DECENTRALIZED STORAGE SYSTEM WITH VERIFICATION

IVAN LUCHEV, 202202879

THE GROUP NAME

PROJECT REPORT

January 2024

Advisor: Niels Olof Bouvin

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CONTENTS

1 ABSTRACT

Version 1 of this project proposed a storage system based on Kademlia, which used verification to ensure the system's integrity. In this project, we will propose a system that is resilient to adversary attacks aimed at deleting or altering data. Like many others, the system implements key-value storage (also known as distributed hash tables). A verification mechanism is introduced to improve the resilience of the network, which is absent in other similar systems. The system was divided into Keeper (storage) and Verifier (maintaining integrity) nodes.

Version 1 was a proof of concept focusing on introducing the main ideas, whilst in this second iteration we are refining them and building on top. The main changes are:

1. Decentralizing the Verifier node and its responsibilities
2. Building the Verifier logic on top of the Kademlia network
3. Unifying the Keeper and Verifier nodes into a single logical node
4. Joining the network is gated by a crypto puzzle
5. Verification is more efficient thanks to Proof of Retrievability

2 INTRODUCTION

Peer-to-peer (p2p) networks are distributed systems where there is no central authority. A lot of the proposed networks implement a Distributed Hash Table (DHT), which is the basis for persistent file storage.

Such networks are considered resilient to attacks because they are distributed and replicate the stored data. However, some attacks can disrupt the integrity and storage guarantees of a decentralized storage system. Such attacks can be classified as:

- Malicious peer deleting a stored file
- Malicious peer modifying a stored file
- Malicious peers overloading the system with requests (DDoS)
- Malicious peers joining the network in bulk, aiming to disrupt the established connections/topology
- A query dying in a malicious subnetwork
- Malicious peers performing a man-in-the-middle attack
- etc.

Most of these attacks have been addressed and there exist mechanisms to fend them off. S/Kademlia[**skademlia**] addresses the majority of such attacks, by building security mechanisms on top of Kademlia[**kademlia**]. In particular, S/Kademlia makes the following contributions:

- Sybil Attack Prevention - preventing malicious peers from joining the network in bulk
- Secure Communication - encrypting the communication between peers
- Improved Node Authentication - using digital signatures to authenticate peers
- etc.

In this project, we are focusing on attacks that cannot be controlled via communication over the network, i.e., attacks that happen on the peers themselves — modifying/deleting stored data. These attacks are possible because the peers in the network are usually not controlled by an authority, as is the case in centralized storage systems. We attempt to prevent such attacks by continuously monitoring the stored files in the system.

Version 1 of the project focused on combining the decentralized storage from p2p systems with a centralized authority — a single Verifier node, which performed verification. This centralized authority is a bottleneck to the scalability and goes against the idea of a decentralized network. This Verifier kept a ledger of information on which peer (IP address) stored what file, and on a given period it would contact that peer and verify that the file was intact.

In Version 2, we are moving away from directly communicating with peers over IP and are instead moving that communication to it using the Kademlia network. By doing this, the Verifier would now become just another peer. This allowed us to merge the Verifier and Keeper nodes. Now each Keeper runs a Verifier subprocess. We discuss this in more detail in Chapter **??**. We are still going to refer to the nodes with Keeper or Verifier, depending on which logical part we are referencing, but they are combined into one peer at the end. To add to this, we are also distributing the responsibilities of the Verifiers, by making each one responsible only for a portion of the stored files. This leaves the file catalog as the only centralized part of the system.

Having too many peers join the network rapidly might destabilize it in case many of these peers are malicious. To avoid this, we are introducing a cryptographic puzzle, which must be solved before joining the network. This restrains joining the network behind a time-consuming CPU procedure. Such a procedure can stop Sybil attacks on the network.

Version 1 made use of simple hashing to verify the availability of files. This relies on honest peers, that would rehash the file each time, and not just store the hashed value once and return it in subsequent queries. This proof-of-concept procedure does not fare well with malicious peers in reality. Therefore, in Version 2, we are switching it out for a Proof of Retrievability[**porfirst**] protocol. Proof of Retrievability protocols allows the client to store a small amount of metadata and perform many unique queries for the availability of a file on the server. While the protocol is developed for a client-server architecture, it can be expanded to work in a peer-to-peer network if we imagine that the Verifiers are clients and the Keepers are servers. Replacing simple hashing with a Proof of Retrievablity protocol allows us to detect malicious peers with higher probability.

