or introduction of a new technology such as blockchain. With networks becoming so complex, they tend to focus on a particular problem, for example anonymity or security. This is also due to the fact that the solutions to some problems are mutually exclusive, for example, anonymity and security. The tendency in the security direction is to address as many attacks as possible, with the latest networks making use of ledger stores and reputation systems.

2.2 PROOF OF RETRIEVABILITY

Proof of Retrievability (PoR) [18] is a cryptographic protocol that allows a client to verify that a server is storing a file. The server is challenged to prove that it is storing the file, and the client can verify the proof. PoR protocols are designed to be used to check the integrity of the data stored by cloud providers mostly.

2.2.1 *PoR predecessors*

Before PoR became an interest and a field of research, there were other methods to ensure data integrity.

The most basic method is to send the whole file to the client, which is inefficient and bandwidth-consuming.

Another method is to send a hash of the file, which is more efficient, but easy to attack since the server can store the hash of the file instead of the file itself. To address this, the client could send a challenge in the form of a salt to the server, which would then hash the salt with the file and return the hash. This is inefficient because the client has to preprocess the file and store salts for each file.

It is also possible to use a Merkle tree, which is a binary tree of hashes. The leaves of the tree are the hashes of the file chunks, and the internal nodes are the hashes of their children. The root of the tree is the hash of the file. The client can request a proof of a chunk by requesting the sibling nodes of the path from the leaf to the root. This solution also requires the client to precompute challenges with salts and store them.

Merkle trees are used by modern PoR protocols [7] to build upon. They are used mainly to enable updates to the file or when the file is split into chunks.

2.2.2 *First PoR protocols*

The first paper that introduced PoR was "Proofs of Retrievability: Theory and Implementation" [18]. It is focused on providing a PoR protocol to be used in a cloud environment, where the client can verify that the cloud provider is storing the file. This is an assumption a lot of the following papers on PoR make, as it provides good motivation and limitations for the protocol. We want to use little bandwidth, little processing power on the server and client side, and we want the server to be unable to cheat the client. The protocol proposed in the paper is probabilistic and checks only certain parts of the file. The protocol also includes encoding of the file in advance, as well as error correction

codes which allow the client to reconstruct the file if only some parts are corrupted.

2.2.3 *Modern PoR*

The later PoR protocols check the whole file and rely on cryptographic theory, which makes it close to impossible for the server to cheat the client with more than a negligible probability. The papers [6, 7] also start exploring the idea of public auditability, where the file can be audited without having a shared secret. In other words, the client does not need to store a secret in order to perform audits. All following papers also optimize different parts of the protocol, such as reducing bandwidth, reducing client storage, reducing algorithm complexity, and improving the security of the protocol.

The latest non-public PoR protocols provide down to $O(1)$ client storage, $O(N)$ server storage, $O(logN)$ client computation, $O(N)$ server computation, and $O(logN)$ bandwidth.

There are 2 state of the art PoR techniques:

- From 2020, "Dynamic proofs of retrievability with low server storage" [6] — based on an algebraic approach, involving matrix multiplication.
- From 2022, "Efficient Dynamic Proof of Retrievability for Cold Storage" [7] — based on a purely cryptographic approach with Merkle trees.

"Dynamic proofs of retrievability with low server storage" introduces a protocol, which requires $N + O(N/\log N)$ server storage, where N is the size of the file. The protocol is dynamic, meaning that the client can update the file and the server can update the proof. This feature could be important for some decentralized storage systems, which split the files into chunks and store them on different nodes. However, splitting the file between multiple nodes will make the audit process more complex and time-consuming. The client stores $O(\sqrt{N})$ data. This is still not perfect, as in a decentralized storage system each node is a client, hence knows the secret. This can be addressed with the last contribution of the paper — public verifiability. Public verifiability is a variant of the proposed protocol that allows any untrusted third party to conduct audits without a shared secret.

"Efficient Dynamic Proof of Retrievability for Cold Storage" makes improvements to the bandwidth and client storage requirements of the previous protocol. The audit proof is $O(1)$ — a fixed number of group elements sent from the server to the client. Client storage is reduced to a single master key, the size of the security parameter, which is again constant. Public verifiability is also possible with logarithmic overhead.

Proofs of Retrievability are an important instrument to ensure integrity in decentralized storage.

2.3 Overview of Cloud Storage

Cloud storage provides a way to store and access data over the Internet. The user does not need to worry about the physical location of the

storage, the hardware, the maintenance, the replication, the backups, etc., as it is abstracted away by the cloud storage provider. If we want to build a decentralized storage system, we need to provide similar guarantees and features as centralized storage systems. For this purpose we will explore the requirements for Amazon's S3 [1], which is one of the most popular cloud storage providers.

1. The system can store up to 5 terabytes of data in a single object.
2. The system can store an unlimited number of objects.
3. The system can store an object for an unlimited amount of time.
4. The cloud provider will not access the data for any purpose, except when required to do so by law.
5. Files will be stored under keys, which are unique identifiers and can be constructed to mimic a hierarchical structure.
6. The system will provide a simple REST API to store and retrieve the objects.
7. Files stored by the user can be accessed only by the user, unless the user decides to share the file.
8. Data is stored encrypted, and the user can provide their own encryption key.
9. The system can provide 99.999999999% durability of the data (achieved via verification).
10. The system can provide 99.99% availability.
11. The system can provide 99.99% of the time the objects will be retrieved in less than 100 milliseconds.
12. Deletes are guaranteed to be permanent, and no data can be recovered after a deletion.
13. The system provides different types of replication — region and cross-region.

These features come at a cost, which is usually a fee for the storage and the bandwidth.

If we want to build a decentralized storage system, we need to provide the similar guarantees and features. While we cannot restrict which users can access the data, we can encrypt it, which will make it unreadable to anyone who does not have the decryption key. There is one feature that is hard to achieve in a decentralized network — deletion. Because the data is stored on multiple nodes, and we cannot guarantee that all of them will delete the file, but since the data is encrypted, it is not readable to anyone who does not have the decryption key.

Most of the features are covered by existing decentralized storage systems, but the one that is not is durability. Durability or data integrity is the main focus of this thesis. We will approach the problem by using the same method as centralized storage systems — verification. Before we discuss the verification method, we will briefly talk about decentralized networks and security, in order to provide the necessary background for the rest of the thesis.

3

PRIOR WORK

In this chapter, we discuss the prior work that has been done before this thesis.

The system has undergone two major iterations.

3.1 VERSION 1

The goals for Version 1 were to implement a proof of concept of a decentralized storage system using a centralized form of verification/auditing. With this we aimed to demonstrate the feasibility of using a verification mechanism to ensure the integrity of the stored data.

In version 1, the core of the system was implemented in Rust using the p2p crate *libp2p*. We will not go into implementation details, but some notable points, which determine how the architecture of the system was designed, are worth mentioning. During startup the node reads the local configuration file, which contains the node's identity, bootstrap nodes, and any module specific configuration. Based on the module configuration, different modules are dynamically loaded into the node. For example, the storage module would load the Kademlia storage module in Version 1, but in Version 2 it would load the custom implemented storage module.

The core functionalities included the base of a decentralized storage system, implemented on top of the Kademlia DHT. We implemented the system as two separate peers — Storage and Auditor nodes. The Storage nodes (Keepers) implemented the Kademlia communication and stored the data in the default Kademlia storage. The Auditor node (Verifier) was implemented as a centralized server that would process storage requests, as well as perform audits on the Keepers The Verifier had access to a ledger store — ImmuDB, which was used as an index of the stored files and necessary metadata. During the audit, the Verifier would request the hash of the file from the Keeper and compare it with the file hash stored in the ledger. The communication between the Keepers and the Verifier was done over gRPC. The architecture is shown in Figure 4.

3.2 VERSION 2

Version 2 aimed to work towards decentralizing the verification mechanism. However, the problem turned out to be more complex than initially anticipated, which resulted in decentralized verification, but one that is easily attacked. Luckily during the development of Version 2, we reworked the verification to use a Proof of Retrievability (PoR) scheme, which allowed us to address the attacks against the decentralized verification mechanism in the future.

In version 2, major flows were addressed. The storage module was replaced with a custom implementation, which made use of the local disk using an S3-compatible API. The Verifier was decentralized, so it

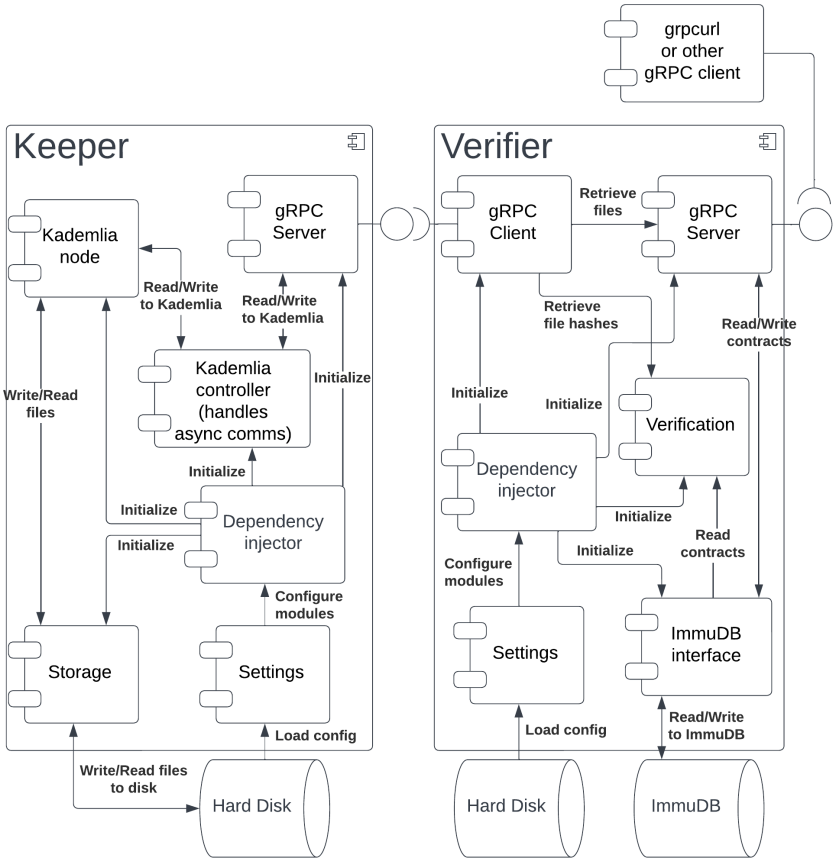

Figure 4: Architecture of the system in Version 1

was no longer a single point of failure. The Verifier and Keeper nodes were merged into a single node as two modules, which would perform both storage and audit operations as seen in Figure 5. In doing so, the communication between the Verifier and the Keeper was rewritten to happen over Kademlia. A new auditing mechanism was introduced, which was based on a Proof of Retrievability (PoR) scheme. After introducing PoR, we had to rewrite the storage and audit logic to support it as seen in Figure 6 and Figure 7. Finally, some attacks were addressed, such as the Sybil attack and the Eclipse attack by introducing a requirement to solve a cryptographic puzzle upon joining the network.

3.3 SUMMARY

This project started with the aim of creating a fully decentralized storage system with a verification mechanism to address storage integrity attacks. In the first iteration we implemented a centralized verifier as a proof of concept, which we later tried to decentralize only to find the problem to be more complex than initially anticipated. Decentralizing the verification lead to making the system vulnerable against other kinds of attacks. However, in the second iteration we improved the verification mechanism by introducing a Proof of Retrievability (PoR) scheme, which enabled us to address said attacks in the future. Our next goal is to implement a defensive mechanism against the attacks we identified in Version 2 by making use of PoR and to evaluate the system's performance and security.

