KISS

Kooky and (maybe) Innovative decentralized Storage System

The high level idea

The idea is to create a distributed storage system based on a structured p2p network. The main goals of the system are to ensure durable storage by replication. The system will also try to prevent some of the more popular attacks by using a blockchain network, which keeps track of peers' reputations.

The environment

The system must run on a Unix machine. Linux is required, Mac is a nice-to-have. Supporting windows is not within scope of this project. If possible the whole system should run inside a Docker container, so it can be run on Windows as well, but since we might need to use low level primitives to interact with the hard drive/network interfaces this cannot be guaranteed.

Dictionary

- Keeper (node): a peer in the network storing files
- Validator (node): a peer in the network validating the integrity of stored files by Keeper nodes
- Gatekeeping: The ability of Keeper nodes to decide which files to store, based on whom the client storing them is
- Contract: Agreement between a client and the system to store a file for a certain period of time

Programming language

The language of choice is Rust. It is preferred due to code style and speed, that is comparable to C/C++ (Source). We will not go into details why Rust is better to alternatives as these can be found in numerous articles online. The downside of Rust will be the development velocity.

A viable alternative is Golang, which will increase development velocity, but will decrease performance of the application. Go also has more libraries than Rust, but these might not be relevant for the project.

Keeper nodes

Keeper nodes store files in the system. They contribute to the network with their hard drive and network bandwidth.

How will Keeper nodes store files? - the interface

Files should be stored on the Keeper's hard drive. This is a suggestion and not a hard requirement. The only requirement is that the Keeper can provide the stored file when prompted for it. The reason behind this will become clear when we discuss different storage solutions.

Option 1: Implement a simple disk file storage module

Pros

- Short code = smaller binary size

Cons

- Simple implementation means it will be harder to extend this implementation if we want to store files in a slightly different way e.g. on a NAS/in a database
- It would be hard to implement an algorithm for efficient writing to disk

(Preferred) Option 2: Use the object_store lib

The object_store Rust library implements interfaces for object stores. Examples for object stores that are supported include AWS S3, Azure Blob Storage, and Local files. While S3 is proprietary PaaS, the Azure stack can be installed locally. Also, the Local files implementation uses the hard drive of the machine.

Pros

- Provides a single interface for multiple storage solutions
- Allows for easier testing by easily swapping the implementation of the interface for example from Azure to a local file system one

Cons

- If the storage solution doesn't have an implementation for its interface we'll have to implement it manually

This option is preferred because not only does it provide interface for the local filesystem, but there are database solutions, which implement the S3 interface, which would make them compatible with this library.

How will Keeper nodes store files? - the underlying data store

An obvious solution is to use the machine's hard drive directly, but we should explore storage solutions for blobs as they might provide useful features such as caching, efficient write to disk, etc.

All the solutions we'll look into provide an S3 interface. These also turn out to be the most popular solutions, which are also open source. We'll favor solutions that are easy to set up and have a bigger community.

(Preferred) minio

- 37k stars on GitHub
- Written in Go
- The general opinion is that it's easy to set up and use
- Comes as a binary or as a Docker container

ceph

- 11k stars on GitHub
- Written in C++
- The general opinion is that it's hard to set up, more suitable for enterprise solutions

riak

- 4k stars on GitHub
- Written in Erlang

cloudserver

- 1.5k stars on GitHub
- Written in JS

How to keep the stored files isolated from the rest of the system?

This choice comes down to Docker vs namespaces + cgroups.

(Preferred) Option 1: Docker

The most popular containerization system is Docker. It is also pretty much the only one that runs on Mac, Windows, and Linux. Source.

The downside of Docker is that it has overhead, and it cannot directly control the allocated hard drive space (but this can be worked around).

Option 2: cgroups + namespaces

cgroups in combination with namespaces is what other containerization systems are built on top (at least under Linux). They can limit CPU, Memory, I/O, Network, and can isolate network interfaces as well as the file system.

The upside of cgroups + namespaces is that there isn't as much overhead as with Docker. The downsides are that this approach is limited to Linux systems and is harder to implement.

