

vasa [Follow](#)

Entrepreneur | Co-founder/Developer @TowardsBlockChain, an MIT CIC incubated startup | <http://vaibhavsaini.com>

Aug 13 · 42 min read

StoragePedia: An Encyclopedia of 5 Blockchain Storage Platforms

An In-depth Comparison of all major Distributed Storage Platforms



Some of the most important things when it comes to blockchains and distributed technology are consensus algorithms(how the network as a whole reaches to a common decision) and smart contracts(for implementing applications we use every day in this centralized world). We have talked about them earlier here:

ConsensusPedia: An Encyclopedia of 30 Consensus Algorithms

A complete list of all consensus algorithms.

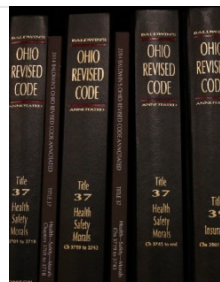
hackernoon.com



ContractPedia: An Encyclopedia of 40 Smart Contract Platforms

A Complete List of all Smart Contract supportive Platforms

hackernoon.com



But when talking about everyday applications, these properties are not enough to support today's world's needs. It's hard to imagine enjoying your favorite Movie/TV Series like you do on Netflix, storing/sharing your memorable videos/photos like you do on facebook or playing your favorite online games like DOTA on blockchains, if we just rely on the above 2 properties.

The thing we are missing is a robust, safe and decentralised content storage and distribution system to serve today's applications.

Here we will explore and evaluate some of the most popular distributed storage platforms.

1. Swarm



Status: *Live*

Explanation:

Swarm is a distributed storage platform and content distribution service, a native base layer service of the ethereum *web3* stack. The primary objective of Swarm is to provide a sufficiently decentralized and redundant store of Ethereum's public record, in particular to store and distribute dapp code and data as well as blockchain data. From an economic point of view, it allows participants to efficiently pool their storage and bandwidth resources in order to provide these services to all participants of the network, all while being incentivised by Ethereum.

Objective

Swarm's broader objective is to provide infrastructure services for developers of decentralised web applications (dapps), notably: ***messaging, data streaming, peer to peer accounting, mutable resource updates, storage insurance, proof of custody scan and repair, payment channels and database services.***

From the end user's perspective, Swarm is not that different from the world wide web, with the exception that uploads are not hosted on a specific server. Swarm offers a peer-to-peer storage and serving solution that is DDoS-resistant, has zero-downtime, fault-tolerant and censorship-resistant as well as self-sustaining due to a built-in incentive system which uses peer-to-peer accounting and allows trading resources for payment. Swarm is designed to deeply integrate with the devp2p multiprotocol network layer of Ethereum as well as with the Ethereum blockchain for domain name resolution (using ENS), service payments and content availability insurance.

Note: In order to resolve ENS names, your Swarm node has to be connected to an Ethereum blockchain (mainnet, or testnet).

Please refer to our development roadmap to stay informed with swarm's progress.

Overview

Swarm is set out to provide base layer infrastructure for a new decentralised internet. Swarm is a peer-to-peer network of nodes providing distributed digital services by contributing resources (storage, message forwarding, payment processing) to each other. The Ethereum Foundation operates a Swarm testnet that can be used to test out functionality in a similar manner to the Ethereum testnet (ropsten). Everyone can join the network by running the Swarm client node on their server, desktop, laptop or mobile device. See [Getting started with Swarm](#) for how to do this. The Swarm client is part of the Ethereum stack, the reference implementation is written in golang and found under the go-ethereum repository. Currently at POC (proof of concept) version 0.3 is running on all nodes.

Swarm offers a **local HTTP proxy** API that dapps or command line tools can use to interact with Swarm. Some modules like messaging are only available through RPC-JSON API. The foundation servers on the testnet are offering public gateways, which serve to easily demonstrate functionality and allow free access so that people can try Swarm without even running their own node.

Swarm is a collection of nodes of the devp2p network each of which run the bzz protocol suite on the same network id.

Swarm nodes can also connect with one (or several) ethereum blockchains for domain name resolution and one ethereum blockchain for bandwidth and storage compensation. Nodes running the same network id are supposed to connect to the same blockchain for payments. A Swarm network is identified by its network id which is an arbitrary integer.

Swarm allows for *upload and disappear* which means that any node can just upload content to the Swarm and then is allowed to go offline. As long as nodes do not drop out or become unavailable, the content will still be accessible due to the 'synchronization' procedure in which nodes continuously pass along available data between each other.

