DECENTRALIZED STORAGE SYSTEM WITH VERIFICATION

(KISS)

Ivan Luchev, 724727

DECENTRALIZED STORAGE SYSTEM WITH VERIFICATION

May 2023

Advisor: Niels Olof Bouvin

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

1

ABSTRACT

KISS is a decentralized storage system with verification mechanisms. The main goals of this system are to ensure durable replicated storage and to prevent or at least limit malicious peers' influence on the network. In particular, attacks that aim to disrupt the integrity and storage guarantees of a decentralized storage system. The system implements a key-value storage (aka. a distributed hash table) built on top of Kademlia in order to provide logarithmic time for querying. The nodes in the network are split into two kinds - one strictly storing files, and another taking care of indexing and validation. The verifier nodes keep track of what files should be stored in the system and periodically verify that these files actually exist. In other words the verifier nodes are the super-users of the system that keep track of which nodes behave correctly and need to be rewarded or are malicious and need to be punished. This reward-and-punishment mechanism is what keeps the system in check.

2

INTRODUCTION

3 ARCHITECTURE & IMPLEMENTATION

3.1 ARCHITECTURE

The system is split into 2 components - Keeper and Verifier. Keepers implement the Kademlia protocol and store the actual data, while Verifiers keep track of what data should be stored and periodically verify that the data is still there. In the current version there is one Verifier for simplicity reasons, but in the future the Verifiers will be part of a p2p network as well.

The Keepers communicate between each other using the Kademlia protocol. Kademlia is the backbone of the Keepers as it implements the peer discovery and routing. All Kademlia communication is event-based and asynchronous. In order to keep track of the Kademlia messages the Keepers use a local Hash Table.

The Verifiers communicate with the Keepers using GRPC. This protocol was chosen because it is language agnostic and fast. The Verifiers also use GRPC to communicate with the database - ImmuDB.

ImmuDB is an immutable database that uses Merkle trees to ensure data integrity. The Verifiers use ImmuDB to store metadata about the files stored in the system. The metadata includes the file name, and the hash of the file's content and the expiration date of the file. The expiration date is used to determine when a file should be deleted from the system. Deletion is implemented as a soft deletion - the Verifiers simply stop checking for the file's existence.

There was no library that implemented an API for ImmuDB in Rust and ImmuDB did not document how to create one, so we had to reverse engineer the python library and implement a Rust library. The Rust library is very limited as it implements only the necessary endpoints for this project. The library is fairly simple as it performs a login request, which returns a token and then every subsequent request uses this token to authenticate itself.

3.2 IMPLEMENTATION

The system is implemented in Rust. The main reason for this is that Rust is a systems programming language, which means that it is fast and has a low memory footprint. It also provides a lot of safety guarantees, which is important for a system that needs to be resilient to malicious peers.

The common modules of the project containing constants and errors are separated in a separate library crate. The custom errors are

implemented using the error_chain crate. It provides a useful macro that allows to define custom errors with a description and a display message.

```
error_chain! {
    errors {
        ConfigErr(e: ConfigError) {
            description("loading config failed"),
            display("loading config failed: {}", e),
        }
    }
}
```

This allows the custom errors to encapsulate any other errors from other crates.

Both the Keeper and the Verifier follow the same code architecture - asynchronous main with dependency injection relying on yaml settings.

The main function looks similarly to:

```
async fn main() {
    match run().await {
        Ok(()) => info!("shutting down"),
        Err(err) => die(err), // prints the error and exits
    }
}
async fn run() -> Res<()> {
    env_logger::init(); // initialize the logger
    // initialize the dependency injector
    let injector = dependency_injector()?;
    let grpc_handler: Svc<dyn IGrpcHandler> = injector.get();
    let kad: Svc<dyn ISwarm> = injector.get();
    // start the services asynchronously
    try_join!(grpc_handler.start(), kad.start())?;
}
```

The logger has to be initialized before any other code is executed, because the other code might use the logger. The dependency injector is initialized next. It lazily initializes the modules when one is requested for the first time, and caches the result. For example, GrpcHandler depends on the Settings module, so Settings will be initialized and then passed to GrpcHandler. Finally, we start all modules and wait for them to error out. The start methods are pretty much infinite loops that act as event listeners.

```
pub fn dependency_injector() -> Res<Injector> {
    let mut injector = Injector::builder();
    injector.add_module(p2p::module());
    injector.provide(
```

```
        SettingsProvider
            .singleton()
            .with_interface::<dyn ISettings>(),
            // Settings is an interface
    );
    injector.provide(
        KeeperGatewayProvider
            .singleton()
            .with_interface::<Mutex<KeeperGateway>>(),
            // No interface and no good concurrency :(
    );
    ...
}
```