3 RELATED WORK

3.1 PROOF OF RETRIEVABILITY

Proof of Retrievability (PoR) is a fairly new concept introduced in 2007 Provable data possession at untrusted stores [**porveryfirtst**]. PoR is a cryptographic technique ensuring that a stored data object in a distributed system can be efficiently and verifiably retrieved, assuring data integrity and availability. Later in 2009 "Proofs of retrievability: Theory and implementation" [**porfirst**] provides a neat formal description of the field, that a lot of following papers use. There are 2 state of the art PoR techniques:

- From 2020, Dynamic proofs of retrievability with low server storage [**poralgebra**] — based on an algebraic approach, involving matrix multiplication.
- From 2022, Efficient Dynamic Proof of Retrievability for Cold Storage [**pormerkle**] — based on a purely cryptographic approach. This paper achieves the best bandwidth, client storage, and complexity overall, however, its implementation is incredibly complex.

While PoR is not exactly intended to be used in p2p systems, it fits the use case of this project nicely.

3.2 PEER-TO-PEER NETWORKS

Many decentralized storage systems have been proposed and implemented. Modern distributed storage systems achieve up to logarithmic time for insert, lookup, and delete operations. To name a few such systems — Chord [**chord**], Pastry [**pastry**], and Kademlia [**kademlia**]. The main difference between these systems is how they organize the network nodes. Chord and Pastry use a ring structure, while Kademlia uses a binary tree. Kademlia is more resilient to churn and performs faster lookups, but generates more network traffic [**kadvschordvspastry**]. Most of these systems serve as a foundation for applications that implement some decentralized storage, but we won't go into detail about them, because they are not the focus of this project.

There are also multiple proposals on how to make decentralized storage systems more resilient to malicious peers. To name a few — ARA [**ara**], Freenet [**freenet**], and S/Kademlia [**skademlia**]. The main idea behind these proposals is to have some kind of auditing mechanism that checks if the peers are behaving correctly. ARA proposes a system where peers allocate "credits" to other peers they communicate with. If peer A requests files from another peer B, B gains "credits", while A informs other neighboring peers that B has received "credits" from A. Peers also share information about these "credits" with other peers and use them as an auditing mechanism. S/Kademlia focuses on node authentication with expensive node ID generation and verifiable messages using public and private key cryptography. S/Kademlia also routes requests through multiple different nodes at the same time to reduce the chance of a request falling into a malicious subnet. Freenet's approach to security is to encrypt all data and route requests through multiple nodes. It works in a way similar to Tor — nodes in the path to the target cannot tell what other nodes are in the path, except for the previous and the next node. Nodes storing the data are also obscured in the response, so the source cannot be tracked.

4 ARCHITECTURE & IMPLEMENTATION

We will look into the details of each component of the system and how they are implemented. Then we will discuss the higher level architecture of the system and how the components interact with each other.

4.1 PROOF OF RETRIEVABILITY

Proof of Retrievability[**porfirst**] (PoR) is a protocol that allows a client to verify that a server is storing a file correctly. The client can do this without downloading the entire file. While this protocol is meant to be used in a client-server setting, we can use it in a peer-to-peer setting. The clients are the Verifiers and the servers are the Keepers.

Some PoR protocols allow file updates or even partial file updates, however we do not use these features. The update procedure is often expensive and complex. Instead, we can use Kademlia to store the file and update it if necessary. The only part we are changing from Kademlia is the way the file is stored. Instead of storing the file directly, we perform a preprocessing step, providing us with the metadata needed for the PoR protocol. Then we store the metadata in the ledger/catalog and the file in the DHT. Later we can use the metadata to verify the file in the DHT.

This is not secure once again since the Keeper and Verifier nodes are conjoined. PoR protocols are designed to be secure against malicious servers, but not malicious clients. In our case, a malicious Keeper can use their Verifier to retrieve the metadata from the ledger and break the PoR protocol. However, modern PoR protocols propose a public auditability feature that allows anyone to verify the file. This would make the protocol secure against malicious Keepers. We do not implement this feature in our system, because of the complexity it adds to the protocol. We leave it to future work.

The PoR protocol we are using is based on the one described in Dynamic proofs of retrievability with low server storage[**poralgebra**]. The full protocol is described in the paper, so we will only describe some important parts. The protocol treats the input file as a matrix of bytes with n rows and m columns. The parameter n is related to the security of the protocol as well as the size of the metadata that the client needs to store. The parameter m is related to the size of the messages that need to be exchanged during the verification/audit step. To balance the security and the efficiency of the protocol, we need to choose the parameters n and m carefully. In practice, it makes sense to choose them such that $n \approx m$.