Public gateways

Swarm offers a local HTTP proxy API that Dapps can use to interact with Swarm. The Ethereum Foundation is hosting a public gateway, which allows free access so that people can try Swarm without running their own node.

The Swarm public gateway can be found at <https://swarm-gateways.net> and is always running the latest stable Swarm release.

Note: This gateway currently only accepts uploads of limited size. In future, the ability to upload to this gateways is likely to disappear entirely.

Uploading and Downloading Data

Uploading content consists of “uploading” content to your local Swarm node, followed by your local Swarm node “syncing” the resulting chunks of data with its peers in the network. Meanwhile, downloading content consists of your local Swarm node querying its peers in the network for the relevant chunks of data and then reassembling the content locally.

The Content Resolver: ENS

Note: In order to resolve ENS names, your Swarm node has to be connected to an Ethereum blockchain (mainnet, or testnet).

ENS is the system that Swarm uses to permit content to be referred to by a human-readable name, such as “theswarm.eth”. It operates analogously to the DNS system, translating human-readable names into machine identifiers—in this case, the Swarm hash of the content you’re referring to. By registering a name and setting it to resolve to the content hash of the root manifest of your site, users can access your site via a URL such as `bzz://theswarm.eth/`.

Note: Currently The bzz scheme is not supported in major browsers such as Chrome, Firefox or Safari. If you want to access the bzz scheme through these browsers, currently you have to either use an HTTP gateway, such as <https://swarm-gateways.net/bzz://theswarm.eth/> or use a browser which supports the bzz scheme, such as Mist.

You can learn more about uploading, downloading and handling content on swarm ***here***.

Mutable Resource Updates

Mutable Resource Updates is a ***highly experimental feature***, available from Swarm POC3. It is under active development, so expect things to change.

We have previously learned in this guide that when we make changes in data in Swarm, the hash returned when we upload that data will change in totally unpredictable ways. With *Mutable Resource Updates*, Swarm provides a built-in way of keeping a persistent identifier to changing data.

The usual way of keeping the same pointer to changing data is using the Ethereum Name Service `ENS`. However, `ENS` is an on-chain feature, which limits functionality in some areas:

1. Every update to an `ENS` resolver will cost you gas to execute.
2. It is not possible to change the data faster than the rate that new blocks are mined.
3. Correct `ENS` resolution requires that you are always synced to the blockchain.

Mutable Resource Updates allows us to have a non-variable identifier to changing data without having to use the `ENS`. The Mutable Resource can be referenced like a regular Swarm object, using the key obtained when the resource was created (`MRU_MANIFEST_KEY`). When the resource's data is updated the `MRU_MANIFEST_KEY` will point to the new data.

If using *Mutable Resource Updates* in conjunction with an `ENS` resolver contract, only one initial transaction to register the `MRU_MANIFEST_KEY` will be necessary. This key will resolve to the latest version of the resource (updating the resource will not change the key).

There are 3 different ways of interacting with *Mutable Resource Updates* : HTTP API, Golang API and Swarm CLI.

You can check them out ***here***.

THINGS TO NOTE:

- Only the private key (address) that created the Resource can update it.
- When creating a Mutable Resource, one of the parameters that you will have to provide is the expected update `frequency`. This indicates how often (in seconds) your resource will be updated. Although you can update the resource at other rates, doing so will slow down the process of retrieving the resource.

Encryption on Swarm

Introduced in POC 0.3, symmetric encryption is now readily available to be used with the `swarm up` upload command. The encryption mechanism is meant to protect your information and make the chunked data unreadable to any handling Swarm node.

Swarm uses Counter mode encryption to encrypt and decrypt content. When you upload content to Swarm, the uploaded data is split into 4 KB chunks. These chunks will all be encoded with a separate randomly generated encryption key. The encryption happens on your local Swarm node, unencrypted data is not shared with other nodes. The reference of a single chunk (and the whole content) will be the concatenation of the hash of encoded data and the decryption key. This means the reference will be longer than the standard unencrypted Swarm reference (64 bytes instead of 32 bytes).

When your node syncs the encrypted chunks of your content with other nodes, it does not share the the full references (or the decryption keys in any way) with the other nodes. This means that other nodes will not be able to access your original data, moreover they will not be able to detect whether the synchronized chunks are encrypted or not.

When your data is retrieved it will only get decrypted on your local Swarm node. During the whole retrieval process the chunks traverse the network in their encrypted form, and none of the participating

peers are able to decrypt them. They are only decrypted and assembled on the Swarm node you use for the download.