Validator nodes

Validator nodes check Keepers' contracts for storing files. Validators are responsible for creating the Contracts between clients and the system for storing files, (read more in the Storing section).

Validation of file integrity and replication invariant

Files in the system must be replicated 3 times. In order to adhere to this requirement, the system must handle Keeper nodes, which drop files or leave the network. Validators will check every couple of minutes if a file is stored on all Keeper nodes, that promised to store it. If a Keeper node is inaccessible or cannot prove that they are storing the file, the Validator will store the file elsewhere.

(Preferred) Option 1: Zero knowledge proof

Ideally this will verification will be handled with a zero-knowledge proof.

Option 2: Hash trees

If this turns out to be impossible we'll have to rely on hash trees (Merkle trees).

Option 3: Complete file retrieval

Another way to verify the file integrity is to request the whole file from the Keeper node and compare it to a checksum for the file, which has been calculated at the time of adding the file to the system.

Cons

- Increases traffic in the system

Pros

- Provides cover traffic. Read more under the Attacks against integrity section

TODO: Investigate if zero-knowledge proof is possible in this case TODO: Investigate these options in more detail

Rotating Validator Keepers

In order to avoid malicious Validators and Keepers being matched, we need to rotate the relationships Validator-files.

Option 1: Random exchange

Two Validators randomly communicate and agree to exchange the files they are responsible for. This option has a downside that if there is a malicious Validator it might choose to not switch its responsibilities or to switch them with another malicious Validator.

Option 2: Move by 1

The Validators are in a structured p2p network, and they know their neighborhood. In this case we can have at a certain period of time, all validators move their responsibilities to the next Validator in the ID space. i.e. We rotate all Validators by 1.

To ensure this is happening Validators will either keep a small part of the files they are responsible for and will check with +2/+4/... Validators in ID space if they received the correct files. If they didn't - the validators in the middle will be flagged as untrusted. After a number of such flags, the untrusted Validator will be kicked from the network.

The downside of this option is that if many malicious peers connect to the network and happen to have neighboring IDs, they can exploit the system by not doing any work.

Option 3: Use a consensus algorithm

Occasionally all Validators have to agree on a number that they are going to use for rotation. e.g. if the number chosen is 3, all Validators will give their responsible files to the Validator that is 3 positions away from them in the ID space.

This makes the algorithm more unpredictable and harder for Malicious peers to exploit. The downside is the consensus algorithm. It can take time to reach consensus, it's hard to implement, and it can be compromised by malicious Validators if they are >50%. To mitigate this issue the Validators might fall back to Option 2 if too many random numbers are rejected by the consensus algorithm.

(Preferred) Option 4: Use a centralized system

All Validators can be centralized, or can be controlled by a centralized component, which has a database, where the responsibilities of each Validator are stored. This database will also have all the Contracts, so it can reshuffle the responsibilities at will.

While this option is centralized, it's much easier to implement. It is preferred because it can be used as a stepping stone and can always be expanded or swapped by any of the other options at a later time.

TODO: Think what can go wrong if the Validators misbehave. Can we validate the Validators? Then maybe we don't need Validators, but the Keeper nodes can employ that validation logic and validate each other?

Resource naming scheme

Option 1: A flat naming scheme

The scheme would be `file-name-hash`.

Users may choose their desired resource identifier, or it can be generated as a hash from the original resource name to ensure uniqueness.

The downside is that this will make it harder to distinguish which client is storing what files. i.e. Isolating files from different clients will be more difficult and not as apparent, because we'll need to rely on metadata for the file.

(Preferred) Option 2: Hierarchical naming scheme

The scheme would be `/client/namespace/UUIDv4/chunk-number`. This scheme is not fixed and can be extended by clients. The critical parts are the client name and the filename. The chunk number is the file part, it may be omitted for small files. Read more on large files in the Storing section of this document.

Using this scheme Keeper nodes can separate files into different locations and isolate them if necessary. It would also make gatekeeping easier, as the Keeper node can at any point in time choose to drop files for a certain client if that client is deemed to store malicious data.