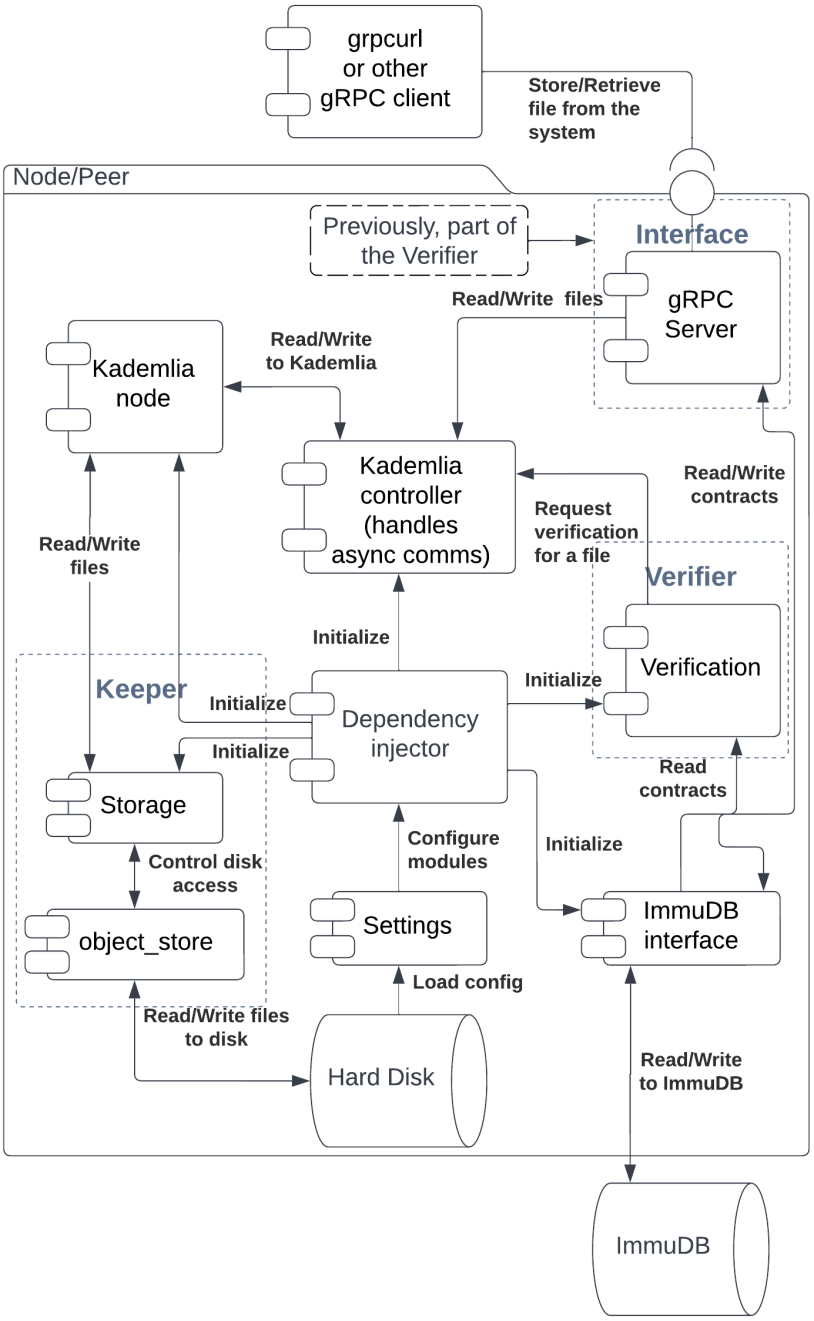

Figure 5: Architecture of the system in Version 2

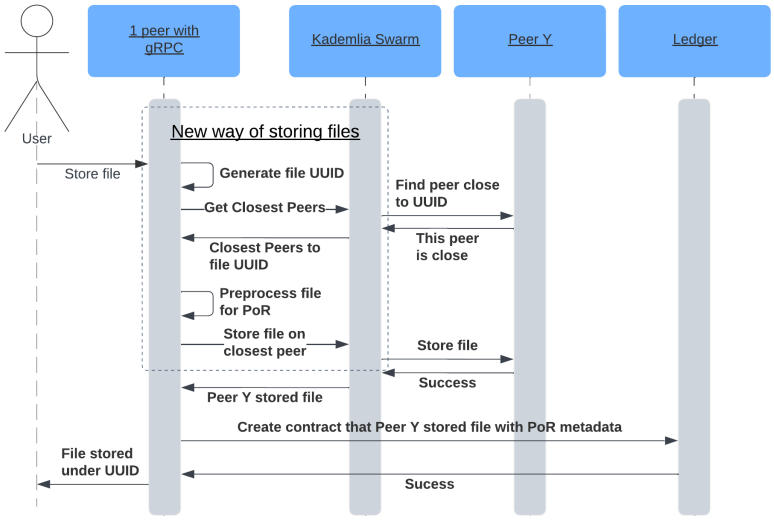

Figure 6: Storage flow in Version 2

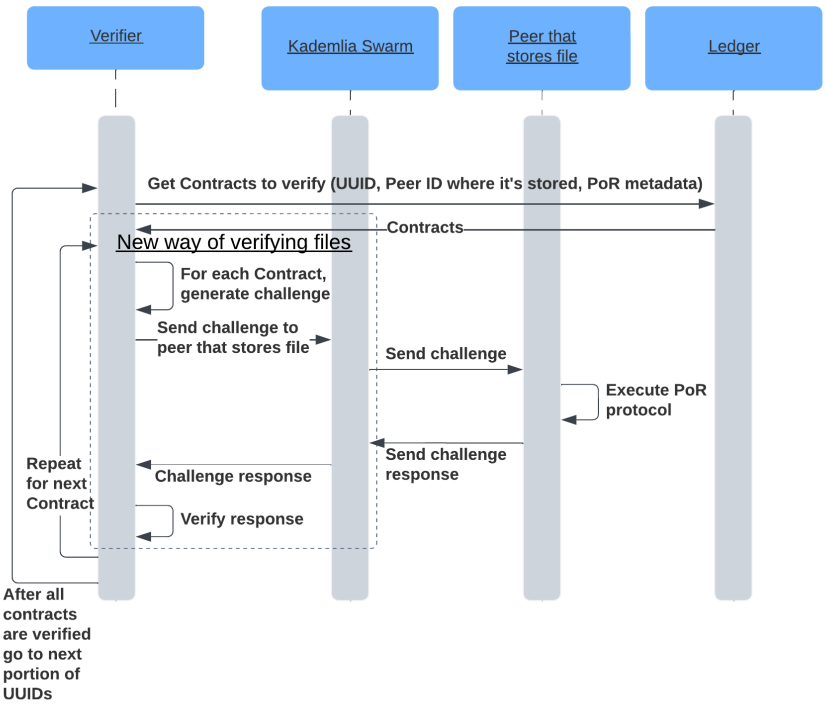

Figure 7: Audit flow in Version 2

4

ANALYSIS

In Chapter 3, we discussed how the project started as a decentralized storage system with verification and then evolved into integrating PoR into the verification, as soon as limitations of the decentralized verification mechanism became apparent. In this chapter we discuss how we are solving the problem of storage attacks and how PoR integrates into the reputation system. It is worth noting that apart from malicious peers, it is also possible for well-behaved peers to lose data due to hardware failures and bit rot. We do not cover this case as a special case, because it is a light version of what malicious peers are capable of doing.

4.1 MALICIOUS PEERS

There could be many types of malicious peers in a decentralized network, but we cover only the relevant ones for the scope of the project. The Related Work Chapter 2 covers other systems and how they handle the other types of attacks, not covered here.

There are two types of scenarios that can affect the integrity of the data in the network:

1. **Data availability degradation** — an attacker claims to store data, but does not.
2. **Data integrity degradation** — an attacker claims to store data, but stores different data, or data that has become corrupted without the peer being necessarily malicious.

While the two seem similar, we cover them separately, because they are performed by different types of attackers and the detection mechanisms are different. We would like to investigate the different behaviors of peers and how to deal with them. How severe should the punishment for each type of attack be?

Data availability degradation could be caused by the following peer behaviors:

1. **Not storing the data** — the peer does not store the data at all.
2. **Peer goes offline permanently** — the peer goes offline at some point in the future after accepting to store the data.
3. **Peer goes offline temporarily** — the peer goes offline for a short period of time. This could be a peer that is restarting or lost connection to the network. This is not necessarily a malicious peer, but it should be accounted for, because malicious peers could be performing an eclipse attack.
4. **The peer is slow to respond** — the peer is slow to respond to requests. This could happen when a peer is overloaded or has a slow connection. It could also be a malicious peer that is not storing the data and tries to request it from another peer in the network, before answering the request.

Data integrity degradation could be caused by the following peer behaviors:

1. **Storing corrupted data** — the peer stores corrupted data. It could be that the peer is trying to save space by storing partial data or is trying to disrupt the network. Or perhaps the data has become corrupt over time without the peer being malicious.

2. **Storing corrupted metadata** — the peer stores the correct data, but the metadata is corrupted. This could mean storing the wrong expiry date or the wrong key. This kind of behavior either wouldn't affect the operation of the network or it would fall under the other types of behaviors. For example, if the peer is storing the wrong key, it would be the same as not storing the data, as querying for the original key would result in no response.

The impact of all of these behaviors on the network is always dependent on the time it takes to detect them. If the time to verify a piece of data is too long, a malicious peer could delete it and then request it from another peer in the network just before the audit. Thereby passing the audit, but not contributing to the network. While this is a possible attack, it is an expensive one to execute, because the attacker needs to perform a lot of work to pass the audit. The attacker also needs to contribute considerable amount of their bandwidth to execute such an attack. This is precisely the reason why we want lightweight audits for the auditor and heavier audits for the auditee. For the above reasons, we do not consider this attack in our analysis.

4.2 SOLVING THE INTEGRITY PROBLEM

To solve the integrity problem, we need to be able to check if the data is stored correctly. To achieve this we need a verification/auditing mechanism, which is efficient and secure. We will use a Proof of Retrievability (PoR) scheme, which will allow us to efficiently check the integrity of the data stored in the network. PoR schemes are a type of cryptographic scheme that allows a verifier to check if a node stores the data it claims to store. The verifier does not need to download the whole file, but only exchange a few messages, which are much smaller.

Once we have a method to check the integrity of the data, we need someone to perform this check. We will use the nodes in the network to perform the checks. It makes sense for honest nodes, which are users of the network, to do audits, as they have a vested interest in the integrity of the data. However, we also need to make sure that malicious nodes are also doing audits.

We can achieve this by using rewards and penalties. If a node is found to be storing data incorrectly (by failing an audit), we will penalize the node. If a node is found to not be performing audits, we will penalize it. On the other hand, if a node is storing data correctly (by passing an audit), we will reward the node. And if a node is performing audits, we will reward it.

One way to penalize nodes is to remove them from the network. However, it would be better if we could have lighter penalties. Here we are proposing a reputation system based on a ledger store, e.g., a blockchain. We will use a blockchain to store the reputation of the

nodes. This will allow us to have a light penalty system, as we can downgrade the reputation of the node instead of removing it from the network. Using a ledger store allows us to have a transparent and immutable record of the reputation of the nodes.

Using a reputation system will also allow us to reward nodes for performing audits. Successfully storing a file and passing audits will increase the reputation of the node. Successfully performing audits will also increase the reputation of the node. Failing to store a file or failing to perform an audit will decrease the reputation of the node. In the 1.5 we will discuss how very high or low reputation nodes will be treated.

We know how to perform audits, but we also need to know how to make sure none of the audits are faked. To solve this, we would require the nodes performing an audit to record the results of the audit and keep it in a ledger store. When a node is suspicious of the correctness of the audit, it can check the ledger store to see if the audit was performed correctly. This could also be used to check if the audits are being performed at all.

4.3 Using proof of retrievability (PoR) to prevent attacks

Preventing a Data availability attack can be done by querying the peer whether it has the data. In this case we assume that the peer is honest and always returns a simple answer whether it has the data or not. The assumption that a peer is honest is naive, but allows us to easily check if the peer is responsive. One of the reasons we want to separate this attack is, so we can analyze peers going offline separately. We will discuss how to handle temporary and permanent offline peers in the evaluation section.

Preventing a Data integrity attack can be done by requesting the peer to return the data and checking if the data is correct. This is a costly operation, because the data could be large, and the network could be slow. This is a stronger requirement for the network, because peers performing this attack are a superset of the peers performing the data availability attack.

In order to address data integrity attacks, we make use of Proof of Retrievability (PoR). PoR is a cryptographic protocol that allows a peer to prove to a verifier that it is storing a file. The verifier does not need to know the file, but only a small secret that is generated from the file. The verifier can then challenge the peer to prove that it is storing the file by sending a challenge and comparing the response to an expected response. Both the challenge and the response are small. The exact size depends on the PoR protocol used, and in our case is $O(\sqrt{N})$ [6] for both the challenge and the response, where N is the size of the file in bytes.

Since the PoR protocol is cheap to execute both on the network and on the computation side of the verifier, it can be used to address both types of attacks.

PoR has two versions — default and public. The default version is when the Verifier requires some form of secret to be able to run the audit. The public version is when the Verifier does not require any secret to run the audit. If the default version is used it is important to

not reveal the secret to the Keeper, because the Keeper could then use the secret to generate the response without storing the data. Ideally we want to use the public version of the PoR protocol whenever possible.

In Section 5.10 we discuss how the PoR protocol can be used in conjunction with a reputation system to keep track of the behavior of the peers in the network long-term. However, there is another approach that Filecoin [8] uses, which is to use Proof of Spacetime. Proof of Spacetime runs the verification protocol on regular occasions and uses chained hashes or signatures to combine the old proofs into a single proof.

4.4 REPUTATION SYSTEM

To keep track of the behavior of the peers in the network, we use a reputation system. In short, we want well-behaved peers to have a high reputation and malicious peers to have a low reputation. Also, well performing peers should have a higher reputation than poorly performing peers. e.g., a peer that responds slowly or fails audits because their hard drive malfunctioned should have a lower reputation than a peer that is always online and responds quickly to requests. While this might seem punishing to the peer that has a malfunctioning hard drive, it is important to keep the network healthy and to prevent the peer from causing more damage to the network. Peers with higher reputation are prioritized by the network, while peers with lower reputation are avoided and potentially removed from the network. A key part of designing the reputation system is defining how much reputation is adjusted in each case. First, let us look into what situations lead to reputation adjustment, and then we can analyze how much reputation should be adjusted in each case.

Reputation is increased when a peer is behaving as expected, i.e., storing data and providing data, as well as performing audits, all lead to an increase in reputation.

Reputation should be decreased in the case of failure to comply with the network rules. A peer that is not storing data should have their reputation reduced on each audit, until their reputation is so low that they are not trusted anymore and are removed from the network. A peer that goes offline permanently would fall under the same category, as it would fail all its audits. A peer that goes offline temporarily should be punished for the failed audits, but the punishment should be such that for a short period of being offline the peer should not use too much of its reputation. When a peer is slow to respond to requests, the audit is considered failed and the peer has its reputation reduced as if it was not storing the data.

For the data integrity attacks, and in particular for storing corrupted data, the punishment should be the same as for the data availability attacks.

The questions we need to answer are:

1. How many audits can we perform in parallel?
2. What is the performance impact of the audits on the network?
3. How do we balance the punishment so that a peer that has been offline for a short period of time is not punished too much?

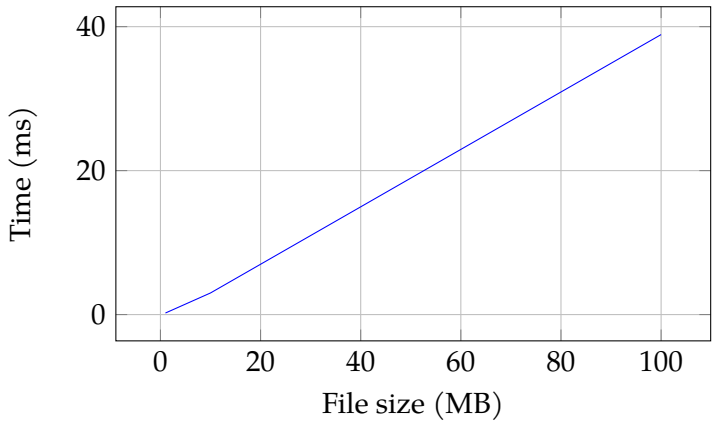

Figure 8: Time required to create the challenge response for PoR on the Keeper side, without reading the file from disk.

4. What is a short period of time to be offline?
5. Can we make the punishment scale with time? The longer the peer is offline, the more severe the punishment.
6. What does it mean for a peer to be slow to respond?

We have mostly answered the last question in our previous work, where we have evaluated the performance of the PoR scheme we have implemented. For small enough files, we cannot guarantee that the peer is storing the data and not just fetching it from another peer in the network. But to answer the question fully, we also need to account for the network latency. If we look at the ping times between London, UK and Christchurch, New Zealand, we can see that the average ping time is around 275ms [17]. These two cities give us a rough estimate of the maximum distance between two peers in the network, as they are on the opposite sides of the globe. These results are achieved under ideal conditions, and the actual ping time could be much higher. We propose 500ms as the maximum time a peer can take to respond to a request. This is almost 2 times the average ping between two very distant cities, so it should cover all cases. If we set such reasonable timeouts, we can be sure that the peer is not fetching the data from another storage, because cheap storage is excruciatingly slow.