More info about how we handle encryption at Swarm can be found [here](#).

THINGS TO NOTE:

- Swarm supports encryption. Upload of unencrypted sensitive and private data is highly discouraged as there is no way to undo an upload. Users should refrain from uploading illegal, controversial or unethical content.
- Swarm currently supports both encrypted and unencrypted `swarm up` commands through usage of the `--encrypt` flag. This might change in the future.
- The encryption feature is non-deterministic (due to a random key generated on every upload request) and users of the API should not rely on the result being idempotent; thus uploading the same content twice to Swarm with encryption enabled will not result in the same reference.

PSS

pss (Postal Service over Swarm) is a messaging protocol over Swarm with strong privacy features. The *pss* API is exposed through a JSON RPC interface described in the API Reference, here we explain the basic concepts and features.

Note: `pss` is still an experimental feature and under active development and is available as of POC3 of Swarm. Expect things to change.

Basics

With `pss` you can send messages to any node in the Swarm network. The messages are routed in the same manner as retrieve requests for chunks. Instead of chunk hash reference, `pss` messages specify a destination in the overlay address space independently of the message payload. This destination can describe a *specific node* if it is a complete overlay address or a *neighbourhood* if it is partially

specified one. Up to the destination, the message is relayed through devp2p peer connections using *forwarding kademia* (passing messages via semi-permanent peer-to-peer TCP connections between relaying nodes using kademia routing). Within the destination neighbourhood the message is broadcast using gossip.

Since `pss` messages are encrypted, ultimately *the recipient is whoever can decrypt the message*. Encryption can be done using asymmetric or symmetric encryption methods.

The message payload is dispatched to *message handlers* by the recipient nodes and dispatched to subscribers via the API.

Note: `pss` does not guarantee message ordering (Best-effort delivery) nor message delivery (e.g. messages to offline nodes will not be cached and replayed) at the moment.

Privacy Features

Thanks to end-to-end encryption, `pss` caters for private communication.

Due to forwarding kademia, `pss` offers sender anonymity.

Using partial addressing, `pss` offers a sliding scale of recipient anonymity: the larger the destination neighbourhood (the smaller prefix you reveal of the intended recipient overlay address), the more difficult it is to identify the real recipient. On the other hand, since dark routing is inefficient, there is a trade-off between anonymity on the one hand and message delivery latency and bandwidth (and therefore cost) on the other. This choice is left to the application.

Forward secrecy is provided if you use the Handshakes module.

You can check out more about using `pss` *here*.

Architecture

You can know more about swarm's architecture *here*.

Example Dapp

You can check out a sample working swarm dapp *here*.

THINGS TO REMEMBER:

- Always use encryption for sensitive content! For encrypted content, uploaded data is 'protected', i.e. only those that know the reference to the root chunk (the swarm hash of the file as well as the decryption key) can access the content. Since publishing this reference (on ENS or with MRU) requires an extra step, users are mildly protected against careless publishing as long as they use encryption. ***Even though there is no guarantees for removal, unaccessed content that is not explicitly insured will eventually disappear from the Swarm, as nodes will be incentivised to garbage collect it in case of storage capacity limits.***
- Uploaded content is not guaranteed to persist on the testnet until storage insurance is implemented (see Roadmap for more details). All participating nodes should consider participation a voluntary service with no formal obligation whatsoever and should be expected to delete content at their will. ***Therefore, users should under no circumstances regard Swarm as safe storage until the incentive system is functional.***
- The Swarm is a Persistent Data Structure, therefore there is no notion of delete/remove action in Swarm. This is because content is disseminated to swarm nodes who are incentivised to serve it.