One problem would be that the client's name is exposed. This can be solved by hashing the client name and namespace name. This means that as when querying the client and namespaces will be hashed before the query is propagated to the network.

TODO: does this make sense, are there any other downsides being overlooked

Indexing (searching for file by name/metadata)

Searching in the network will be impossible. The client is expected to know what file they are looking for. i.e. the client is responsible for keeping an index of files, metadata about them, and their identifiers (reference the naming scheme).

Querying

The Keeper nodes will be part of a structured p2p network. The replication factor of the system will be 3.

Querying for a resource will be done given its name. The first node that receives the name of the resource will hash it. The hash value will be used to determine the position of the file (if it exists) in the ID space.

Using exponential search, the current node will redirect the query to a node, which is closer in the ID space to the hash value. After at most log(N) number of steps the query will arrive at a node, which is just after the position of the hash value in the ID space. i.e. The previous node in the network will be exactly behind the hash value in the ID space. The node we arrived at should have the file if it exists. If it doesn't, we check the 2 nodes following it (since we use replication factor of 3). If a node doesn't want to store a file, they're still required to store a pointer to it (the IP/ID of the Keeper node where it's been moved).

TODO: Expand this section with more details

Storing

Files storage will be handled similarly to querying. First, a Validator node is contacted by a client, who wants to store a file. The Validator node generates the file name identifier. Then, it proceeds to query the Keeper nodes for that identifier. If a file name with the same identifier is found, the request is rejected. Otherwise, the query for the file will return the Keeper node, which should have the file. The Validator node stores the file on that Node and on the 2 nodes after it in the ID space. The Validator also stores in a ledger, metadata about the file, in order to be able to verify the file integrity at a later point in time.

How to store large files?

Large files are files with size over 100mb. Such files will be split into chunks of size 100mb. Each chunk will be saved separately, as it has a different hash of its name - `/client/namespace/UUIDv4/chunk-number`. The number of chunks will be saved as metadata when saving the file. Chunk 0 will have no chunk number. When a client wants to download the file they will retrieve chunk 0 and information that there are X other chunks with numbers [1, X-1]. Then the client can proceed to download the other chunks as well.

Peers leaving the network

Peers (Keeper nodes) keep information about their neighbors in ID space. To keep information about neighbors, peers use exponential back-off heartbeat. Once a peer fails to contact a neighbor (let's call it W) for 1 minute, they contact their other neighbors to inform them that peer W is dead. The neighbors either confirm or deny this statement. Once all neighbors confirm that the peer is dead, the neighbors redistribute the files that that peer was keeping. This way the replication factor of 3 will always be preserved.

Peers joining the network

When a peer (Keeper node) wants to join the network it contacts a node in the network, which generates the new node's ID by hashing its IP. The node then performs a query on its ID in order to find nodes close to this ID. Once at least one such node has been found, the current node can contact a fixed number of nodes in the neighborhood (3-5 close peers) and request to join the network. All nodes in the neighborhood require the newly joining peer to complete a hard cryptographic puzzle before allowing it to join. The nodes of the neighborhood exchange information between them whenever the peer solves a puzzle. After all puzzles are solved, the nodes in the network have to reach consensus that the new node is joining the network. Consensus is achieved by having all nodes acknowledge that all the crypto puzzles are solved, and all neighbors approve of the new peer.

Once a peer joins a network it starts to request files from its neighbors, which it should own. Now that the peer owns files it can start answering queries, but files stored on the peer don't count towards the replication invariant. After a peer has been active in the network and actively contributing (answering queries) for a while (24h up to 1 week), it can join the network as a permanent peer. At that point, peers from the neighborhood can start dropping files, which they shouldn't own (which are father away in the ID space). This decision for dropping files is made by the old nodes in the network, not the node that recently joined. This decision is also synchronized with the Verifier nodes.