This time is not accounting for the time it takes to process the request, but only the time it takes for the request to reach the peer. The time to process the request is the sum for reading the file from disk and executing the PoR protocol. The time to execute the PoR protocol is around 40ms for a 100 MB file as our benchmarks show in Figure 8. The time to read a 100 MB file from disk is around 14ms on an SSD and 200ms on an HDD, if we use the average read speed of 7000 MB/s for an SSD and 500 MB/s for an HDD. Assuming that most peers use HDD for cheap storage, we shall use the 200ms time to read a 100MB file from disk. Adding these up, we should expect a Keeper's response to take 500ms + 250ms for each 100 MB of data.

4.4.1 *Attacks on the reputation system*

One of the questions we want to answer in this thesis is if a reputation system based on a ledger store is secure. In other words, can we use

the ledger to store the reputation of the peers in the network and be sure that the reputation is not tampered with, and does introducing a reputation system based on a ledger store perhaps introduce new vulnerabilities to the system? To answer this question we need to talk about possible attacks on the reputation system.

Ranking peers in the network based on their reputation means that having higher reputation gives merits to a peer. Therefore, peers might try to game the system in order to increase their reputation.

We focus on the following attacks:

1. **Increasing reputation attack** — a peer tries to increase its reputation.
2. **Decreasing reputation attack** — a peer tries to decrease another peer's reputation.

Increasing reputation attack is when a peer tries to increase its reputation by cheating the system. The reputation is stored in a ledger, which means that any changes to the data are recorded. If a peer tries to increase its reputation, the other peers in the network can detect this by checking the ledger's history. This leaves the attacker one option - award itself reputation, i.e., a Sybil attack. Since reputation is only awarded for storing data and performing audits, the attacker could join the network with multiple identities and perform audits on itself. This wouldn't work, because audits are performed on a rotating basis, i.e., each cycle a peer audits only a certain subset of the network. If an attacker tries to audit itself not following the cycle, the other peers in the network can detect this, because the audits are stored in the ledger as well and can be checked by other peers in the network. This leads us to the situation where if a peer tries to perform a Sybil attack and cheats, it is detected by the network. If the peer tries to perform a Sybil attack and doesn't cheat, it simply contributes to the network as any other peer.

Decreasing reputation attack is when a peer tries to decrease another peer's reputation. We can apply the same reasoning as for the increasing reputation attack. Directly increasing another peer's reputation is not possible, because the ledger is immutable and keeps track of all the changes. The second way to decrease reputation is to perform audits on the peer and fail them. Since the audits' results are stored in the ledger, the other peers in the network can check the results, and see if the peer is being honest.

As a conclusion - while attacks are possible, they are easily detected by the network and do not pose a threat to the system as a whole.

4.5 AUDITS

Audits are a key part of the reputation system. They check if the peers are storing the data they claim to store. The audits are performed by the Verifier against the Keepers and the results are stored in the ledger. The audit process follows the PoR protocol, where the Verifier sends a challenge to the Keeper, and the Keeper responds with the proof, which the Verifier checks.

Verifiers perform audits on a rotating basis, i.e., each cycle a Verifier is responsible for auditing a subset of the files in the network. e.g., If we

have two Verifiers, in cycle one Verifier A audits the first half of the key space, and Verifier B audits the second half. In cycle two, they switch — Verifier A audits the second half and Verifier B audits the first half. The length of the cycle depends on how much data is in the system — this determines how much time it takes to audit all the files in the network.

Upon completing the audit of all the files, the Verifier stores the results in the ledger. The results are signed and timestamped by the Verifier to ensure they can be checked by other peers in the network. The result of an audit has a similar structure to the following JSON object:

```
{
    "keeper": "A",
    "file": "1.txt",
    "result": "success",
    "cycle": 1,
    "time": "2020-01-01T00:00:00Z"
}
```

Upon a successful audit the Keeper gains reputation points. Upon a failed audit the Keeper loses reputation points. The Verifier also gains reputation points for performing the audit.

These 3 changes in reputation are stored in the ledger and can be checked by other peers in the network. We discussed attacks where the Keeper aims to adjust its reputation in the previous section. We will now look into how we can stop the Verifier from misbehaving.

The Verifier can misbehave in the following ways:

1. **Not performing audits** — the Verifier claims to perform audits, but does not.
2. **Performing audits incorrectly** — the Verifier performs audits, but does not check the results correctly.
3. **Wrongly Accusing the Keeper of not storing the data** — the Verifier accuses the Keeper of not storing the data, but the Keeper is storing the data.

Not performing audits can be caught by inspecting the result of the audits, which is stored in the ledger. In other words, in a given cycle, when a Verifier is auditing a set of Keepers, it also audits the Verifiers that were previously responsible for auditing the Keepers. These audits are used to compare the results of the current Verifier with the results of the previous Verifiers, which can be found in the ledger. There are two ways to audit the previous Verifiers. One is to audit the Verifiers from the previous few cycles. And the second is to randomly sample the Verifiers to audit, i.e., randomly sample previous cycles to audit. Auditing the Verifiers from the previous few cycles is more secure short term as it ensures that the most recent audits were performed correctly. However, if we want to check old cycles as well, it becomes more expensive — for each previous cycle, we have to fetch the results from the ledger, compare them, and write the comparison results back to the ledger. Random sampling is cheaper to execute, but does not provide strict guarantees that the audits were performed correctly in the most recent cycles. If we have very few Verifiers in the network, we can use the first method, as the cycle length will be short and the cost

of auditing the previous Verifiers would be low. If we have many Verifiers in the network, we can use the second method, which will not provide strict guarantees, but will be cheaper to execute. Finally, we can combine the two methods to get the best of both worlds.

Performing audits incorrectly can be caught in the same way as not performing audits.

Wrongly Accusing the Keeper of not storing the data can be caught in the next cycle when the next Verifier audits the same Keeper. If the Keeper is not storing the data, the audit would fail, and the Keeper would have its reputation reduced. If the Keeper is storing the data, the audit would pass, and the Verifier that accused the Keeper would have its reputation reduced. In theory this can put the Keeper in a position of power, where it can fail the audit of the Verifier and in the following cycle pass the audit, falsely accusing the first Verifier. This is not a viable attack since the Keeper would need to sacrifice its reputation to perform the attack (on a failed audit the Keeper loses reputation). It becomes a game of who has more reputation to lose. At this point, if the malicious Keeper(s) have more reputation to lose than the Verifier(s), they would also have full control of the network, since with more reputation they would have the majority of the network's trust. While this is a possible attack, we will not consider it in our analysis, because it is a very expensive attack to perform, and we do not consider attacks where the malicious party has more than 50% of the reputation in the network or more than 50% of the nodes in the network.

4.5.1 *Conclusion*

In conclusion, if we want to introduce Verifiers and allow each peer in the system to be a Verifier, we need to ensure that the Verifiers are performing their duties correctly, similarly to how we ensure that the Keepers are storing the data correctly. In this case we cannot use PoR, but instead we can perform simple audits on the Verifiers, by comparing the other Verifiers' results with the current Verifier's results. We are assuming that the current Verifier is one that is honest and is performing the audits correctly.

The downside of this approach is that if there are more than 50% malicious Verifiers in the network, they can collude and cheat the system, so we require the majority of the Verifiers to be honest. In particular, it is possible for Keepers to attack the Verifiers by selecting a Verifier and failing its audits. On the following cycle, the Keeper would be audited by another Verifier, and it would pass the audit, resulting in the first Verifier losing reputation. While this is a possible attack, it can be prevented by introducing some form of onion routing. This would prevent the Keeper from knowing which Verifier is auditing it, which would make the attack meaningless.

4.6 LEDGER

The ledger is used as a catalog for the files stored in the network — it stores contracts, which state the key of the file, what peer is storing the file, and the expiry date of the file. The ledger also stores the reputation of the peers in the network.

In the previous section we discussed how the ledger is used to store the results of the audits. These results could be very large, if we have many files in the network. The most basic way to store the results is to store them as a list of JSON objects:

```
{
    "keeper": "A",
    "file": "1.txt",
    "result": "success",
    "cycle": 1,
}
```

To optimize the storage of the results, we can store only the failed audits. We can also only store the contract ID, which is a reference to the file being stored, and the peer ID where it is stored. Finally, we could store the results in the System itself and only store the signed hash of the results in the ledger. This is particularly useful if we want to use a blockchain as the ledger, because then we want to store as little data as possible in the blockchain, so it does not grow too large.

Ideally the ledger should be a blockchain as it is a form of distributed ledger, that all peers in the network can access and check.

4.6.1 *Blockchain as a ledger*

Typical blockchains store transactions of tokens, where an amount of the coin or token is transferred from one account to another. In our case we want to store the results of the audits, as well as the reputation of the peers in the network.

Let us discuss storing the reputation first. We want to treat the token of the blockchain as reputation points. When a peer gains reputation, it gains tokens, and when it loses reputation, it loses tokens. The wealth of a peer is the same as its reputation. We do not want to use the coin as a payment method, but as a form of trust. Blockchains such as Filecoin [8] use the coin as payment for storing data. We want to trade "trust". In other words, when we want to store a file in the network, we pay trust to the Keeper, increasing the trust of that Keeper if they store the file. Unlike typical blockchains where the coin is traded between users, we want the coin to be traded between peers. In essence, making the peers the real users of the blockchain.

4.7 Augmenting the PoR protocol with receipts (rejected idea)

One alternative idea to improve the audit process is to augment the PoR protocol with receipts. After an audit the Keeper would receive a receipt from the Verifier and would store this receipt locally. The receipt indicates that the audit was performed and the result of it. The receipt would also be signed by the Verifier. This way the Keeper can prove to other peers that the audit was performed in case the Verifier tries to accuse it of not storing the data.

The downside of this approach is that it increases the complexity of the system, but does not provide much benefit.

Let us look at the four possible scenarios. For each of them we shall assume the Verifier in the following cycle is honest and performs the audit correctly. If it is not honest, and lies about the result of the audit, then the next honest Verifier will catch it and the malicious Verifier will lose reputation. In other words it is okay to have a chain of malicious Verifiers, as long as it ends with an honest Verifier that will catch the malicious Verifiers (assuming the system is not fully consisting of malicious Verifiers).

Keeper has the data & audit succeeds — This is the happy path where the Keeper has the data and the Verifier writes in the ledger that the audit was successful. This is the expected good behavior of nodes in the system. The next Verifier in the next cycle (assuming honest) will also perform a successful audit.

Keeper does not have the data & audit fails — The other almost happy path. Both the Keeper and the Verifier are honest, but the Keeper has lost the data. The next Verifier in the next cycle will also perform an audit and see that the Keeper does not have the data. This is the expected behavior of the Verifiers in the system.

Keeper does not have the data & audit succeeds — This is the case where the Keeper does not have the data, but the Verifier writes in the ledger that the audit was successful. This is a case where the Verifier is malicious and tries to increase the Keeper's reputation. The next Verifier in the next cycle will also perform an audit and see that the Keeper does not have the data. Hence, it will reduce the first Verifier's reputation. This is the case where the malicious peers try to increase each other's reputation, however, if there are more honest peers in the network, they will lose more reputation than they give each other.

Keeper has the data & audit fails — This is the case where the Keeper has the data, but the Verifier writes in the ledger that the audit failed. This is the interesting case, that the receipt would aim to solve. The Verifier is malicious and tries to decrease the Keeper's reputation. In the next cycle, the next Verifier will also perform an audit and see that the Keeper has the data. Hence, it will reduce the first Verifier's reputation. In essence, the first Verifier will trade its reputation for the Keeper's reputation. This falls under the same category as the previous case, where the malicious peers try to increase each other's reputation,

5 IMPLEMENTATION

In this chapter we discuss the implementation of the system. Our main goal is answering the questions posed in Chapter 1. In particular — can we use PoR to solve the integrity problem, can the reputation system assist in solving this problem, and how does the validation impact the performance of the network?

In Chapter 4 we discussed how the system would work in an ideal world, and in this chapter we discuss how we have implemented the system — what shortcuts we have taken, what trade-offs we have made, how do they affect the system, and what we have learned from the implementation.

While answering the research questions is our main goal, we also want to design the system with the best software practices in mind.

The code is available at `https://github.com/luchev/kiss`.

5.1 ARCHITECTURE

The overall architecture has not changed much from the prior work. We have added a new module responsible for simulating malicious peer behavior, which uses the local storage in order to tamper with the files. The other modules have seen some refactoring and new additions, but the core architecture remains the same. The architecture of the system is shown in Figure 9.

5.2 MODULES AND DEPENDENCY INJECTION

The architecture of the system revolves around dynamically loading modules. We want to easily exchange the different implementations of the components in the system. For example, we want to change the storage layer from local storage to a Docker container, or for testing we want to quickly change the behavior of peers — whether it is malicious or not.

To achieve this we make use of the `config` crate to load the configuration. Configuration is hierarchical and is loaded from multiple sources. First we load the base configuration from a `.yaml` file. Then we load an optional configuration `.yaml` file that can override the base configuration. Finally, we load any environment variables that can override the configuration. In Listing 1 we show an example of a partial configuration file. We need partial configuration files because during testing we want to start each peer on a different port. It is useful to be able to change the `grpc.port` field or to be able to pass it as an environment variable `GRPC_PORT` because a lot of our end-to-end tests are bash scripts, which start hundreds of peers in parallel.