The dependency injector adds all dependencies as singletons. This means that only one instance of each dependency will be created. This is good because we ensure no different instances will try to access the same resource and cause a race condition. But it also causes a huge pain to initialize some dependencies. For example, the KeeperGateway module in the Verifier, which is responsible for communicating with the Keepers over GRPC, contains a Mutex<KeeperGrpcClient<_> >. This is needed because the GRPC client does not implement the necessary traits to be thread-safe and mutable. Rust is very pedantic about having mutable variables being accessed from different threads, so it disallows the GRPC client to be modified without a mutex. So we have to wrap the GRPC client in a mutex, so we can provide the KeeperGateway as a singleton dependency. But, if we want to take control of a mutex and modify the data inside, we need a mutable reference to it. This means that the whole KeeperGateway needs to be mutable, i.e., provided in a mutex. However, providing a module surrounded in a mutex breaks the idea of providing an interface, because the mutex does not implement the KeeperGateway's interface. We also cannot provide mutex surrounding an interface, because the interface does not have a size known at compile time.

tl;dr; Rust is very pedantic about mutability and thread-safety, which is good, but it makes it very difficult to use dependency injection in a multithreaded (async) environment. We have to encapsulate modules, that are not implemented as thread-safe, into mutexes, which makes them not implement the necessary interfaces, and that breaks the dependency injection idea.

Switching to something simpler - the Settings module reads off the settings from a YAML config file. The Settings utilize the serde crate to deserialize the YAML file into a struct. The config file specifies any variables that change between instances - for example the port number and any secrets. Nothing fancy happens here.

The storage is pretty simple as well. It uses the object_store crate, which provides the necessary APIs to work with HDD storage or S3 storage.

The GRPC module is implemented using the tonic crate. It is very well documented and explains in details how to implement a GRPC server and client. It boils down to writing a protobuf definition and then tonic generates all the boiler-plate code, leaving the developer to implement only the interface methods for the server and the client.

Another pretty hard to implement module is the Kademlia module. The module itself is not hard to implement, but any interaction with the module is painful to say the least. Kademlia is implemented with 1 field - Mutex<Swarm<CombinedBehaviour> >. It has to be a mutex again, because this is the only way to provide it as a singleton, and we need to have mutable access to the Swarm, so we can read events from it. Where's the problem? Kademlia is initialized as an infinite loop that listens for events from the Swarm. This means that we have to take a mutable reference to the Swarm and then pass it to the infinite loop. Yes. The mutex is now locked, and we cannot access it from anywhere else. The solution was to make a separate module, that - SwarmController, which initializes a tunnel between itself and the Kademlia module, that it uses to send messages to the Kademlia module. i.e., These messages are the getters and setters of the Kademlia module. And they are asynchronous, because we do not know when the Swarm will be unlocked or will not be processing Kademlia events and will be ready to be used. Here is how this looks in code:

```
async fn start(&self) -> Res<()> {
    let mut swarm = self.inner.lock().await;
    let mut receiver = self.swarm_controller.lock().await;
    loop {
        select! {
            instruction = receiver.recv() => {
                // handle controller (getter/setter) event
            },
            event = swarm.select_next_some() => {
                // handle kademlia event and
            }
        }
    }
}
```

The select! macro allows us to wait for 2 blocking events and whenever 1 happens to execute the corresponding code. Since this happens inside the Kademlia module, we can safely take ownership of the swarm (make it mutable). This allows us to use the swarm and handle any Kademlia events at the same time.

The most excruciating part of the communication is how exactly the SwarmController communicates with the Kademlia module. I will try

to put it into words, but it is very difficult to wrap one's head around it. Here goes.

The SwarmController shares a two-way channel with the Kademlia module. When someone calls the SwarmController getter a message is sent to the Kademlia module indicating that a getter was called. Kademlia does its p2p magic and whenever it receives a response from the Swarm it has to send back a message. In order to achieve that in a non-blocking way the Kademlia module keeps a hash table with the IDs of the requests made to the swarm together with the way to send back a response.

Here's what the code in the SwarmController looks like for a getter:

```
async fn get(&self, key: String) -> Res<Bytes> {
    // Create a channel<channel>
    let (sender, receiver) = oneshot::channel::
        <OneReceiver<Res<Bytes>>>();
    self.swarm_api .send(SwarmInstruction::Get { key, resp: sender });
    // Get the channel where the response will be sent
    let receiving_channel = receiver.await.unwrap();
    // Get the actual response
    let result = receiving_channel.await.unwrap();
    result
}
```

We send a channel containing a channel to the Kademlia module. Next, the Kademlia module accepts the request and sends back a channel where it will send the response when it receives it from the Swarm. Kademlia also saves the channel where the response will be sent to the SwarmController together with the ID of the request sent to the swarm in its hash table. This happens in this piece of code:

```
async fn handle_controller_get<'t>(...) {
    let key = Key::new(&key);
    let (sender, receiver) = oneshot::channel::<Res<Bytes>>();
    // Send the channel receiver where the response will be
    // sent to the SwarmController
    resp.send(receiver).unwrap();

    let query_id = swarm.behaviour_mut().kademlia.get_record(key);
    // Add ID:sender-channel to the hash table
    self.queries.insert(query_id, QueryResponse::Get { sender });
}
```

Finally, when the Kademlia module receives a response from the Swarm it sends it back to the SwarmController over the channel, which was saved in the hash table in the previous code snippet. Here's what the code looks like:

```rust
async fn handle_get_record(&self, message: GetRecordResult, id: QueryId) {
    let response_channel = self.queries.remove(&id);

    match message {
        Ok(GetRecordOk::FoundRecord(PeerRecord {
            record: Record { value, .. },
            ..
        })) => response_channel
            .send(Ok(value))
            .expect("swarm response channel closed"),
        ...
    };
}
```

I felt the need to put in code snippets, even if they are mostly pseudo-code, because it is very difficult to explain the communication between the modules in words. We send a channel, containing a channel, over a channel. This is the only way to ensure we won't deadlock and keep the whole communication asynchronous.