The procedure of the protocol is as follows:

`Init`

		Server	Client
[sharp corners, colback=white,]	**Input**		$\underaccent{\tilde}{M}$
	Output	$st_{\underaccent{\tilde}{S}}$	$st_{\underaccent{\tilde}{C}}$

[1] $\underaccent{\tilde}{M}$ is the input file treated as a square matrix, padded with empty bytes if needed **Client** sets $t := \left\lceil \frac{\lambda}{\log_2(q-1) - \log_2(n)} \right\rceil$ q is the size of the field we work in (in practice a number close to 2^{64})
In practice we choose $t \approx n \approx m$ **Client** chooses a random vector $\underaccent{\tilde}{u} = [u_1, u_2, ..., u_t]^T \xleftarrow{\$} (\mathbb{F}_q^*)^t$ with non-zero entries **Client** computes the matrix $\underaccent{\tilde}{U} \in \mathbb{F}_q^{t \times n}$ by setting the entries $\underaccent{\tilde}{U}[i, j] := (u_i)^j$

$$\underaccent{\tilde}{U} = \begin{pmatrix} u_1 & u_1^2 & \dots & u_1^n \\ u_2 & u_2^2 & & u_2^n \\ \vdots & & \ddots & \vdots \\ u_t & u_t^2 & \dots & u_t^n \end{pmatrix}$$

Client computes the matrix $\underaccent{\tilde}{V} := \underaccent{\tilde}{U}\underaccent{\tilde}{M} \in \mathbb{F}_q^{t \times m}$ **Server** saves state $st_{\underaccent{\tilde}{S}=(\underaccent{\tilde}{M})}$ **Client** saves state $st_{\underaccent{\tilde}{C}=(\underaccent{\tilde}{u},\underaccent{\tilde}{V})}$

`Audit`

		Server	Client
[sharp corners, colback=white,]	**Input**	$st_{\underaccent{\tilde}{S}}$	$st_{\underaccent{\tilde}{C}}$
	Output		π

[1] **Client** chooses a random, non-zero challenge $x \xleftarrow{\$} \mathbb{F}_q^*$, computes the vector $\underaccent{\tilde}{x}^T := [x^1, x^2, ..., x^n]$ and sends $\underaccent{\tilde}{x}$ to **Server** **Server** computes $\underaccent{\tilde}{y} := \underaccent{\tilde}{x} \in \mathbb{F}_q^k$ and sends $\pi = (\underaccent{\tilde}{x}, \underaccent{\tilde}{y})$ to **Client** On receiving the audit transcript $\pi = (\underaccent{\tilde}{x}, \underaccent{\tilde}{y})$, the **Client** computes the matrix $\underaccent{\tilde}{U} \in \mathbb{F}_q^{t \times k}$ by $\underaccent{\tilde}{U}[i, j] := (u_i)^j$ and checks whether $\underaccent{\tilde}{U}\underaccent{\tilde}{y} \overset{?}{=} \underaccent{\tilde}{V}\underaccent{\tilde}{x}$

The original paper [**poralgebra**] has precise complexity bounds, but for our purposes, we will approximate them, as the overhead is mostly negligible:

- The server stores $O(N)$ data, where N is the size of the input file
- The client stores $O(\sqrt{N})$ data
- The bandwidth of `Init` is $O(N)$
- The bandwidth of `Audit` is $O(\sqrt{N})$
- The computation for the Server during `Audit` is $O(N)$

- The computation for the Client during `Audit` is $O(\sqrt{N})$

We have implemented the PoR protocol, closely following the paper and the implementation by the authors. This choice was made, so we can compare our results with theirs and ensure that our implementation is correct. It is important to note that the original implementation uses the Mersene Twister random number generator. This is not a cryptographically secure PRNG, so we have replaced it with the ChaCha20 PRNG[**chacha**].

Other minor differences are:

1. Unlike the original implementation, we do not parallelize the multiplication of the matrices, which makes our implementation slower. This is something we leave for future work.
2. We have to use more modulo operations to ensure the calculations do not overflow. The original algorithm sometimes allows overflows, but these are not allowed in Rust. If an overflow happens in Rust, the program will crash.
3. We are hard-coding the values that the original hard-coded as well, which are needed to argue about the security.
4. Instead of relying on custom read functions, we pad the file with empty (zero) bytes to enable proper alignment. This is done only when calculations are performed and never affects the file.