TODO: Research cryptographic puzzles TODO: Check the E/S Kademlia algorithm hash(hash) = node ID

Blockchain vs Ledger stores

In order for the Validators to keep the Keeper nodes in check, they need to store information about which nodes are behaving properly. In order to achieve this Validators need to keep track of what contracts have been established between clients and the system. The last thing that needs to be kept track of is which Validators are responsible for which Keepers at a given point in time. Potentially, there is 1 more piece of information that can be stored - what files with what names are being stored, but this can be inferred from the contracts, which are stored anyway. Additionally, we are not handling indexing inside the system so keeping more metadata for files is meaningless.

To be able to store this kind of data - Behavior of Keepers, Contracts, and Validators state, we can employ two storage strategies - blockchain or a Ledger store.

Blockchain

A blockchain is essentially a decentralized ledger. It would be best if we can use this storage method, but it has 2 major downsides:

- Development complexity (it is harder to work with)
- Bad scalability = low number of queries/writes per second (depending on the blockchain used)

(Preferred) Ledger

Using a ledger is the centralized version of a blockchain. We get all the benefits of immutable data storage, but we lose the decentralization part. However, the pros of a ledger are that we'd get better scalability and easier development as it's not too different from a SQL data store.

The ledger options we have are mainly

- https://github.com/google/trillian
- https://github.com/codenotary/immudb

TODO: investigate which option to use, maybe immudb

Available blockchains

After a short look into these options, they all seem to have detailed development docs. Rust having more successful blockchains is a good sign for the maturity of the language in the field. It also provides alternatives if one of the blockchains turns out to be unusable for the purposes of this system.

(Preferred) Rust-based blockchains

The more popular Rust blockchains are

- Polkadot (using https://substrate.io/)
- Solana (https://solana.com/developers)
- Near protocol (https://near.org/)

Golang-based blockchains

There is 1 popular Golang-based blockchain. Algorand https://github.com/algorand

Attacks

sybil malicious peers try to get into 3 consecutive positions and exchange info about what files they have and simultaneously drop files.

- Leaving and joining the network - don't count nodes who recently joined as a part of the replication factor.

Potential attack: If a Keeper is malicious, and it starts receiving requests to duplicate its state (all their files), it might decide to disconnect from the network in order to harm the integrity.

How to avoid malicious Keepers?

A malicious Keeper is a Keeper that breaks the storage Contracts. The Validators are responsible for checking these Contracts periodically. If a Validator finds a misbehaving Keeper/disconnected Keeper, that Keeper is flagged and the Validator will prompt other Validators to check on the Keeper. If the Validators reach consensus that the Keeper is misbehaving, the Keeper is kicked from the network and loses all its rewards.

TODO: Expand on this section

How to avoid malicious Validators?

The files a Validator is responsible for will be rotated.

TODO: Can we validate the validators? Maybe a cross-validator check?

Attacks against the integrity

When a Keeper node disconnects the files it stored should be duplicated hence a lot of traffic is generated to its neighbors. This can be abused by malicious neighbors to also disconnect from the network in order to destroy the integrity. i.e. If all 3 Nodes that have a single file disconnect in short succession, that file is lost. The best way to solve this issue is with cover traffic. Alternatively we could restore the lost Node slower, but this increases the risk of the other nodes disconnecting.

Reliability

Redundancy

The replication factor of the system will be 3. i.e. Each file will be stored on 3 different nodes. The 3 nodes must be with sequential IDs.

If one of these 3 nodes doesn't want to store the file, they have to store a pointer to the file on another machine.

The Validator nodes will occasionally check for the existence of the 3 replicas of each file and if 1 or more of these replicas are down, the Validators will store it to another Keeper node.

Availability

The system should be able to achieve 99.999% uptime. This is assuming normal operation. We have not dived deep into DDoS attack prevention, but anything else should be accounted for. We're also assuming peers joining the network intend to stay in the network for at least 24 hours. Without long-lived peers the system cannot exist.

Performance

Querying should run in O(log N) time, where N is the number of Keeper nodes in the network. Queries should be optimized for proximity. i.e. Queries should be answered by the Keeper node, closest to the requester.