Listing 1: Example of a configuration file

```yaml
storage:
  type: local
```

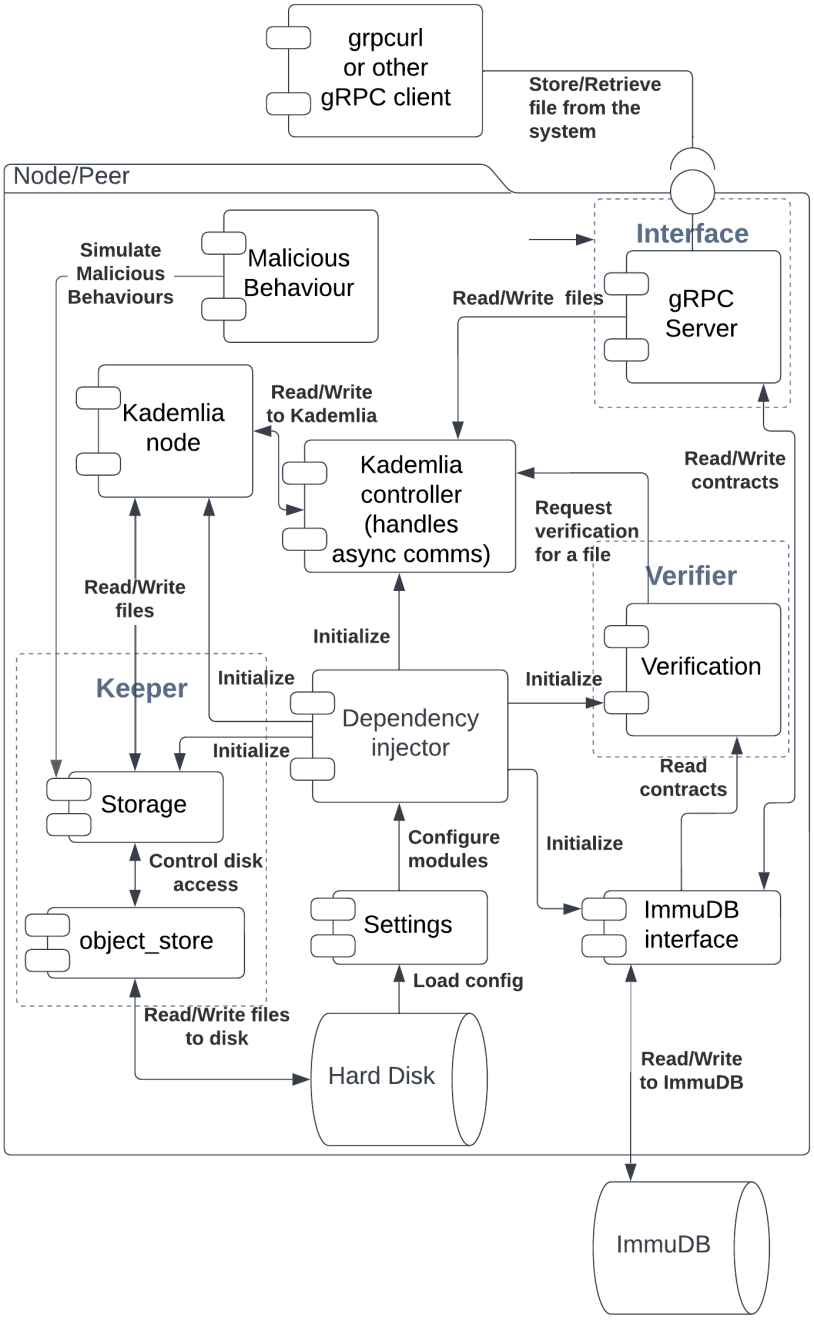

Figure 9: Updated architecture of the system. The `Malicious Behavior` module was added. The other modules had additions, changes, or refactoring done.

```
  path: data/peer1
grpc:
  port: 2000
malicious_behavior:
  type: none
```

Once the configuration is loaded we pass it to the dependency injector, for which we use the crate `runtime_injector`. It reads the configuration and instantiates the different modules based on it. For example when we initiate the malice, inject the implementation of the storage module and based on the settings in the configuration, we initialize a different malicious behavior, i.e., we provide a different implementation for the malice module. An example of how this works can be seen in Listing 2.

Listing 2: Example of dynamically loading a module

```
let settings = injector.get::<Svc<dyn ISettings>>()?.
    malicious_behavior();
let storage = injector.get::<Svc<dyn IStorage>>()?;

match settings {
    MaliciousBehavior::None =>
        Ok(Box::<MaliceNone>::default()),
    MaliciousBehavior::DeleteAll =>
        Ok(Box::new(MaliceDeleteAll::new(storage))),
    MaliciousBehavior::DeleteRandom(_) =>
        Ok(Box::new(<MaliceDeleteRandom>::new(storage))),
}
```

The dependency injector in combination with the configuration is a powerful tool, as it allows us to easily change the behavior of the system, without making code changes. It also allows us to develop the system in a modular way, where each module is independent of the others, and only depends on the interfaces of the other modules. The dependency injection crate handles converting each module to a singleton service, passing a reference to that service to the other modules that depend on it, and resolving the dependency graph. The one thing it does not handle is the case when each module runs in a different thread, and we need to ensure that the modules are thread safe. In this case we have to manually enclose the modules in a Mutex, which is not ideal, but it is a trade-off we are willing to make for the flexibility of the system.

5.3 LEDGER

The only component in the system we have implemented that is not distributed is the ledger. The ledger acts as the catalog for the files stored in the network, as well as the reputation of the peers. Ideally the ledger should be distributed in order for the system to be fully decentralized. An example of a distributed ledger technology is any blockchain. Initially we considered using one of Near, Solana, or Substrate [13, 21, 22]. These are all blockchains implemented in Rust, which is the language we are using for the implementation. This would have allowed us to easily integrate the ledger more easily with the rest of the system.

Entries	Workers	Batch	Batches	Time (s)	Entries/s
1M	20	1,000	50	1.06	1.2M/s
1M	50	1,000	20	0.54	1.8M/s
1M	100	1,000	10	0.62	1.6M/s

Table 1: Immudb performance according to the authors [24]

However, we decided against using a blockchain for the ledger for this implementation mainly because of the complexity of integrating with a blockchain. We see this as a future work in order to make the system production ready, and we discuss the integration with a blockchain in the Chapter 7.

5.4 Using the immutable database immudb

Immudb [11] is a ledger store that is used to store the results of the audits. We will be using it in the implementation as it is a simpler solution than using a blockchain.

An immutable database ensures we can store the metadata necessary to run the system in a tamper-proof way. The database acts as a form of catalog for the files stored in the network, storing the keys and metadata required to perform audits. Immudb is a lightweight, high-speed immutable database that is designed to be used as a key-value store. It is based on the Merkle tree data structure, which allows for efficient verification of the integrity of the data. The database is designed to be tamper-proof, meaning that although the database supports updates and deletes, the history of the data is always preserved. If a record is changed or deleted, the database keeps a record of the change, and the history of the record can be traced back to its creation. This property is crucial for the reputation system, as it ensures that the reputation scores are not tampered with, or if they are tampered with the client of the database can detect the tampering.

5.4.1 *Immudb performance*

Having a centralized ledger means we have a potential bottleneck. The benchmarks that the authors of Immudb are presented in Table 1. While processing more than a million requests per second is impressive, it is not performance we can expect because we are storing the metadata for PoR in the ledger, which is considerably more than empty requests. Based on the benchmarks we can conclude that we can use Immudb as a ledger and have a throughput of more than a million requests per second, but we need a distributed version of Immudb, because the limiting factor becomes the network throughput and the disk read/write speed. Immudb is not bad as a ledger, but the fact it is not distributed limits its use in a production system.

5.4.2 *Using Immudb at scale*

One downside of the database is that it is not innately designed to be distributed. This means that the database is not designed to be run on multiple nodes, and it does not support replication out of the box. In a real-world scenario, running the database on a single node is a huge flaw, as it makes the database a single point of failure. For the purposes of our experiments, we run the database on a single node. However, we will briefly discuss how the database can be made distributed in the future. The first approach is by distributing the Immudb itself. We could shard the data and run multiple instances of the database on different peers. Since the database is immutable, peers can be sure that the data is consistent across all the nodes. For example, sharding the data could be done by taking the file key, or the hash of the file key as the shard key. This way it would be easy to locate where the metadata for a given file is stored. The second approach is to use the storage system itself, completely removing the need for the database. This would require more work to ensure the system can support the same operations as the database, but it would simplify the querying, as the metadata would be stored in the same place as the files.

5.4.3 *Atomic Operations in Immudb*

The second downside of the database is that it does not support complex queries.

For example, we need to atomically update the reputation of a peer after a successful audit. Since audits may happen in parallel, we need to ensure that the reputation is updated correctly. This means that we need to read the current reputation of the peer, increment it, and write it back to the database. This operation needs to be atomic, as we do not want to lose any updates.

Listing 3: SQL query to update the reputation of a peer

```
1 BEGIN TRANSACTION;
2 DECLARE @value INT;
3 SELECT @value = reputation FROM reputations WHERE peer_id =
      @peer_id;
4 UPDATE reputations SET reputation = @value + 1 WHERE peer_id =
      @peer_id;
5 COMMIT TRANSACTION;
```

The above query would work in a traditional SQL database, but it would not work in Immudb, because Immudb supports only simple SELECT, INSERT, UPDATE, and DELETE operations. Luckily it does support transactions, which we can use to ensure that the operation is atomic.

Immudb supports two ways of authentication, both making use of a token. The first way is to log in as a temporary user, which enables the use of single SQL queries. For example, after a login, we can execute an INSERT followed by a SELECT followed by another INSERT. These queries are independent of each other. If we want to use transactions, we need to log in and establish a session, followed by initiating a transaction. These two operations return unique tokens that we need

to pass as headers with each subsequent request. Initiating a transaction is the equivalent of the SQL BEGIN TRANSACTION command. This way, the database knows which session and transaction we are referring to. The session token is used to authenticate the user, while the transaction token is used to ensure that the operations are atomic. After we execute the operations, we need to commit the transaction to make the changes permanent. Lastly we need to close the session to free up resources.

5.4.4 *Conflict Resolution*

Immudb does not support conflict resolution out of the box. Under the hood, immudb uses transactions even in the case of a non-session login. The delay in the transactions is minimal, but it can affect the queries. In particular, if two writes happen at the same time (or close to each other), the database returns an error — `ErrTxReadConflict`. This error stands for "tx read conflict" and it means that the transaction was not successful because another transaction was committed in the meantime. It occurs even when the query is not a transaction, so for example when adding a new file to the database, and we write the contract to the database, it can fail. Immudb does not provide a way to automatically retry the transaction, so we had to implement the retry mechanism in the client. The official documentation acknowledges this issue and mentions that MVCC (Multi-Version Concurrency Control) is on the roadmap, but not yet implemented, hence such failures are expected and need to be handled by the client.

5.4.5 *Using Immudb to store the files (rejected idea)*

One way for us to verify that a malicious peer is not modifying/deleting the files is to store the files in an immutable database (this does nothing against bit rot or hardware failures). Since it is immutable, it preserves the history of the data, and we can always request to see the history of the file to verify that it has not been tampered with. So why do we not run an Immudb instance on each peer and store the files in it? The reason is that this introduces additional complexity.

While the peer cannot modify the file, without leaving a trace, it can always wipe the database and recreate it. This erases the history and make it seem like no changes were made. Any data stored on the peer is at risk of being deleted or modified, and therefore cannot be trusted. Perhaps we can check the timestamp of the time of storing the file, but this can be easily modified by the peer if they use a modified Immudb instance.

These and more considerations would need to be made if we were to use this approach. It enables a different class of attacks for which we would need to augment the system. In particular, we would need to implement ways to deal with these attacks in the Kademlia layer, and we would need to implement a new storage interface for Immudb, that Kademlia can use.

While this is not impossible, it is considerably more work than adapting Kademlia to work with PoR and store files on the disk.

5.5 KADEMLIA

Kademlia is the distributed hash table (DHT) that we use to store the files in the network. We are using the implementation from the official Rust libp2p crate [25]. We have made some extensions to the implementation to support our use cases. One of these is the local storage, which we touch upon in the following section. We have also included a direct peer to peer communication channel, which is referred to as a request-response protocol in the libp2p documentation, and we use it to perform audits required for the Proof of Retrievability. This was required since the base implementation of Kademlia is closed for extension, and we needed a way to communicate with the peers directly.

An important note here is that the Kademlia implementation supports only some basic operations, some of which differ from the original Kademlia paper. For example, the storage operation stores the value on the current node, while the original Kademlia paper stores the value on the closest node to the key. Therefore, to achieve the original behavior of the Kademlia paper, we had to implement part of the Kademlia protocol ourselves. For the PUT operation in particular, we first use GET_CLOSEST_PEERS to find the closest peers to the key, and then perform the PUT operation on the peers we found. This was not a huge downside since it allowed us to easily inject the PoR protocol into this operation. The initialization step of the PoR protocol happens when we store the file for the first time. We have to create a secret vector that the Keeper, where the file is stored, does not know. So once we find the closest peers to the key, we initialize the PoR secret vector, and then we store the file on the Keeper, while simultaneously storing the PoR secret as well as additional metadata in the ledger.

5.6 LOCAL STORAGE

We rely on Kademlia to store files in the network. Or more precisely, we rely on the Kademlia protocol to route the files to the correct peer. However, the implementation of the Kademlia protocol in the `libp2p` crate does not support storing the files on the disk. It only comes with a simple in-memory storage, designed for testing and examples.

Since we want the storage to be persistent, we need to implement our own storage module. Fortunately, the `libp2p` crate is designed to be easily extensible. It provides a storage interface that we can implement to store the files on the disk. The implementation can be split into two parts — converting the Record to bytes and storing the bytes to disk.

Files in Kademlia are shared as Records. A Record is the key, value, publishers, and expiration date. We cannot store only the value, because when the file is requested, it is requested as a Record, and we need to reconstruct the Record. If we are missing the metadata, we cannot construct a valid Record. Unfortunately, serializing and deserializing a Record is not trivial. The Record is not designed to be serializable, mainly due to the expiration date field. The expiration date is a Duration, which is a struct intended for internal use in code. It supports very precise time calculations, additions, and comparisons, but it is not designed to be sent over the network or serialized. Sending the

Duration over the network is not an issue for us, since we do not need nanosecond precision for the expiration date.

We implemented a custom serializer for the Record, which serializes the Record to bytes in a YAML format. Record is a struct from the `libp2p` crate, and therefore we cannot change it or extend it by implementing serialization for it. However, we can use a workaround — we can create a wrapper struct around the Record, and implement all the necessary functionality for it. A wrapper struct is not ideal, since we cannot break the scope of the Record fields, i.e., we cannot access the private fields of the Record. But with using the getter/setter methods, we can access the fields of the Record, and implement the serialization and deserialization.

The second part of the storage module is writing or reading the bytes to or from the disk. To achieve this we use the crate `object_store`, which provides asynchronous file operations using an S3-like interface. The choice of using an S3-like interface is guided by the fact that a lot of storage solutions provide an S3-compatible interface. These range from cloud storage, to local disk storage (as this crate), to storage apps running in Docker containers. This essentially means that, we can easily isolate the stored files from the rest of the system, by running a Docker container and connecting to it using the S3 interface. This is a huge advantage, as it allows us to not worry about what files are being stored. Even if a malicious file is stored, it is isolated from the rest of the system, and it cannot affect the rest of the system.

Having asynchronous file operations is a good feature as they do not block the rest of the system, and file operations can be slow. Kademlia's interface, however, is synchronous, and we need to adapt the asynchronous file operations. We achieve this by spawning a new thread for the file operation and blocking until the operation is finished. This is a workaround that prevents parallel file operations, but it does not block the whole system, since Kademlia is running in a separate thread.

Implementing the storage module is a big step towards making the system production ready. It also helps immensely in testing, as we can easily inspect the files stored on the disk, and rerun benchmarks without having to initialize the state of the system every time.

5.7 Malicious peer behaviors

When we talk about malicious behaviors in the network, we are not necessarily interested whether they are intended or not. Corrupting a file could be both intentional and unintentional, and both cases can be implemented in the same way. We will refer to these behaviors as malicious behaviors.