Swarm Reddit | Swarm Twitter | Swarm Github

2. IPFS



Status: *Live (It's incentivisation system, "Filecoin" is Not Live)*

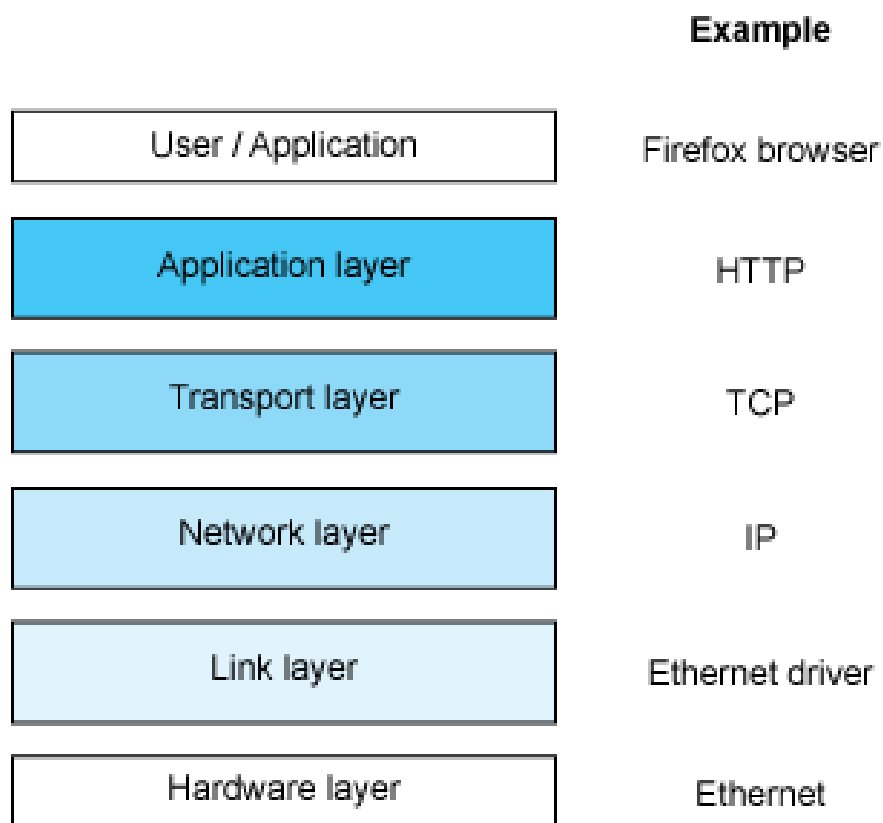
Explanation:

IPFS(Interplanetary File System) a peer-to-peer (p2p) filesharing system that aims to fundamentally change the way information is distributed across & beyond the globe. It is somewhat similar to Swarm, or we can also say Swarm is somewhat similar to IPFS.

IPFS consists of several innovations in communication protocols and distributed systems that have been combined to produce a file system like no other. So to understand the full breadth and depth of what IPFS is trying to achieve, it's important to understand the tech breakthroughs that make it possible and what all problems it's trying to solve.

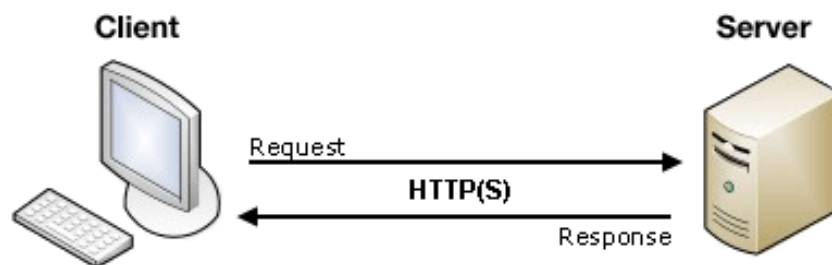
IPFS boasts to replace *http*. So, let's take a look on how our internet works today.

Simply put, the internet is a collection of protocols that describe how data moves around a network. Developers adopted these protocols over time and built their applications on top of this infrastructure. One of the protocols that serves as the backbone of the web is HTTP or HyperText Transfer Protocol. This was invented by Tim Berners-Lee in 1991.



Internet Protocol stack aka. OSI Model

HTTP is a request-response protocol. A client, for example a web browser, sends a request to an external server. The server then returns a response message, for example, the Google homepage back to the client. This is a location-addressed protocol which means when I type google.com into my browser, it gets translated into an IP address of some Google server, then the request-response cycle is initiated with that server.



How people talk on Internet

Problems with http

Let's say you are sitting in a lecture hall, and the professor asks you to go to a specific website. Every student in the lecture makes a request to that website and are given a response. This means that the same exact data was sent individually to each student in the room. If there are 100 students, then that's 100 requests and 100 responses. This is obviously not the most efficient way to do things. Ideally, the students will be able to leverage their physical proximity to more efficiently retrieve the information they need.

HTTP also presents a big problem if there is some problem in the network's line of communication and the client is unable to connect with the server. This can happen if an ISP has an outage, a country is blocking some content, or if the content was simply deleted or moved. These types of broken links exist everywhere on the HTTP web.