Integrity

We assume normal operation conditions (conditions under which the network can continue to exist). Under those conditions the network should never lose a file. This implies that if a peer goes down the Validator responsible for this Keeper should prioritize restoring the redundancy of the files that were stored on that Keeper.

Scalability

The system should be able to handle increase in traffic with linear increase in resources.

Maintainability

The components of the system should check for updates every 1h. When two peers communicate they should exchange versions and if they are on different versions, they have to check the current latest version and update themselves.

The different components of the system should be able to update themselves with downtime of less than 1 minute.

Security

TODO: improve on this section

- Preventing peers from hurting the durable storage guarantee: Peers will be required to “stake” their tokens in order to store files. File integrity will be checked randomly and if the file storage contract isn’t obeyed, the peer’s tokens will be slashed
- Preventing Sybil attacks: Peers joining the network need to solve a crypto puzzle before joining. Also, the previous point
- Eclipse attacks: The reason we are choosing a structured p2p network
- DDoS: *Unclear*

Privacy

TODO: improve on this section

- (Optional) Files will be stored encrypted
- (Optional) Access to files will be allowed only for clients, which have an access token (key)
- File editing/deletion will be allowed only for clients, which have a certificate (key)

Normal operational flow

TODO: Refine this section with more details

- We start with at least:
    - 1 Keeper node
    - 1 Verifier node
    - 1 Client who wants to store a file, which is 100MB
- The Client contacts the Verifier with a request to store a file with size 100MB, for 10 days
- The Verifier proposes a contract, which will cost the Client X number of tokens to store the file for that period
- The Client accepts
- The Verifier takes the file and contacts Keeper nodes, offering them a contract to store the given file for 10 days for Y number of tokens
- The Verifier distributes the file to the Keeper nodes that accept the contract
- The Verifier creates a hash of the file and verifies that the Keeper nodes have the file by asking them to send the Verifier the hash of the file. This check occurs regularly
- The Verifier chooses another 2 Verifiers (based on proximity in the ID space), which should also hold the hash of the file
- The Client is informed that the contract is complete and is given the IDs of the Verifiers that know where the file is stored.
- If the Client wants to retrieve the file, they contact the Verifiers, which forward the request to the Keeper nodes
- If the Client wishes to store the file for longer, they need to establish a new contract before the 10 days period ends

Award points system

TODO: This needs to be refined.

- Keeper nodes will be awarded tokens whenever a contract they made is verified
    - Tokens will be kept on a blockchain to avoid malicious peers lying about their tokens
- Ideas for token awards by Verifiers:
    - Longer storage period → More tokens
    - Faster download → More tokens
    - More popular file → More tokens (probably a bad idea because nodes will prefer popular files and drop less popular ones)
        - This can be mitigated by increasing the rewards for unpopular files. This will cause files to oscillate between being “popular” and “unpopular”, which pay out high rewards, but files that are neither popular nor unpopular (in the middle) will be less profitable.

- Clients that want to store a file will have to “pay” for that file storage with their own tokens
    - Ideally, tokens can be “purchased” using tokens from other chains
    - (Optional) have a more practical method of payment

Sources

- https://www.forbes.com/sites/forbestechcouncil/2022/08/17/immutable-databases-versus-blockchain-platforms-for-tamper-proof-data/?sh=6b54d7c24c02
- More on cgroups v2

TODO

- Read https://www.dolthub.com/blog/2022-03-21-immutable-database/
- Read https://www.tibco.com/reference-center/what-is-immutable-data
- Read https://www.dock.io/post/verifiable-credentials
- File storage contracts will be timed. After the specified time, the file will be removed

IPFS - lookup

Immutable databases - immudb instead of a blockchain

Look into freenet

Ensure it can’t be spammed

Zero knowledge proofs

Merkle trees

store files at differently aged peers, don't store only on old peers as they can still be sybils or store on different scored nodes

think about prioritizing downloading from younger peers if a request comes from someone with a low score don't give high rewards

harnessing the power of disruptive technologies (peer to peer) andy oram

think about centralized authority allowing peers to join

ARA