In order to test the system, we need to simulate malicious peer behaviors. We have implemented a few malicious peer behaviors that we use to test the system. These behaviors get injected as a module into the peer, and they can be turned on or off at any time. The dependency injector is a huge help here, as we can easily change the behavior of the peer by changing the configuration file.

All behaviors are implemented similarly to each other. They run an infinite loop, that triggers every few seconds. Inside the loop they perform a malicious action. For example, the behavior that does not store

any files, checks if any new files have been stored on the peer, and deletes them.

5.8 gRPC Gateway

Interacting with the system is done through the gRPC gateway. gRPC is a high-performance, open-source universal RPC framework. It is designed to be efficient and lightweight, and it is based on HTTP/2. For our purposes these are nice to have features but not crucial, because we use gRPC only for the communication between the client and the first peer. For example, if we want to upload a file to the network, we send a request to upload a file with the file's contents to one peer over gRPC. Since clients usually run their own peer, this communication is local, and therefore very fast, regardless of the protocol used.

We chose gRPC because it is widely adopted and has good support in the face of tools and libraries. Also, in an earlier version of the system, we used gRPC to communicate between the peers, in particular the verification was done over gRPC. We have since moved the verification to the Kademlia layer, but we kept the gRPC gateway for the client to communicate with the first peer.

To implement gRPC we define the interfaces in a `.proto` file, and then we use the `tonic` crate to generate the server and client code. The `tonic` crate is a gRPC implementation for Rust, that generates the boilerplate code of running a gRPC server and client for us. To implement the server we simply implement the generated interfaces and provide the implementation to the server.

An example of a `.proto` definition could be Listing 4.

Listing 4: Example of a .proto file

```
message RetrieveRequest {
    string name = 1;
}

message RetrieveResponse {
    bytes content = 2;
}

service KissService {
    rpc Store(StoreRequest) returns (StoreResponse);
}
```

This definition defines a service called `KissService` with one method `Retrieve`, that we can then implement in Rust such as Listing 5.

Listing 5: Example of a gRPC server implementation

```
impl KissService for KissServiceImplementation {
    async fn retrieve(
        &self,
        request: Request<RetrieveRequest>,
    ) -> Result<Response<RetrieveResponse>, Status> {
        // ... get file from Kademlia

        Ok(Response::new(RetrieveResponse {
```

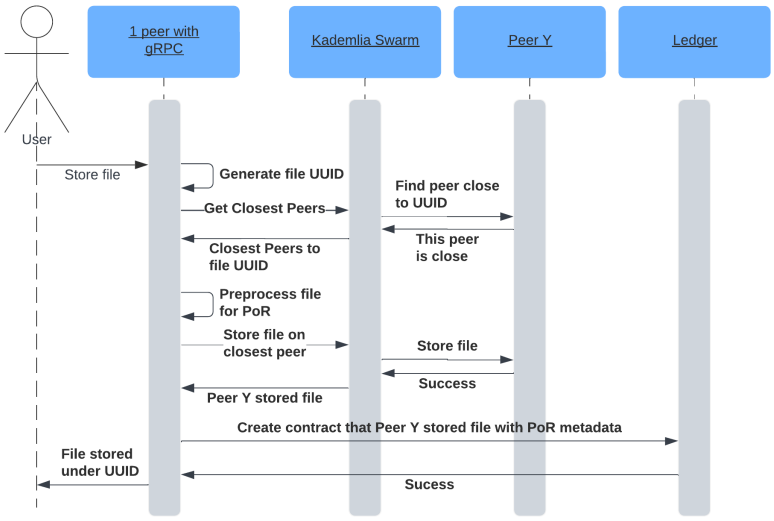

Figure 10: Storing a file in the network

```
            file_content,
        }))
    }
```

Once we have the server implemented, we tell `tonic` to run the server, and handle requests by calling the methods of `KissServiceImplementation`.

Calling the peer is now as simple as sending a gRPC request with a tool like `grpcurl` Listing 6.

Listing 6: Example of a gRPC request

```
grpcurl \
-plaintext \
-import-path proto \
-proto kiss.proto \
-d "{\"name\": \"<OUR-FILE-UUID>\"}" \
'[::1]:2000' \
kiss_grpc.KissService/Retrieve
```

In Figure 10 we can see the sequence of operations that happen when we store a file in the network through the gRPC gateway.

An important note is that by default the gRPC crate, `tonic`, processes requests with maximum size of 4MB. We need to increase this size by setting the `max_decoding_message_size` and `max_encoding_message_size` in the gRPC configuration to a higher value. Similarly, in the Kademlia module we have to set the `set_max_packet_size` to a higher value, because by default Kademlia does not allow sending packages larger than 1MB.

We have chosen to use an external tool to make the requests, instead of implementing a client, because it allows faster iteration and testing. If the server changes we do not need to recompile the client, or change the names of the endpoints in the client. For testing purposes, UNIX provides plenty of tools that can record network traffic, time the requests, and so on, so it is easier to use them than to implement a client.

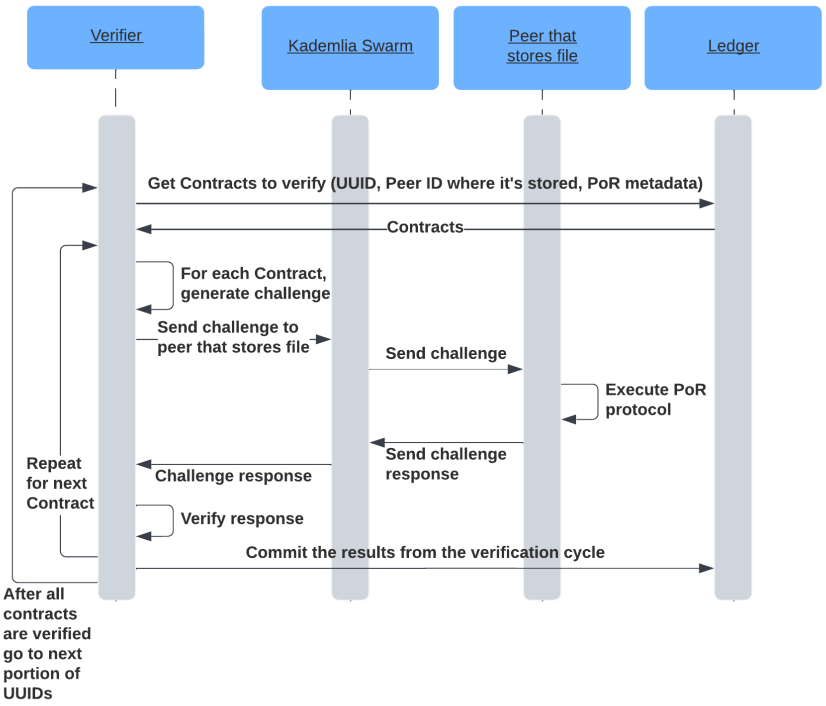

Figure 11: Verification of a file in the network using PoR

5.9 PROOF OF RETRIEVABILITY

We run the PoR protocol during each verification of a file. The protocol has two steps — the Verifier generates a challenge and the Keeper responds to the challenge, as seen in Figure 11.

The Proof of Retrievability (PoR) is the protocol that we use to verify that the file is stored on the peer. The protocol is based on Dynamic proofs of retrievability with low server storage [6]. The original protocol is implemented in C and was implemented to provide an example that the protocol works. We have implemented the protocol in Rust, and we have made some changes to the protocol to fit our use case.

The original protocol has two versions — the default one that we are implementing, and a public version. The public allows any peer to verify that the file is stored properly without knowing the secret vector. For a decentralized system where we expect malicious peers, the public version is ideal, because we do not need to worry about the malicious peers knowing the secret vector, that is usually required to verify the file. We have opted to not implement the public version of the protocol, because of increased complexity, and it is not necessary to answer our research questions.

We are reusing the constants from the original protocol, such as the prime number and chunk size. The prime number is used for modulo arithmetic, and the authors of the paper have proven that the protocol is secure for the given prime number. The chunk size follows directly from the prime number size, which has 57 bits. Since we use the prime to do modulo arithmetic, the numbers we want to work with should be smaller. Since computers work with bytes, which are 8 bits, we have chosen the chunk size to be 7 bytes. Ideally we would want the chunk size to be 8 bytes, i.e., 64 bits, but this would require a prime number

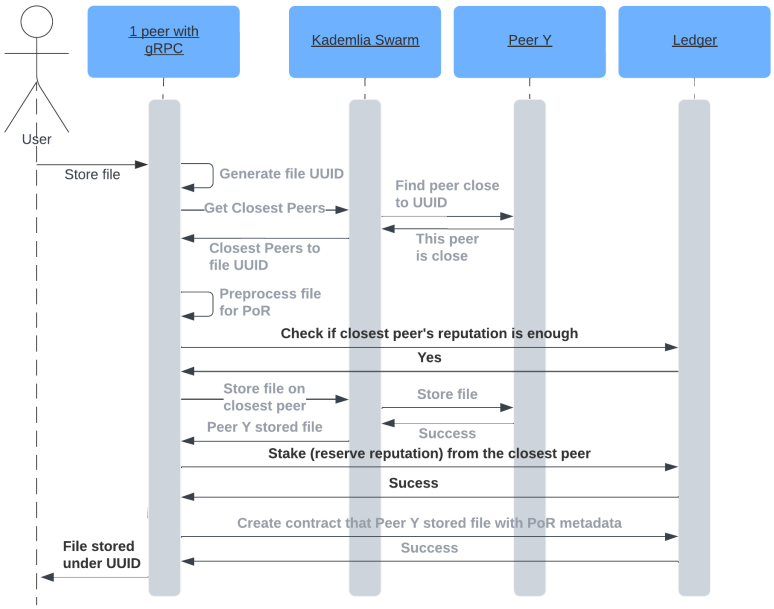

Figure 12: Storing a new file timeline with the reputation system in place

with more than 64 bits, which would make the calculations slower. Not all processors support 128-bit numbers.

Working with 7 bytes is not ideal, as we have to read the files in chunks of 7 bytes, and then pad the last chunk with zeros before converting the chunk to a number. We have implemented the reading of chunks in a Rust way using iterators, which differs from the C implementation, where the language allows reading the memory directly in chunks of 7 bytes. The computations themselves (matrix multiplication) we have not changed, because we wanted to test our implementation directly against the original implementation. Apart from end-to-end tests, we wanted to test the intermediate steps of the protocol to ensure correctness. This is mainly because Rust does not allow direct memory manipulation and implicit type conversions, which the C language does.

The last thing to mention is the random number generator used in the protocol. A random number generator is used to generate the secret vector that the Keeper does not know and the challenges for the Keeper later on. For the purposes of testing we are using a seeded Mersenne Twister, which allows us reproducible results. In a real-world scenario, we would use a cryptographically secure random number generator such as ChaCha20 [4]. For the purposes of this thesis, a seeded Mersenne Twister is sufficient, as it is predictable, and we do not want to test the random number generator.

5.10 Reputation System

Adding a reputation system to keep track of previous performance of the peers is essential. It changes the way we perform the storage and verification of files, since we need to keep track of the reputation of the peers. The changes are reflected in Figure 12 and Figure 13.

Using proof of retrievability we can verify that the file is stored on a peer. This verification happens at a fixed point in time, and it does not

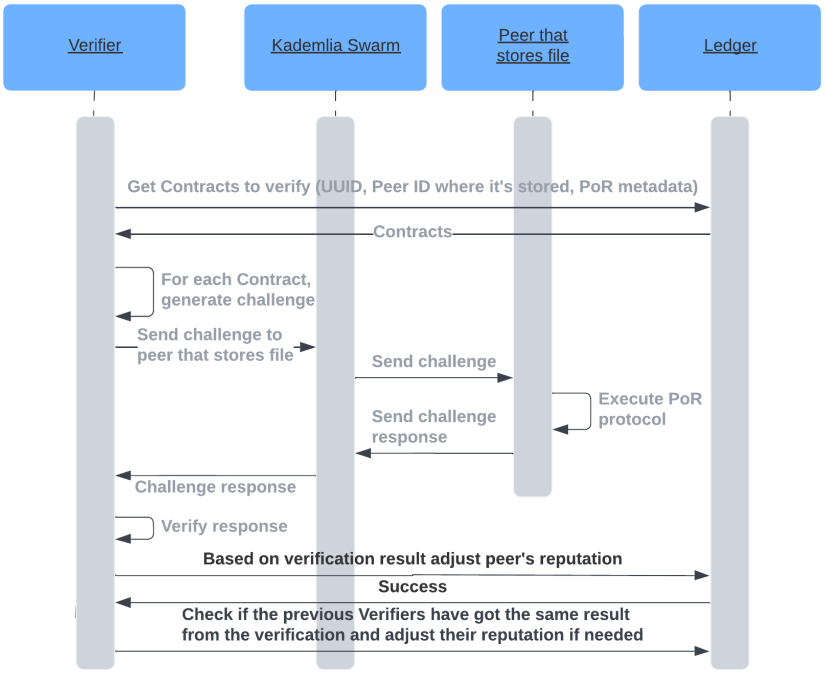

Figure 13: Auditing timeline with the reputation system in place

guarantee that the file will be stored on the peer in the future. It also gives us no information if the file was stored on the peer in the past. Knowing the past performance of a peer is essential for a decentralized system, where we know nothing else about the peers in the network. In Chapter 4 we discussed how a reputation system can help us keep track of the behavior of the peers in the network. Ideally this reputation needs to be stored in a decentralized manner as well, so that no peer can tamper with the reputation of another peer.

Ideally, we could use a blockchain, which is exactly a decentralized ledger, to store the reputation of the peers. Alternatively, we could come up with a way to store the reputation in the system itself, since it is designed to be decentralized and aims to ensure the integrity of the data, i.e., it should be tamper-proof.

Both of these solutions are viable for a production-ready system, however they introduce additional complexity and potential weak points in the system, which we have not analyzed.

For the purpose of this thesis we have decided to use the immutable database Immudb [11] to store the reputation of the peers. It is centralized, but allows us to abstract ourselves from solving the problem of designing a decentralized and tamper-proof reputation system, which is outside the scope of this thesis. We consider the database as a single source of truth for the reputation of the peers during our tests.

The reputation system is designed to keep track of the behavior of the peers in the network. When a peer accepts to store a file, it stakes a certain amount of reputation points. Staking is temporary decrease of reputation points, which is returned to the peer after the expiry date of the file. An alternative approach is to return the reputation points to the peer in small increments after every successful audit.

Altering the reputation is a two-step process, because we need to ensure that the operation is atomic and Immudb does not support com-

plex queries. We instantiate a transaction, read the current reputation of the peer, alter it, and write it back to the database. We go into detail about the atomic operations in Section 5.4.3.

Immudb provides very nice guarantees about the integrity of the data, and we can be sure that the reputation of the peers is not tampered with. Therefore, it is a good fit for the reputation system. However, when we wanted to avoid the complexity of integrating with a blockchain, we did not expect that Immudb would not support atomic operations, and the complexity this would introduce in the system.