The location-based addressing model of HTTP encourages centralization. It's convenient to trust a handful of applications with all our data but because of this much of the data on the web becomes soiled. This leaves those providers with enormous responsibility and power over our information(*cough facebook sucks cough*).

HTTP is great for loading websites but it wasn't designed for the transfer of large amounts of data (like audio and video files). These constraints possibly enabled the emergence and mainstream success of alternative filesharing systems like Napster (music) and BitTorrent (movies and pretty much anything).

Fast forward to 2018, where on-demand HD video streaming and big data are becoming ubiquitous; we are continuing the upward march of producing/consuming more and more data, along with developing more and more powerful computers to process them. Major advancements in cloud computing have helped sustain this transition, however the fundamental infrastructure for distributing all this data has remained largely the same.

The Solution

IPFS began as an effort by **Juan Benet** to build a system that is very fast at moving around versioned scientific data. It is a synthesis of well-tested internet technologies such as DHTs, the Git versioning system and Bittorrent. It creates a P2P swarm that allows the

exchange of *IPFS objects*. The totality of IPFS objects forms a cryptographically authenticated data structure known as a *Merkle DAG* and this data structure can be used to model many other data structures. Or in other words...

“IPFS is a distributed file system that seeks to connect all computing devices with the same system of files. In some ways, this is similar to the original aims of the Web, but IPFS is actually more similar to a single bittorrent swarm exchanging git objects. IPFS could become a new major subsystem of the internet. If built right, it could complement or replace HTTP. It could complement or replace even more. It sounds crazy. It is crazy.”[1]

At its core, IPFS is a versioned file system that can take files and manage them and also store them somewhere and then tracks versions over time. IPFS also accounts for how those files move across the network so it is also a distributed file system.

IPFS has rules as to how data and content move around on the network that are similar in nature to bittorrent. This file system layer offers very interesting properties such as:

- websites that are completely distributed.
- websites that have no origin server.
- websites that can run entirely on client side browsers.
- websites that do not have any servers to talk to.

Let's see how these different tech breakthroughs work together.

Distributed Hash Tables

A hash table is a data structure that stores information as key/value pairs. In distributed hash tables (DHT) the data is spread across a network of computers, and efficiently coordinated to enable efficient access and lookup between nodes.

The main advantages of DHTs are in decentralization, fault tolerance and scalability. Nodes do not require central coordination, the system can function reliably even when nodes fail or leave the network, and

DHTs can scale to accommodate millions of nodes. Together these features result in a system that is generally more resilient than client-server structures.

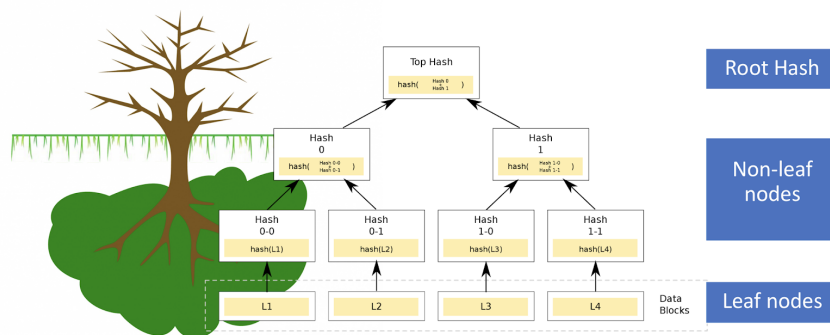
Block Exchanges

The popular file sharing system Bittorrent is able to successfully coordinate the transfer of data between millions of nodes by relying on an innovative data exchange protocol, however it is limited to the torrent ecosystem. IPFS implements a generalized version of this protocol called BitSwap, which operates as a marketplace for any type of data. This marketplace is the basis for Filecoin: a p2p storage marketplace built on IPFS.