4.2 LOCAL STORAGE

The default Kademlia implementation of the storage is an in-memory storage. This means that the data does not persist and is lost when the program (peer) is closed. This is not suitable for our use case, because we want to store the files persistently, even between peer restarts. To achieve this, we have implemented a local storage module, which uses the file system to store the files. The module is implemented to adhere to the Kademlia storage interface, so it can be used as a drop-in replacement for the in-memory storage. Underneath, it uses the `object_store` crate, which provides an S3-like interface for storing objects. This allows us to easily switch between local storage and cloud storage.

We had to implement a wrapper around the `object_store` crate for two reasons. First, we wanted to be able to store and retrieve files that adhere to a custom structure. Second, while Kademlia is asynchronous, it requires a synchronous storage interface. However, the `object_store` crate is asynchronous, so we had to wrap it in a synchronous interface. This posed difficulties because the `object_store` was also shared between threads. The solution was to use a reference-counted pointer wrapped in a mutex to the `object_store`. This allows us to clone the pointer and lock the mutex when we need to use the `object_store`. The mutex is locked throughout the whole operation of reading/writing a file. This is a downside of the implementation because it means that only one thread can read/write a file at a time. However, in practice, multithreaded access to local storage is slow and does not provide any benefits.

We are saving the files named by their unique identifiers. This allows us to quickly check if a file is stored at a peer by checking if the file exists, without keeping a separate index.

Kademlia has a concept of a Record and not a file. A Record is a key, a value, a publisher, and an expiration time. Since we are working in Rust, we cannot dump the memory of a struct to a file and read it back. Instead, we have to serialize the Record to a string or a byte array and then deserialize it back. We have opted to use YAML as the serialization format because it is human-readable and easy to debug. The serialization and deserialization are done using the `serde` crate, which provides a framework for serializing and deserializing Rust data structures efficiently and generically. The one downside is that in Rust, you cannot implement a trait (interface) for a type that is not defined in your crate/project. This means that we cannot implement the `serde` traits for the `Record` struct, because it comes from the `kademlia` crate. To resolve this, we have created a wrapper struct around the `Record` struct, which implements the `serde` traits. Serializing the key and publisher is not straightforward, but both are byte arrays, which can be converted to strings, and strings are serializable. The problematic part is the expiration time, which is an `Instant`. This is a struct that represents a point in time, but it is not serializable, because it is not convertible to a timestamp. It is meant to be used as an opaque type, which is only used for comparisons. To resolve this, we are sacrificing part of the precision of the expiration time, by converting it to a timestamp manually. While this is not an ideal solution, in practice it will have only a few milliseconds of error, which is acceptable. The value is a byte array, which we serialize by encoding to base64. While this is not the most efficient way to serialize the value, it is the easiest to implement. In a future version, we would separate the value from the rest of the record and store it as a binary file.

4.3 CRYPTOGRAPHIC PUZZLE

The cryptographic puzzle is a proof-of-work algorithm that is used to prevent Sybil attacks. Before a peer can join the network, they need to generate a peer ID. During this step, they are required to solve a cryptographic puzzle. Our implementation is in two steps. First, we generate a random peer ID using an ED255519 elliptic curve algorithm. Then we hash the peer ID using SHA-3 and expect the hash to start with a certain number of zeros. The number of zeros is a parameter of the puzzle and is related to the security of the puzzle. If the hash does not start with the required number of zeros, we generate a new peer ID and try again.

This is easily verifiable by other peers because they can hash the peer ID and check if the hash starts with the required number of zeros. This is also a computationally expensive task because the only way to find a peer ID that satisfies the puzzle is to generate random peer IDs and hash them until we find one that satisfies the puzzle. We will discuss the number of zeros in the puzzle in the evaluation section.

4.4 DECENTRALIZED VERIFIER

The decentralization of the Verifier is by far the most impactful change we have made to the system. In Version 1, the Verifier was a centralized entity that was responsible for verifying the files. This was a single point of failure because if the Verifier were down, the system would not work. Also, it did not make sense for a decentralized system to have a centralized entity (authority). In Version 2, we are moving towards a decentralized Verifier. The Verifier is now a logical part of every node in the system.

To achieve this, we merged the verifier module with the keeper module. In the previous version, the Verifier contacted the Keeper nodes via GRPC over TCP. Now the verifier can access the Kademlia interface and interact with peers via the Kademlia protocol. This means we had to rework the Contracts for storing a file, which kept track of the Keeper's IP address, but now only keep track of the Keeper's peer ID in the network.