6 EVALUATION

Before we evaluate the system, we shall make a few notes that apply to the evaluation of the system as a whole.

6.1 NOTES

All the tests and benchmarks were performed on a single machine. The machine has 16GB of RAM, an M1 Apple Silicon processor 3200 MHz, and a 512GB SSD. Due to these specs, the benchmarks are run with files between 1MB and 100MB. We ran benchmarks with files up to 1GB and the results were consistent with the results from the smaller files, i.e., the algorithms exhibit the same behavior, but rerunning the benchmarks takes a lot of time.

The system is written in Rust, which means no garbage collection or runtime overhead. We expect very little memory footprint while the system is idle, which is indeed the case as seen from running 1000 instances in Figure 14. No CPU usage and around 2MB of memory per instance is used when the system is idle aligns with our expectations. Because the system consumes almost no resources when idle, we can run many instances on a single machine. We will not account for the overhead of the system being idle in the benchmarks.

6.2 PROOF OF RETRIEVABILITY (POR)

In this section, we evaluate how the PoR protocol performs in a real-world scenario. We have discussed how PoR is used to ensure the integrity of the data stored on the nodes. Now, we evaluate how viable is it to use PoR as a part of a storage system. Does it adhere to the Section 1.4? And is it fast enough to be used in a real-world scenario? The protocol has two parts — the initial storing of a file and the subsequent repeated audits.

6.2.1 *Storing a file using PoR*

When storing a file in a distributed system we have to perform some calculations on the data to generate the metadata required to run the

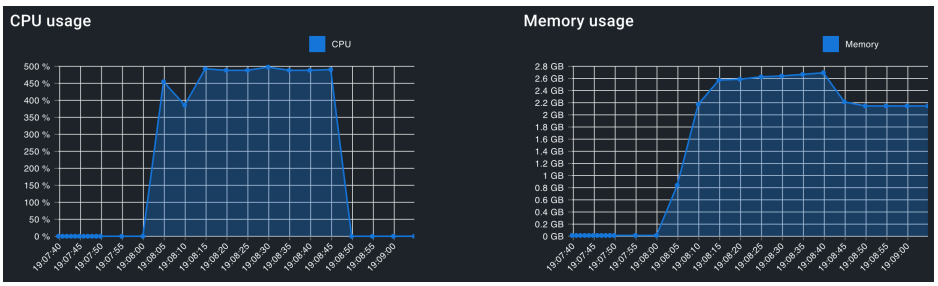

Figure 14: Starting 1000 instances

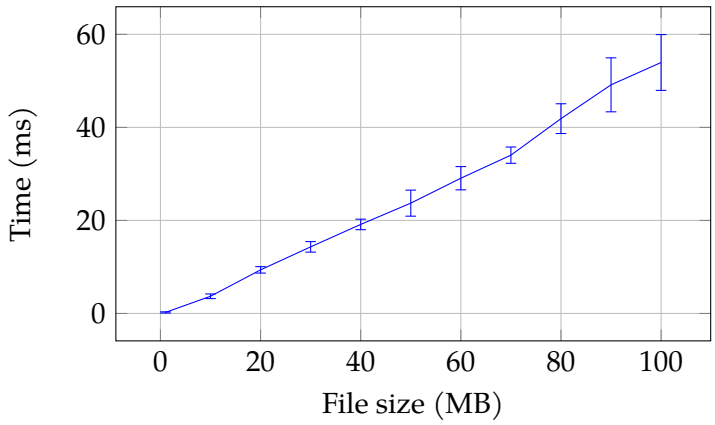

Figure 15: Time required for the first peer to initialize the metadata for PoR when storing a file.

PoR audits later. This is a one-time operation, which is not crucial to be fast, but it would provide better user experience if it is. Ideally the impact should be minimal and proportional to the size of the file, as seen in Figure 15.

6.2.2 *How many audits can we perform in parallel?*

The audit is a two-step process. First, the Verifier sends the challenge to the Keeper and then the Keeper responds with the proof.

We expect the load on the Verifier to be negligible, while the load on the Keeper to be significant. The Verifier should do as little work as possible, because we want to perform as many audits as possible in parallel. This way we can ensure that audits are performed often enough to ensure the integrity of the data. The load on the Keeper is expected to be high, because the Keeper needs to read the file from disk and execute the PoR algorithm. Ideally that algorithm will be linear in time complexity in the size of the file.

The Verifier performs 2 operations — sending the challenge and verifying the proof. We benchmarked these operations and the results are shown in Figure 16 and Figure 17.

The results show that generating the challenge and auditing are both very fast operations — for a 100MB file, the Verifier can generate the challenge in 0.027ms and audit the proof in 0.137ms.

We can observe something interesting the in the results — the graph looks almost logarithmic, instead of linear. This is due to cache locality since both the operations are executing code on sequential memory addresses. Both the challenge and the response are essentially a vector of numbers, which is a very cache-friendly data structure.

The Keeper performs 1 operation — generating the proof. This is done by reading the file from disk and running the PoR algorithm on it. The PoR algorithm is expected to be linear in time complexity in the size of the file. We benchmarked this operation and the results are shown in Figure 18.

The operation gets slower as the file size increases. In theory the time complexity should be linear, but it is superlinear in practice. This is most likely caused by the unoptimized matrix multiplication algorithm that we are using to implement PoR.

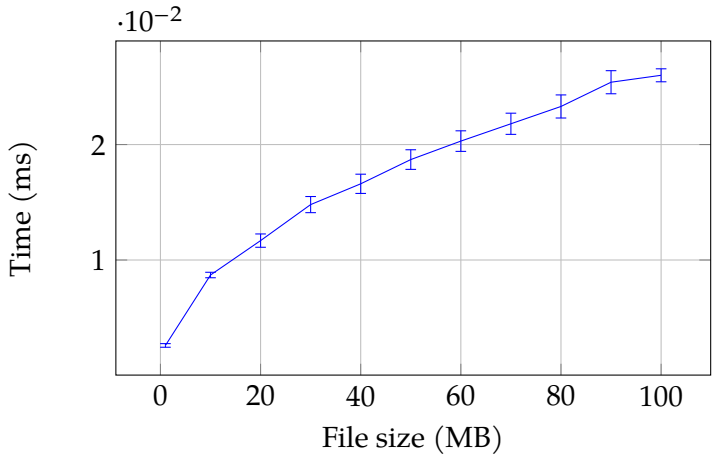

Figure 16: Time required for the Verifier to create the challenge for PoR.

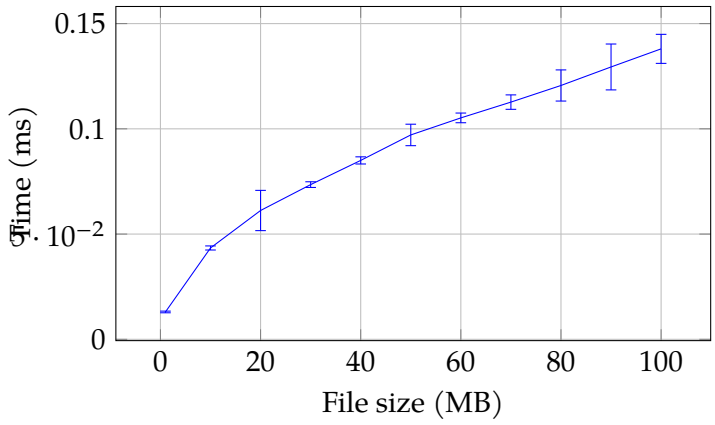

Figure 17: Time required for the Verifier to audit the challenge response PoR.

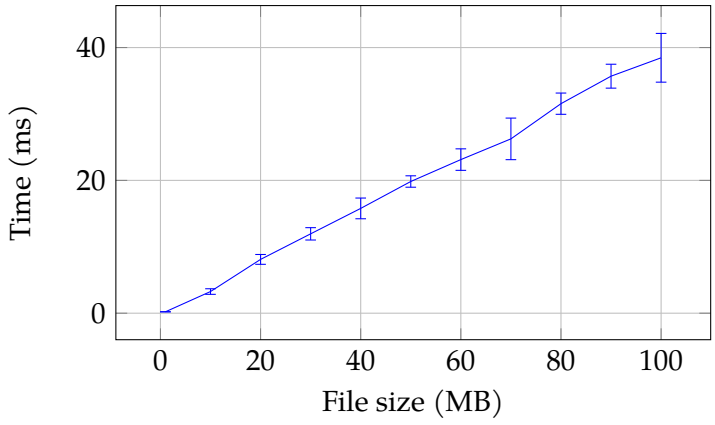

Figure 18: Time required to create the challenge response (proof) for PoR on the Keeper side, without reading the file from disk.

6.2.3 *Is PoR the limiting factor in the system?*

One of the main questions we want to answer in this thesis is whether the validation component affects the system's performance. From the results we can see that while the evaluation aspect introduces more workload, however, it is within reasonable levels.

In the previous section we saw that the Verifier can generate the challenge and audit the proof very quickly. Since the speed of reading from an SSD is on average 7000 MB/s, the PoR protocol in the Verifier will not affect the system's performance. It is more likely, that the network connection between the Verifier and the Ledger will be the limiting factor, but we will not evaluate network speeds as they are out of the scope of this thesis.

The Keeper can generate the proof at a rate of 38ms for a 100 MB. This is faster than the access speed of an HDD, which is around 500 MB/s or 200ms for a 100 MB. However, it is slower than the access speed of an SSD, which is around 7000 MB/s or 14ms for a 100 MB. Whether the PoR protocol is the limiting factor or not depends on what kind of storage the Keeper is using.

In conclusion, the Verifier can send challenges in parallel, but the Keeper nodes will still be the limiting factor in the system.

6.3 PERFORMANCE OF THE SYSTEM

Our final goal is to augment a distributed storage system with verification based on PoR. In the process we want to ensure that the performance of the system is not degraded. The main part we are changing is the way files are stored, by injecting the PoR initialization, and the idle state of the peers — during which they perform audits. We performed the benchmarks starting with 10 and up to 100 peers in the network. Since the tests are run on a single machine, running more peers would not give us any useful information, because the resources of the machine will be exhausted from running the peers in idle state. This is also the reason we are using files of size up to 10MB. Larger files would take more time to read and write to disk, but would not give us any additional information about the performance of the system.

From Table 2 we can see the overhead of running the PoR algorithm and writing the metadata to the ledger. It takes six to 10 times longer for the storage request to be processed in comparison to running only the base Kademlia. While our implementation is not optimized and processes the requests sequentially, we can see that the overhead is significant. We run PoR separately for each replica that is to be stored. We would have expected two to three times longer processing time, not six times longer. The main cause for this is the ledger. Sending requests to the ledger is sequential and while the ledger has high throughput, a lot of requests tend to fail. A lot of times when we send a request to the ledger, it would fail because the state has changed. This is not something documented in the Immudb's documentation, but to circumvent it, we are retrying the requests until they succeed, which is usually on the second or third try.

Despite the overhead, storing a file is a one time operation, so while we can improve it, it is not a critical issue.

6.3.1 *Running a Verifier and a Keeper separately or together*

The final goal is for each peer in the system to run one binary, that is both a Keeper and a Verifier. However, our implementation is not highly parallelized, so we expect that running the Keeper and Verifier separately will be faster.

The results of the benchmarks are once again in Table 2.

It is important to note that having 10 or 100 nodes in the network does not seem to impact the time to store files. This is the case because before running each benchmark we wait for the peers in the network to connect to each other. We do not run the tests immediately after a cold start of the system.

The Time without Verifier from Table 2 is when the node that accepts the storage request does not run the Verifier, and the Time with Verifier is when the node runs the Verifier as well. The time to process a request with the Verifier running is significantly longer. This is mainly due to the outlier requests, which have to be processed while a verification is running. The longest request time column shows the extreme outliers, with the standard deviation showing how much the requests vary. With our current architecture, the ledger module and the Kademlia swarm module are both singleton modules, locked behind mutexes, to ensure that no race conditions occur. This causes the Verifier to occasionally take a hold of these mutexes and block the storage requests from being processed. This can be circumvented by running a pool of connections to the ledger and the swarm. We have left this for future work as it can be circumvented by each peer running 2 nodes — one for storage and one for verification.

Number of peers	Data stored	Kademlia	Time without Verifier	Time with Verifier	Longest request time
10	1KB	5ms	53ms	170ms	375ms
10	10KB	5ms	54ms	182ms	470ms
10	100KB	10ms	83ms	184ms	457ms
10	1MB	45ms	320ms	462ms	530ms
10	10MB	450ms	2710ms	3280ms	3050ms
100	1KB	9ms	56ms	113ms	7200ms
100	10KB	10ms	72ms	82ms	7000ms
100	100KB	12ms	81ms	108ms	5600ms
100	1MB	50ms	327ms	500ms	5000ms
100	10MB	410ms	2724ms	4523ms	13000ms

Table 2: Storing files in the system with replication factor of 3. Each benchmark is ran 100 times. Longest request time and Standard deviation both refer to the column for Time with Verifier.

We are interested in how fast the system can detect a malicious peer. We define detecting a malicious peer by the time it takes for the system to find the first sign of misbehaving. The faster a malicious peer is detected, the faster the system can react to it. We expect with the increase of files in the system or increase of peers, the time to detect a malicious peer to decrease.

6.4.1 *Detecting a malicious Keeper*

For Keepers, the malicious behavior is corrupting data. We expect with the increase of files in the system, the time to detect a malicious Keeper to decrease.

All files are verified each cycle in our testing environment. In a real world scenario, this is achievable only with very long cycles. The alternative option is to check a random subset of the files each cycle, which will provide probabilistic guarantees that a malicious Keeper will be detected.

The Verifiers split the work of verifying the files based on the file's key. As an example, if we have two peers (2 Keepers and 2 Verifiers), the Verifiers would split the files of each Keeper and take turns verifying them during alternating cycles. This is illustrated in Figure 19.

This means that when a peer becomes malicious they have at most one cycle of time, before they are detected. We are running the benchmarks in Table 4 with various amount of data in the system. The cycle times are chosen arbitrarily in order to provide an overview of how the system behaves with different cycle times and to show the connection between the amount of data and the cycle length. The cycle time when running the system in production should be set to a value proportional to the amount of data and number of files stored in the network. We are running the benchmarks with 1 corrupt peer in the network, because if we have more the chance of a corrupt peer being detected increases. We are interested in how long it takes to discover the first signs of misbehaving in the network. The general observation, which is not surprising, is that the more files we have in the network, the faster it is to discover a corrupt peer, because the greater the chance it is for the corrupt peer to store a file. For example, with 100 peers and 100 files in the system, the corrupt peer is expected to store 1 file, That means that all the files need to be verified in order to discover the corrupt peer. This is even more apparent with 10 peers where the time to discover a corrupt peer with 1000 files is 41ms in comparison with 1120ms for 100 files.