Merkle DAG

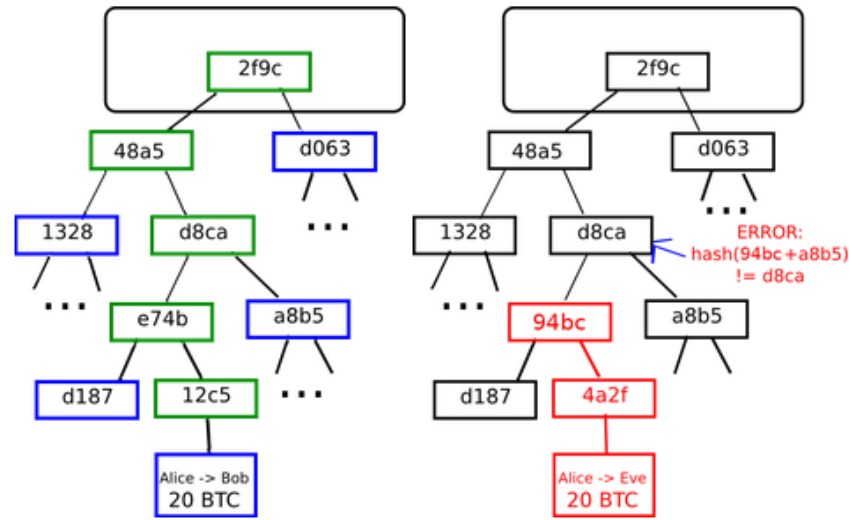
A merkle DAG is a blend of a Merkle Tree and a Directed Acyclic Graph (DAG). Merkle trees ensure that data blocks exchanged on p2p networks are correct, undamaged and unaltered. This verification is done by organizing data blocks using cryptographic hash functions. This is simply a function that takes an input and calculates a unique alphanumeric string (hash) corresponding with that input. It is easy to check that an input will result in a given hash, but incredibly difficult to guess the input from a hash.

The individual blocks of data are called 'leaf nodes', which are hashed to form 'non-leaf nodes'. These non leaf nodes can then be combined and hashed, until all the data blocks can be represent by a single root hash. Here's an easier way to conceptualize it:



Tree vs Merkle Tree

A DAG is a way to model topological sequences of information that have no cycles. A simple example of a DAG is a family tree. A merkle DAG is basically a data structure where hashes are used to reference data blocks and objects in a DAG. This creates several useful features: all content on IPFS can be uniquely identified, since each data block has a unique hash. Plus the data is tamper-resistant because to alter it would change the hash, as shown below:



Un-tamperable nature of Merkle tree

The central tenet of IPFS is modeling all data on a generalized Merkle DAG. The significance of this security feature is hard to overstate. One example of showing how powerful this idea is that assets worth trillions of dollars are protected by this principle.

Version Control Systems

Another powerful feature of the Merkle DAG structure is that it allows you to build a distributed version control system (VCS). The most popular example of this is Github, which allows developers to easily collaborate on projects simultaneously. Files on Github are stored and versioned using a merkle DAG. It allows users to independently duplicate and edit multiple versions of a file, store these versions and later merge edits with the original file.

IPFS uses a similar model for data objects: as long as objects corresponding to the original data, and any new versions are

accessible, the entire file history can be retrieved. Given that data blocks are stored locally across the network and can be cached indefinitely, this means that IPFS objects can be stored permanently.

Additionally, IPFS does not rely on access to Internet protocols. Data can be distributed in overlay networks, which are simply networks built on another network. These features are notable, because they are core elements in a censorship-resistant web. It could be a useful tool in promoting free speech to counter the prevalence of internet censorship around the world, but we should also be cognizant of the potential for abuse by bad actors.

Self-certifying File System

The last essential component of IPFS we'll cover is the Self-certifying File System (SFS). It is a distributed file system that doesn't require special permissions for data exchange. It is "self-certifying" because data served to a client is authenticated by the file name (which is signed by the server). The result? You can securely access remote content with the transparency of local storage.

IPFS builds on this concept to create the InterPlanetary Name Space (IPNS). It is an SFS that uses public-key cryptography to self-certify objects published by users of the network. We mentioned earlier that all objects on IPFS can be uniquely identified, but this also extends to nodes. Each node on the network has a set of public keys, private keys and a node ID which is the hash of its public key. Nodes can therefore use their private keys to 'sign' any data objects they publish, and the authenticity of this data can be verified using the sender's public key.

Here's a quick recap of the key IPFS components:

- With the Distributed Hash Table, nodes can store & share data without central coordination.
- IPNS allows exchanged data to be instantly pre-authenticated and verified using public key cryptography.
- The Merkle DAG enables uniquely identified, tamper-resistant and permanently stored data.

You can check out more(in-depth) on how files are distributed in the network in this article by [ConsenSys](#). Also, you can check out [IPFS Whitepaper](#) *here*.