4

RELATED WORK

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 [8], Pastry [6], and Kademlia [**kademlia**]. The main difference between these systems is the way they organize the nodes in the network. Chord and Pastry use a ring structure, while Kademlia uses a binary tree. The ring structure is more efficient in terms of routing, but it is more difficult to maintain the ring structure when nodes join and leave the network. Kademlia is more resilient to churn, but it is less efficient in terms of routing. Most of these systems serve as a foundation for applications that implement some kind of 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 [4], Tarzan [3], and S/Kademlia [2]. 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 share information about what data they are storing and check each other. Tarzan focuses on anonymity of peers in the network by generating fake traffic and hiding the real traffic. 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 in order to reduce the chance of a request falling into a malicious subnet.

Comparison between performance between decentralized storage systems has been done before [7] & [1]. The results are that Pastry is the best performer for networks with under 1000 peers, but the paper also suggests that Kademlia would probably outperform Pastry for larger networks. The results in the 2 papers differ somewhat from the results in [5], which suggests that theoretically Chord should outperform Kademlia, but the paper does not do in-depth evaluation of networks with high churn rate like [7].

5 DISCUSSION

The aim in this project is to build resilience against malicious peers, and not invent a new decentralized storage system. Since we want to build a network to withstand disruptions, we build this project on top of Kademlia. This means that in terms of performance of the storage operations, the system in this project will perform the same as Kademlia.

In this version of the KISS system we will not focus on anonymity of peers or node authentication. Hence, we will not compare the system to Tarzan or S/Kademlia. This will be a future work.

We can compare the approach in this project and the ARA proposal [4]. Both KISS and ARA rely on auditing to detect malicious peers or degradation of the network. Peers in ARA share information about what data they are storing and check each other. This is based on a list of interested peers (peers that interacted recently with a given peer). An issue with this approach would be if a subnet is composed entirely of malicious peers. Then a peer that has only malicious peers can freely adjust its credit score. Another problem would be generation of fake requests in order to increase the score of malicious peers, that store data for these requests. Peers in the KISS network are not as "smart" and do not have such responsibility. Instead, the audit is performed by an outside entity. This entity might be another decentralized network or a centralized authority. This delegates the complexity of keeping the network in check and shifts the problem elsewhere. Currently, KISS uses a centralized Verifier, but in the future it will be possible to use a decentralized network. In this case it won't suffer from malicious subnets, because the Verifiers will be rotated and are not necessarily peers that have recently interacted with a given Keeper. Also, since answering requests doesn't contribute to a peer's reputation there is no reason to flood the network with fake requests. It doesn't mean that a DoS attack against the network is impossible, but it will not affect the reputation/trust of the peers. A strategy to prevent a DoS attack can be deployed separately from the reputation system, and it can work independently without taking into account what is the underlying network.

6

EVALUATION & CONCLUSION

A

AN APPENDIX

BIBLIOGRAPHY

[1] Mohammad Asyraff Mohamad Ariff, Suraya Mohamad, and Ahmad Roshidi Amran. "A review of recent advancement in Kademlia and Chord algorithm." In: *5th International Conference on Green Design and Manufacture (IConGDM 2019)*. Vol. 2129. American Institute of Physics Conference Series. July 2019, 020131, p. 020131. DOI: 10.1063/1.5118139.

[2] 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.

[3] Michael Freedman and Robert Morris. "Tarzan: A Peer-to-Peer Anonymizing Network Layer." In: *Proceedings of the ACM Conference on Computer and Communications Security* (Oct. 2002). DOI: 10.1145/586110.586137.

[4] MyungJoo Ham and Gul Agha. "ARA: a robust audit to prevent free-riding in P2P networks." In: *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*. 2005, pp. 125–132. DOI: 10.1109/P2P.2005.2.

[5] Lacine KABRE and Telesphore Tiendrebeogo. "Comparative Study of can, Pastry, Kademlia and Chord DHTS." In: *International Journal of Peer to Peer Networks* 12 (Aug. 2021), pp. 1–22. DOI: 10.5121/ijp2p.2021.12301.

[6] 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.

[7] Rafiza Ruslan, Ayu Zailani, Hidayah Zukri, Nur Khairani Kamarudin, Shamsul Jamel Elias, and R.Badlishah Ahmad. "Routing performance of structured overlay in Distributed Hash Tables (DHT) for P2P." In: *Bulletin of Electrical Engineering and Informatics* 8 (Mar. 2019). DOI: 10.11591/eei.v8i2.1449.

[8] 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.