6.4.2 *Detecting a malicious Verifier*

For Verifiers, the malicious behavior is not performing audits. We expect with the increase of files in the system, the time to detect a malicious Verifier to decrease. Detecting a malicious Verifier is very similar to detecting a malicious Keeper, and we could show similar results for the minimum times required to detect a malicious Keeper. Instead, we would like to confirm that a malicious Verifier will be detected in

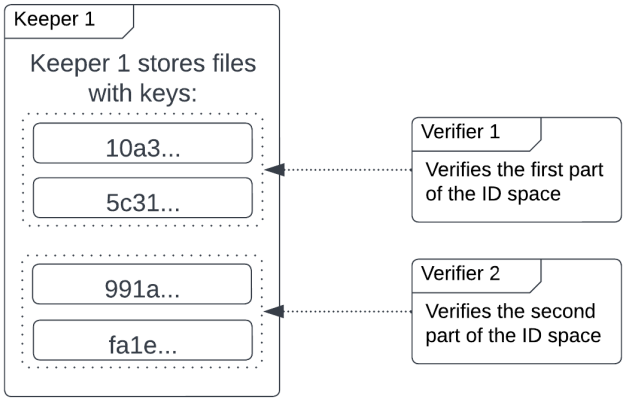

Figure 19: Splitting the ID space between two Verifiers

Number of peers	Data stored	Time per cycle	Average time	Standard deviation
10	100x1KB	6s	2792ms	4281ms
20	100x1KB	6s	5891ms	2713ms
30	100x1KB	6s	2038ms	2000ms
40	100x1KB	6s	722ms	750ms
50	100x1KB	6s	16220ms	6359ms

Table 3: Time to discover a corrupt Verifier in the network with 1 corrupt Verifier

a reasonable time. Reasonable time is within two cycles, because the Verifiers are expected to perform audits every cycle.

The results are shown in Table 3. The average time to detect a malicious Verifier indeed remains within two cycles. There is no significance in the actual time - the difference between 2000ms and 5000ms average is not meaningful, as it depends on what time the peer initiates the auditing cycle. The peers do not have synchronized clocks, so one peer's cycle might just start 3000ms later than the other. Again, at 50 peers or more the hardware becomes overloaded and starts affecting the times to detect malicious Verifiers. This is expected, mainly because of the communication overhead between the peers.

We are downsizing the tests to only run with 100 files in the system with six seconds cycle time. Performing so many verifications requires longer cycle times, otherwise the Verifiers start lagging behind, and we see increasing times to detect malicious peers as seen in Figure 20.

To deal with this, if we want to add more files to the system we would have to increase the cycle time, or increase the number of Verifiers.

6.4.3 *Are the minimum times to detect a malicious peer reasonable?*

During our benchmarks we are assuming that the malicious peers are partially well-behaved. In particular, they would not try to predict when

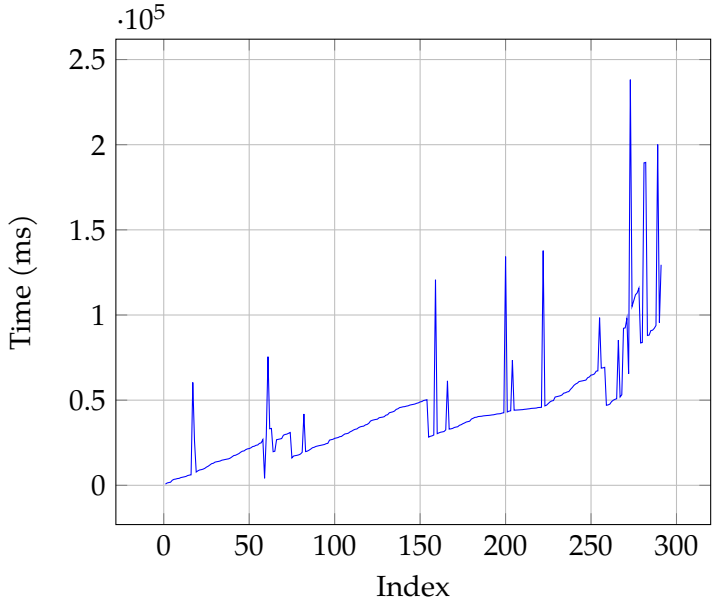

Figure 20: With too many files and short cycle times, the time to detect a malicious peer increases as the Verifiers start lagging behind

the Verifier would audit their files. Under a real attack, that would not be the case. The malicious peer could record the time between audits and then only come online during the next audit cycle. If there are not too many files in the system, during an audit cycle, all the audits would be performed at the beginning and then there would be a downtime until the next cycle. During this downtime the malicious peer could go offline and save resources, making the files unavailable but still passing the audits.

To mitigate this, the Verifier could also record how much time all the audits during a cycle take and then either:

1. Run a consensus algorithm with the other peers in order to shorten the cycle time. — This would increase the load on the system and make the audits more common, but lead to increased security.

2. Randomize the time and order in which audits are performed. — This would make it impossible for the malicious peer to predict when the audits are performed, but it has a probabilistic nature. We are not guaranteed to catch the malicious peer immediately after it becomes malicious, but statistically, after enough cycles it will be caught.

Both of these options are viable and depend on the requirements of the users of the system. If the users require a higher level of security and can spare the resources, the first option is better. However, if the users want to save resources they could opt for the second approach. We are not running the benchmarks with these options, because we are mainly interested in showing that the system can detect malicious behaviors in reasonable time if given the right parameters and resources. The analysis of the relation between the length of the verification cycle, the size of the data in the system, and how they affect the time to detect malicious behaviors, is left for future work.

Number of peers	Data stored	Time per cycle	Time to discover corrupt peer
10	100x1KB	6s	1120ms
10	1000x1KB	6s	41ms
20	100x1KB	6s	987ms
20	1000x1KB	6s	79ms
30	100x1KB	6s	4093ms
30	1000x1KB	6s	861ms
50	100x1KB	6s	8867ms
50	1000x1KB	6s	11607ms

Table 4: Time to discover a corrupt peer in the network with 1 corrupt peer

6.4.4 *Hardware limitations for the tests*

In Section 6.1 we discussed that the benchmarks are run on a single machine, and that we can run 1000 instances of the system. However, this is if we run them idle. If we run all the instances with enabled storage and verification, i.e., full load, we see degrading performance with more than 30 instances. In Table 4 we can see that for 10 and 20 peers, the results are consistent, but with 30 peers and with 50, the time to detect a peer is significantly longer. This is because the hardware becomes overloaded.

In conclusion, with sufficient hardware the system detects malicious peers when the number of peers increases and the number of files stored in the network increases. However, if the peers become overloaded, the time to detect a malicious peer increases.

6.5 Reputation system based on a ledger

The results of the PoR are stored in a ledger, which acts as the source of truth for the performance of the peers in the network. While ideally we would like to have a decentralized ledger, for the sake of simplicity we will use a centralized ledger in our evaluation. For the purposes of evaluation we could have used a simple SQL database or a local file, however, we used Immudb because it is what we used in the previous iterations of the system and at the time we were exploring it as a potential solution for the ledger. Immudb turned out to have a lot of limitations and we would not recommend it for a production storage system, such as the one described in this thesis.

6.5.1 *How does a ledger contribute to the system?*

Using PoR allows peers to detect other peers, which are disrupting the integrity of the data in the system. However, on its own, it is not enough to ensure that malicious peers are detected and removed from the network.

Is a reputation system based on a ledger a viable measure against malicious nodes? We expect the ledger to be the source of truth for the

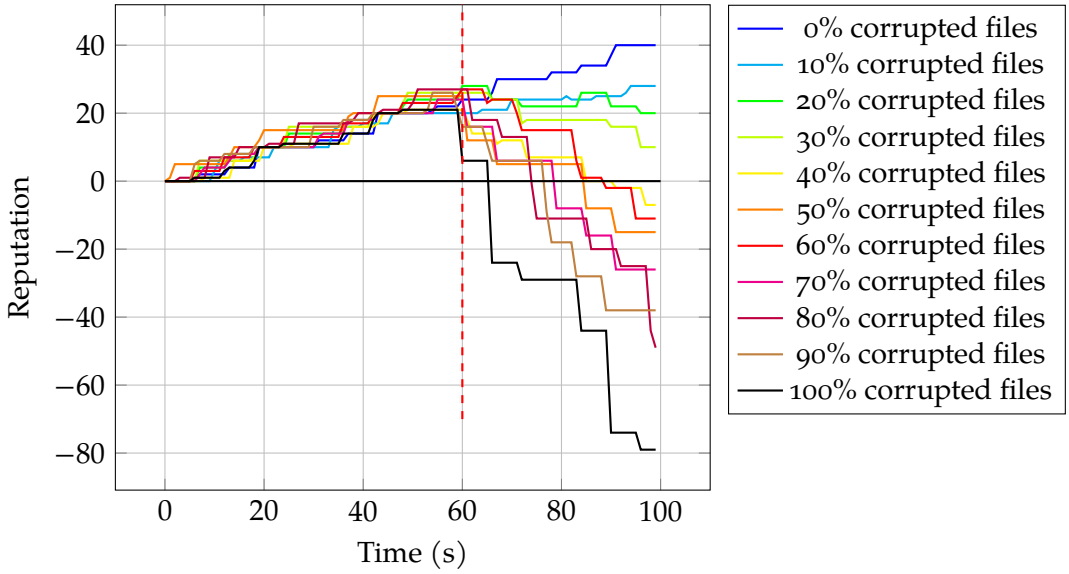

Figure 21: Reputation loss over time for different percentages of corrupted data.

reputation of peers and allow each peer to fetch said reputation and adjust its behavior accordingly.

In Figure 21 we can see how the reputation of a peer changes after it becomes malicious. All peers in the network start with the same neutral reputation, which is 0. After 60 seconds, that is 10 cycles, one peer becomes malicious and corrupts some percentage of the data. The reputation of the peer starts to decrease and goes into the negatives. Once the reputation drops below 0, the peer is seen as malicious by the other peers in the network, and can be removed from the network. This example is for a very short period of time, but in a real-world scenario, the trends will be the same, just over a longer period of time — malicious peers would lose reputation — the more data they corrupt, the faster they lose reputation.

6.6 BALANCING THE REPUTATION SYSTEM

One of the main questions we want to answer in this thesis is how to balance the reputation once we have PoR and the ledger in place.

The reputation is controlled by two parameters:

1. Reputation increase for storing a file or performing an audit.
2. Reputation decrease for corrupting data or not performing audits.

We need to find the right balance between the two. If the reputation increase is too high, the nodes can easily become malicious, as they can absorb the penalties. If the reputation decrease is too high, the nodes will not be able to recover from a mistake, such as being temporarily offline or being under heavy load for a period of time.

6.6.1 *Balancing the reputation gain/loss*

The reputation gain and loss are parameters of the system that can be adjusted. For our evaluation we are using the following parameters:

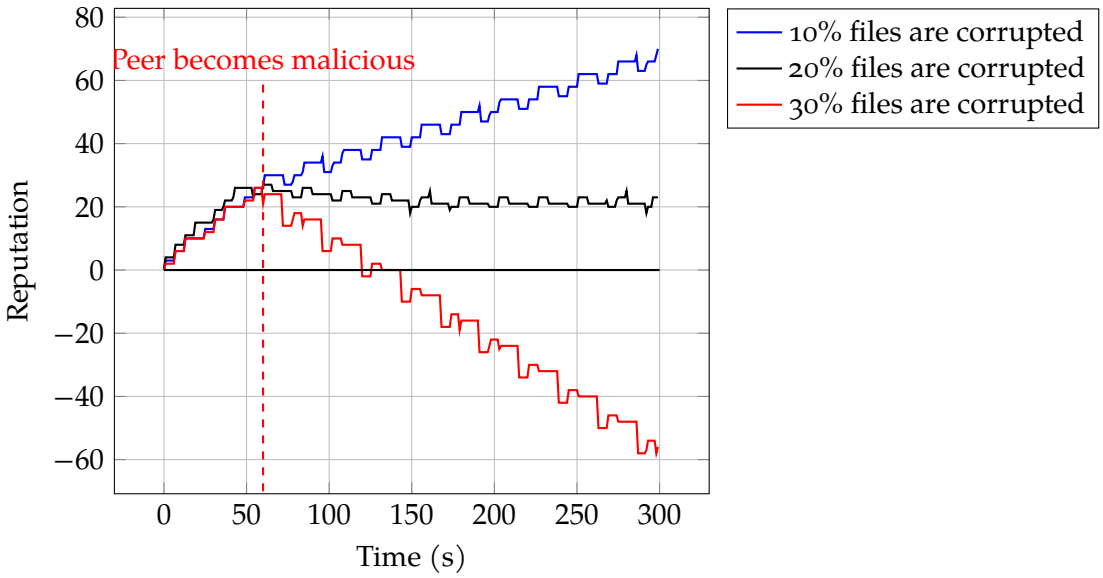

Figure 22: Reputation loss over time for 10%, 20%, and 30% corrupted data.

- Reputation increase for storing a file: 1
- Reputation increase for performing an audit: 1
- Reputation decrease for corrupting data: 5
- Reputation decrease for not performing an audit: 5

These numbers are arbitrary, and the only important thing is that the reward is 20% of the penalty. In such a configuration we expect that:

- Peers that fail less than 20% of the time to keep gaining reputation slowly.
- Peers that fail 20% of the time to maintain their reputation.
- Peers that fail more than 20% of the time to slowly lose reputation.

We ran the benchmark similarly to the previous benchmarks — 1 peer goes malicious after 60 seconds and corrupts some percent of the data. The results are shown in Figure 22. The benchmarks this time are run for 300 seconds, because we want to observe the long-term trends, unlike Figure 21 where we were interested in how the extreme cases compare.

As predicted when the reward is 20% of the penalty, a peer that is corrupting 10% of the data slowly increases its reputation, a peer that is corrupting 20% of the data maintains its reputation, and a peer that is corrupting 30% of the data slowly loses reputation. There are some oscillations in the data, which is due to the cycle time and what are the file IDs, that are being corrupted.

The conclusion we can derive from this experiment is that the leeway we give to peers is customizable, based on what the user requirements are. If we wish to have a very strict system and allow only 5% leeway (corrupted data or peers being offline), we can set the reward to be 5% of the penalty. On the other hand, if we know the peers are going to be going offline for 8 hours each day, because the users turn off their computers at night, we can set the reward to be greater than 33.3% of the penalty.