THINGS TO REMEMBER:

- ***Always use encryption for sensitive content!*** For encrypted content, uploaded data is 'protected', i.e. only those that know the reference to the root Hash (the root hash of the file as well as the decryption key) can access the content.
- IPFS is a Persistent Data Structure, therefore there is no notion of delete/remove action in IPFS. This is because content is disseminated to IPFS nodes who are incentivised to serve it.
- Uploaded content is not guaranteed to persist on the network. All participating nodes should consider participation a voluntary service with no formal obligation whatsoever and should be expected to delete content at their will. ***Therefore, users should under no circumstances regard IPFS as safe storage until any incentive system(Filecoin) is functional.***

[IPFS Reddit](#) | [IPFS Twitter](#) | [IPFS Github](#)

3. Sia



Status: Live

ENS LABS EX1012
Page 18 of 43

Explanation:

SIA offers access to decentralized cloud storage platforms for renters looking to make use of cheaper, faster means of using data centers that are open to anyone and not governed by a single authoritative source. Siacoin is based on an independent Sia blockchain and agreements are made between a storage renter and a provider.

Files Are Divided Prior To Upload

The Sia software divides files into 30 segments before uploading, each targeted for distribution to hosts across the world. This distribution assures that no one host represents a single point of failure and reinforces overall network uptime and redundancy.

File segments are created using a technology called Reed-Solomon erasure coding, commonly used in CDs and DVDs. Erasure coding allows Sia to divide files in a redundant manner, where any 10 of 30 segments can fully recover a user's files.

This means that if 20 out of 30 hosts go offline, a Sia user is still able to download his/her files.

Each File Segment Is Encrypted

Before leaving a renter's computer, each file segment is encrypted. This ensures that hosts only store encrypted segments of user data.

This differs from traditional cloud storage providers like Amazon, who do not encrypt user data by default. Sia is more secure than existing solutions because hosts only store encrypted file segments, rather than whole files.

Sia uses the Twofish algorithm, an open source and secure encryption standard that was a finalist in the Advanced Encryption Standard (AES) contest.

Files Are Sent To Hosts Using Smart Contracts

Using the Sia blockchain, renters form file contracts with hosts. These contracts set pricing, uptime commitments, and other aspects of the

relationship between the renters and the hosts.

File contracts are a type of smart contract. They allow us to create cryptographic service level agreements (SLAs) that are stored on the Sia blockchain.

Since file contracts are automatically enforced by the network, Sia has no need for intermediaries or trusted third parties.

Renters And Hosts Pay With Siacoin

Both renters and hosts use Siacoin, a unique cryptocurrency built on the Sia blockchain. Renters use Siacoin to buy storage capacity from hosts, while hosts deposit Siacoin into each file contract as collateral.

Micropayments flow between renters and hosts using a technology called payment channels, which is similar to Bitcoin's Lightning Network. Payments between renters and hosts occur off-chain, greatly increasing network efficiency and scalability. You can check more about payment channels here:

Difference between SideChains and State Channels

A complete comparison of the two scaling methods.

hackernoon.com



Since hosts pay collateral into every storage contract, they have a strong disincentive to go offline.

Each time a user and a hosting provider enter into a contract on Sia, the user must put up an allowance (to pay for the hosting), and the hosting provider must put up a deposit (to ensure good behavior). Upon conclusion of a contract, 3.9% of both the allowance and the deposit is collected by the software and paid to holders of Siafunds, a secondary token of the protocol. Nebulous Labs, the company developing the protocol, holds ~90% of Siafunds.

This is certainly an interesting long-term funding model and an alternative to the “all-at-once” ICO model in vogue today; however, this feature may impose a structural cost on the system that makes it more expensive than competitors. File storage is likely to be a highly competitive market, driving down prices significantly (compared to current centralized options). Once several different decentralized cloud storage platforms have been deployed, these protocols will compete not just with centralized alternatives but also with one another. Sia levies their 3.9% fee on both the allowance and the deposit, but in most contracts both of these fees are paid by the user. Other protocols do not have this structural cost, and that may make them less expensive and more attractive to users in the long-term.

Prices change, but you can expect to pay in the region of 109 Siacoins or \$1 per TB/month.

Contracts Renew Over Time

Renters prepay for storage within file contracts, setting aside a fixed amount of Siacoin to be spent on storing and transferring data. File contracts typically last 90 days.

Sia automatically renews contracts when they are within a certain window of expiring. If contracts are not renewed, Sia returns any unused coins to the renter at the end of the contract period.

As individual hosts go offline, Sia automatically moves renter data to new hosts in a process called file repair.

Hosts Submit Storage Proofs

At the end of a file contract, the host must prove that he/she is storing the renter’s data. This is called a storage proof. If the storage proof appears on the blockchain within a certain timeframe, the host is paid. If not, the host is penalized.