In the implementation of Kademlia we are using PUT queries to store the file at the peer that received the query. To store the file with other peers and achieve replication, we need to send PUT queries to other peers, which are called PUT_TO queries. There is a slight issue with this query because it reuses the response of the PUT query, which does not contain the peer IDs of the peer where the file is stored. To circumvent this issue, we are making use of the GET_CLOSEST_PEERS query, which returns the peer IDs of the k closest peers to a given key. We then issue PUT_TO queries to the k closest peers and note down which peers have stored the file. Now we have a list of peers that have stored the file, with which we can create the Contracts and, later on, use them to verify the file. The final thing that allows us to use this approach is the fact that the GET query returns the peer ID that answers the query. This allows us to issue verification requests using the GET query to the peers that have stored the file, and get information about which peers answer the query.

5 EVALUATION

We are going to compare the new or changed components of our system with the old ones.

5.1 PROOF OF RETRIEVABILITY VERIFICATION

The trivial verification method is to download the whole file and compare it against the original. This defeats the point of distributed storage, but it can be optimized by comparing the hash of the file instead of the file itself. This method requires too much bandwidth ($O(N)$) and is therefore slow, but it is a 100% accurate and requires only $O(N)$ computation (where N is the size of the file) on the client side and $O(1)$ space for the hash.

Can we do better? We can precompute multiple hashes from the file concatenated with a random string (salt) and store these hashes together with the salts. Then, we for each verification we can send the salt, then the storage node sends back the hash of the file with the salt, and finally we compare the hash to the one we have precomputed. This method requires $O(1)$ bandwidth and $O(N)$ computation on the client side (the cost of precomputing each hash) and $O(1)$ space for the hashes and salts.

Keeping the bandwidth low is important, because it is the slowest part of the verification process. The computation can be usually parallelized and make use of data locality to be much faster than network communication. This is where our current implementation lies. The Proof of Retrievability protocol we use requires $O(N^3)$ reducible to $O(N^{2.37287})$ [**matrixmultiplication**] for matrix multiplication during the precomputation step. The verification step requires $O(N)$ server computation and $O(\sqrt{N})$ client computation, as well as $O(\sqrt{N})$ bandwidth. The space requirement is $O(N)$ for the server and $O(\sqrt{N})$ for the client. These numbers can be further improved with a better Proof of Retrievability protocol [**pormerkle**]. This is marginally better thanks to the lower bandwidth and storage required on the client. We would like to argue that these are the two most important resources because of the goals of the client - they want to store their data in the system, i.e., they want to use as little space as possible, and they want to be able to verify their data as fast as possible, i.e., they want to use as little bandwidth as possible. It does not make sense to compare algorithms with different CPU and bandwidth requirements because the network communication can either be really fast or really slow, depending on the network conditions, distance between peers, etc. Therefore, we

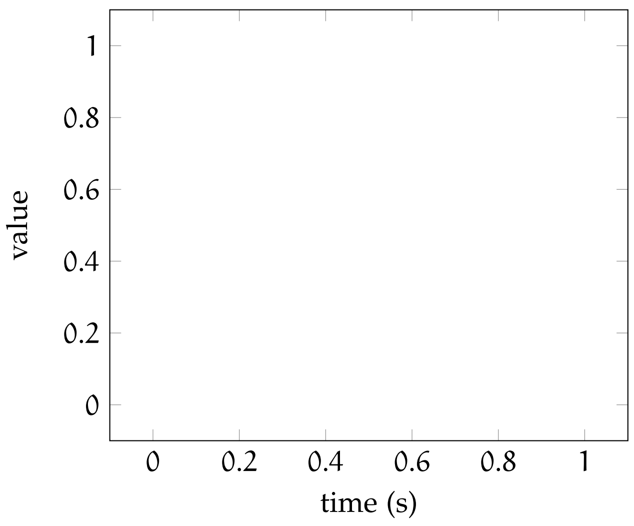

Figure 1: Generating figures directly in LaTeX is possible, and they can be very consistent stylewise with the rest of the document.

TIME	VALUE	σ%

Table 1: This table has been generated from a `.csv` file, which sometimes can be very handy and a great timesaver. Note, how numbers have been normalised and aligned properly.

will only present results for the time required to initialize a file and verify it on a local machine to give an idea of the performance of the system.