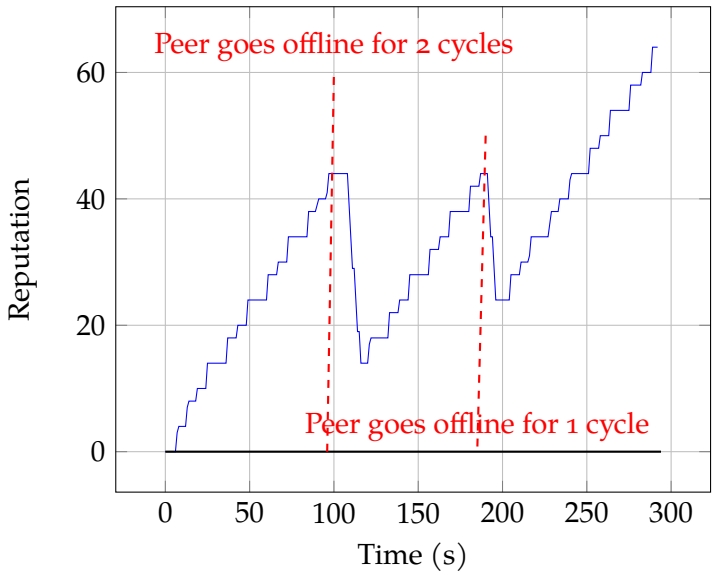

Figure 23: Reputation of a peer, which goes offline twice for two and for one cycles respectively.

6.6.2 *Recovering reputation*

It is possible for a peer to have downtime. Perhaps the network was disturbed, or the peer had to be restarted for updates. In such cases, the peer would lose reputation, which it can recover if it behaves well afterwards.

The time that a peer can be offline should depend on its contribution thus far. Since we treat going offline as failing all audits, this is not much different from a peer behaving maliciously for a short period of time. The main thing we want to see is that a peer can recover its reputation after failing a few audits, as long as it is above the threshold of being removed from the network (zero reputation).

We ran a simulation with a peer that goes offline for two cycles and then for one cycle. The reward in our simulation is 20% of the penalty. The results are shown in Figure 23. From the results we can see that the peer's reputation starts dropping shortly after it goes offline. The exact time depends on when the next audit will happen, but is within the one cycle range. Since the peer does not drop below zero reputation, after coming back online, it starts recovering its reputation, which is exactly what we want to see. There are no long-term effects of going offline for a short period of time.

As a conclusion, a peer can go offline for a short period of time and recover its reputation afterwards, as long as it has contributed to the network sufficiently before going offline. How long a peer can go offline depends on the configuration of the reputation adjustments, similarly to the previous section. If we configure the increase of reputation for succeeding audits to be ten times less than the decrease for failing audits, the peer can go offline for 10% of the time and maintain its reputation.

6.7 USING RUST

Rust is a systems programming language that is known for its performance and safety. For these reasons we chose it for this project.

Rust provides thread safety and memory safety without a garbage collector. Since the system is multithreaded, and it is composed of different modules that communicate with each other, Rust forces us to use mutexes or channels to communicate between threads. Without them, the compiler would not allow us to compile the code. We also get no memory leaks, as Rust has a strict ownership model. It is also a well-rounded language, with a lot of libraries and frameworks available, which makes it a good choice for a project like this. Also, because of the small memory footprint of the system and the low CPU usage, we can run many instances of the system on a single machine, allowing us to test the system with many peers.

While the pros of using Rust are many, there are some cons as well. In particular, the learning curve of Rust is steep and takes time to get used to the different coding style. Even after being comfortable with the language, it takes longer to develop in Rust because of the strict compiler rules and explicitness of the language. Running benchmarks is a difficult task as they are not fully supported — we have to use either external libraries or unstable features of the language.

Overall, Rust is a good choice for a project like this, because of its safety and performance, but the extra time to develop in Rust should be taken into account.

6.8 SUMMARY

It is difficult for storage system with potential malicious nodes to compete head-to-head with centralized cloud storage providers, which optimize every aspect of their system. Introducing Proof of Retrievability (PoR) into the system brings us closer to the guarantees that cloud storage providers offer. This is not without its costs, as we have seen in the evaluation — any additional work that the nodes have to perform will increase the time to process requests. It is an unavoidable trade-off between security and performance. However, the performance overhead of PoR is not so significant if we take into account that cloud services also perform integrity checks on the data stored on their servers, albeit with slightly lighter algorithms. The main bottleneck is the ledger, which in our tests is centralized. Even if it has high advertised throughput, under load, especially with a lot of data, and the many failed requests, it becomes a bottleneck. For PoR to be a viable solution for a storage system, the ledger has to be decentralized as well.

7

FUTURE WORK

At this point the system is not ready for production use. As seen from the Chapter 6, the centralized ledger is a bottleneck. It is also a single point of attack — the system is not fully decentralized until the ledger is also decentralized. For this purpose, an integration with a blockchain ledger is seen as the natural next step. In preparation for production-ready state and the usage of a blockchain, the data that is written to the ledger should be minimized. In particular, the verification results, reputation scores, etc. can all be stored in the system itself and only have a signed hash of the data written to the blockchain.

The PoR algorithm we are using is not public, meaning that the verifier requires a secret key to verify the proof. Keeping secrets in a decentralized system is not possible. It must be replaced by a public PoR algorithm. Most modern PoR algorithm papers propose both a secret-based and public versions. The PoR algorithm we are using in this thesis also has a public version, albeit less efficient and more difficult to implement.

On the implementation side, the major bottleneck is the communication between the different modules. A few of them have to be locked behind a mutex, which makes some parts of the system sequential. Ideally we want to parallelize all the operations that can be parallelized. This is achievable by using a pool of workers for the modules that support it — ledger, verifier, and grpc server.

8 CONCLUSION

In this thesis, we have presented a system for decentralized storage of data with a focus on data integrity. It shows how we can use PoR to bring these guarantees that cloud storage providers offer to decentralized storage systems, and what are the costs of doing so. We analyze how effective proof of retrievability is in detecting data corruption and how it can be used as a part of a decentralized storage system to increase the trust in the data stored.

Adding PoR to the system allows peers to keep track of what peers are misbehaving, and thus, the system can be self-healing — it can detect and remove said peers. While this solves the integrity problem, it is not a universal solution to attacks. It is always possible for a powerful enough adversary to take control of the system by controlling a majority of the peers. However, in other cases, the system can be resilient to attacks on the data, and can provide guarantees that the data is stored correctly. This comes at a cost of time and resources, as the system has to continuously monitor the peers and the data they store.

If a client has specific requirements, that can only be met by using decentralized storage, e.g., the client wants the data to not be censored, it is worth using a system like this. However, if a client needs simple data storage, that is durable and available, it is recommended to rely on cloud storage providers. These providers can usually provide better speeds, and they rely on proof of retrievability internally, which allows them to provide guarantees about the data integrity.

Combining PoR, a reputation system, and the modern decentralized networks, we can achieve a storage system that has similar guarantees as a cloud storage provider as listed in Section 2.3. Solving the integrity problem was a crucial step to achieve the requirement for data durability.

There are a few of the requirements, which are not yet met, such as the requirement for permanent deletion of data. PoR allows us to verify that data is stored, but we do not yet have a way to verify that the data is deleted upon request. This could pose legal challenged in certain use cases. We also have to account for the fact that the peers in the network are not regulated, so any claim for high availability or durability is under the assumption that the system is not undergoing a large scale attack — with enough resources, an adversary could take control of the network. Challenges like these prevent us from fully replacing cloud storage providers with decentralized storage systems, at least until we have solutions to them.

Finally, we are assuming the peers in the system are uniform, have similar resources, have similar availability, and so on. The results in Section 6.6.1 are based on such an assumption. However, if we want this system to be used by different users — some with their home computers, others on their phones, and some on data-center servers, we have to account for the differences. The reputation system must become dynamic, and the rewards cannot be the same for all peers. This

is a challenge that has to be addressed before the system can be used by a wider audience, that is not preselected to have similar resources.

APPENDIX

A.1 ORGANIZATION OF THE REPOSITORY

The repository can be found at https://github.com/luchev/kiss.

The project is organized as follows:

```
config ..... peer configs
data ..... default directory for peer's data
docs ..... documentation
logs ..... log files from peers
proto ..... protobuf definitions
src ..... rust source code
    build.rs ..... code generation pre compilation
    main.rs ..... main program
submodules ..... external packages
justfile ..... automation scripts
```

The whole project is controlled via just scripts (https://github.com/casey/just). The justfile contains all the necessary commands for building, running and testing the project. Some important commands are:

- `just build` - builds the project.
- `just run X` - builds and runs the project with config `config/X.yaml`.
- `just test` - runs the unit tests.
- `just run-many count` - runs `count` peers.
- `just thesis` - compiles this document.
- `just clean` - cleans logs, database, kills running peers — this is run before every benchmark test.
- `just put data` - inserts `data` into the network as a file.
- `just get uuid` - retrieves the file under `uuid` from the network.

A.2 RUNNING THE PROJECT

To run the project, you need a unix machine, the Rust compiler in nightly mode (cargo 1.74.0-nightly), the `just` tool, grpcurl, the protobuf compiler, and Docker. Compiling the latex files is done additionally with `tectonic`.

After cloning the repository we have to initialize the submodules as seen in Listing 7.

Listing 7: Cloning and setting up the repository

```
git clone git@github.com:luchev/kiss.git
cd kiss
git submodule update --init --recursive
```

For the minimum setup we need two peers running `just run base` and `just run peer1`. Once we have two peers running we can insert a file with `just put <data>`.

Running the benchmarks is done by using the just commands and reading the results from the logs. Most benchmarks output `info` level logs. Running the benchmarks is done as seen in Listing 8. While there is a script for running the benchmarks that uses the same commands, it is not recommended because before each benchmark we want to make sure that the system is in a fully initialized state. Sometimes the peers need a couple of seconds to connect to each other. The Immudb docker image might take a while to start as well. All these factors and more cause the first couple of requests to either fail or take much longer than the rest, which skews the results. For this reason it is often needed to rerun the last step `just put-bytes-times` to make sure we get consistent results.

Listing 8: Running the benchmarks

```
just clean # remove logs, data, reset database
just run-many 10 # run 10 peers
just run base &>base.log & # run the base peer
sleep 2 # wait for the peers to initialize and connect
just put-bytes-times 1000000 100
```

[1] *Amazon Simple Storage Service (S3) FAQs*. https://aws.amazon.com/s3/faqs/. Accessed: February 23, 2024. Amazon Web Services.

[2] Dave Levin, Katrina LaCurts, Neil Spring, and Bobby Bhattacharjee. "Bittorrent is an auction: analyzing and improving bittorrent's incentives." In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM '08. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 243–254. ISBN: 9781605581750. DOI: 10.1145/1402958.1402987. URL: https://doi.org/10.1145/1402958.1402987.

[3] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system." In: *SIGOPS Oper. Syst. Rev.* 44.2 (2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922.

[4] Daniel Bernstein. "ChaCha, a variant of Salsa20." In: (Jan. 2008).

[5] Ion Stoica, Robert Morris, David Karger, M. Kaashoek, and Hari Balakrishnan. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications." In: *ACM SIGCOMM Computer Communication Review, vol. 31* 31 (Dec. 2001). DOI: 10.1145/964723.383071.

[6] Gaspard Anthoine, Jean-Guillaume Dumas, Michael Hanling, Mélanie Jonghe, Aude Maignan, Clément Pernet, and Daniel Roche. *Dynamic proofs of retrievability with low server storage*. July 2020.

[7] Tung Le, Pengzhi Huang, Attila A. Yavuz, Elaine Shi, and Thang Hoang. *Efficient Dynamic Proof of Retrievability for Cold Storage*. Cryptology ePrint Archive, Paper 2022/1417. https://eprint.iacr.org/2022/1417. 2022. DOI: 10.14722/ndss.2023.23307. URL: https://eprint.iacr.org/2022/1417.

[8] Protocol Labs. *Filecoin: A Decentralized Storage Network*. https://filecoin.io/filecoin.pdf. Accessed: February 23, 2024. 2017.

[9] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System." In: *Lecture Notes in Computer Science* 2009 (Mar. 2001). DOI: 10.1007/3-540-44702-4_4.

[10] Juan Benet. *IPFS: A Peer-to-Peer Hypermedia Protocol*. https://ipfs.io/. Accessed: February 23, 2024. 2014.

[11] *Immudb, open source immutable database*. https://immudb.io/. Accessed: February 23, 2024. Codenotary, Inc.

[12] Petar Maymounkov and David Eres. "Kademlia: A Peer-to-peer Information System Based on the XOR Metric." In: vol. 2429. Apr. 2002. ISBN: 978-3-540-44179-3. DOI: 10.1007/3-540-45748-8_5.

[13] *NEAR Protocol*. https://near.org/. Accessed: April 11, 2024.

[14] John Kubiatowicz et al. "OceanStore: an architecture for global-scale persistent storage." In: *SIGPLAN Not.* 35.11 (2000), pp. 190–201. ISSN: 0362-1340. DOI: 10.1145/356989.357007. URL: https://doi.org/10.1145/356989.357007.

[15] P. Druschel and A. Rowstron. "PAST: a large-scale, persistent peer-to-peer storage utility." In: *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. 2001, pp. 75–80. DOI: 10.1109/HOTOS.2001.990064.

[16] Antony Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems." In: vol. 2218. Jan. 2001, pp. 329–350.

[17] *Ping from London to Christchurch*. https://wondernetwork.com/pings/London/Christchurch. Accessed: April 11, 2024.

[18] Kevin Bowers, Ari Juels, and Alina Oprea. "Proofs of retrievability: Theory and implementation." In: Nov. 2009, pp. 43–54. DOI: 10.1145/1655008.1655015.

[19] Ingmar Baumgart and Sebastian Mies. "S/Kademlia: A practicable approach towards secure key-based routing." In: vol. 2. Jan. 2008, pp. 1–8. ISBN: 978-1-4244-1889-3. DOI: 10.1109/ICPADS.2007.4447808.

[20] *Sia: Simple Decentralized Storage*. https://sia.tech. Accessed: March 8, 2024.

[21] *Solana*. https://solana.com/. Accessed: April 11, 2024.

[22] *Substrate*. https://substrate.io/. Accessed: April 11, 2024.

[23] Zooko Wilcox-O'Hearn and Brian Warner. "Tahoe - The least-authority filesystem." In: Oct. 2008, pp. 21–26. DOI: 10.1145/1456469.1456474.

[24] Codenotary. *immudb: Immutable database with built-in cryptographic proof and verification*. Accessed: April 11, 2024. 2024. URL: https://github.com/codenotary/immudb.

[25] libp2p Developers. *rust-libp2p: A Rust implementation of the libp2p networking stack*. https://github.com/libp2p/rust-libp2p. Accessed: April 11, 2024.