Storage proofs are made possible by a technology called Merkle trees. Merkle trees make it possible to prove that a small segment of data is part of a larger file. The advantage of these proofs is that they are very small, no matter how large the file is. This is important because the proofs are stored permanently on the blockchain.

Finally, Sia faces a somewhat significant barrier to entry. Currently, one must purchase Bitcoin, use that to purchase Siacoins, and send the Siacoins to the Sia client software to begin using the network. For everyday users who do not already hold cryptocurrencies, this is a major roadblock. It also presents an issue for enterprise clients, since most businesses remain unwilling to hold or transact in cryptocurrencies for regulatory and financial risk reasons. To date, the Sia team has not prioritized design, accessibility, or usability, and has given no indications that this is a significant priority.

THINGS TO REMEMBER:

- Unlike Swarm and IPFS, Sia is a distributed cloud storage system. It doesn't focus on creating a infrastructure to replace *http*, rather it focuses on creating a distributed, self sustaining and cheap market place for cloud storage.
- Unlike Swarm and IPFS, Sia has a full fledged incentivization system for both, storage users and storage providers. So you have much more control over your data(Well, you are paying for it!). You are fully guaranteed that the data you provide to the network will be stored and will not be lost(as in Swarm and IPFS). Also, you can delete the content from the network and be sure of the fact that your data is fully wiped out from the network.

You can read more about Sia's technology ***here***.

Sia Reddit | Sia Twitter | Sia Github

4. Storj



Status: Live

Explanation:

Storj (pronounced: storage) aims to become a cloud storage platform that can't be censored, monitored, or have downtime. It is one of the first decentralized, end-to-end encrypted cloud storage platform.

Storj is made up of a bunch of interlocking pieces working together to create a unified system. As people interact with various parts of this system, they get a different idea of what Storj is. A home user doesn't need any knowledge of the Bridge or of the protocol in order to share storage space, and a developer doesn't need to know anything about the home users in order to use the Storj API. Each person can have a drastically different experience while interacting with the same system. So what is Storj? It's a protocol, a suite of software, and the people who design, build, and use it.

The Storj Protocol

Storj's core technology is an enforceable, peer-to-peer, storage contract. It's a way for two people (or computers) to agree to exchange some amount of storage for money without knowing each other. We call the computer selling space the "farmer," and the computer purchasing space the "renter." The renter and farmer meet,

negotiate an agreement, and move data from the renter to the farmer for safekeeping.

Contracts & Audits

A contract has a set duration. Over this time, the renter periodically checks that the farmer is still available. The farmer responds with a cryptographic proof that it still has the file. Finally, the renter pays the farmer for each proof it receives and verifies. This process of challenge -> proof -> payment is referred to as an “audit,” as the renter is auditing the farmer’s storage. At the end of the contract period, the farmer and renter are free to renegotiate or end the relationship.

While the core technology allows for any type of payment, some types are better suited than others. Traditional payment systems, like ACH or SEPA, are poorly suited to paying on a per-audit basis. They’re slow, hard to verify, and often come with expensive fees. The ideal payment method for the Storj Protocol is a cryptocurrency micropayment channel. It allows for extremely small payments that are immediately verifiable and secure, with minimal fees. This means that payments and audits can be paired as closely as possible.

Enforcement follows a simple tit-for-tat model: if the farmer fails an audit, i.e. is offline or can’t demonstrate that she still has the data, then the renter doesn’t have to pay. After all, he’s no longer getting the service he was paying for. Similarly, if a renter goes offline, or fails to make a payment on time, the farmer can drop the data, and look for a new contract from someone else. As long as both parties are following the terms of the contract, everyone ends up better off.

Pairing the payments directly to the audits minimizes the risk of dealing with a stranger. If the file gets dropped halfway through the contract, the renter only paid for the service actually performed, as proven by the audits. He needs to find a new farmer, but is not out any significant amount of money. If a renter disappears or stops paying a farmer, the farmer has received payment for all her previous services already. She’s only missing one audit payment, and the time it takes to find a new renter to buy that space.

The Storj Network

To enable renters and farmers to meet each other, the contracting and negotiation system has been built on top of a distributed hash table (DHT). A DHT is basically a way of self-organizing a bunch of nodes into a useful network. We're using a modified version of an algorithm called Kademlia.

Instead of having a central server register every node and coordinate all contracts, the DHT lets farmers and renters broadcast their contract offers to a wide group of nodes. Interested nodes can easily contact the person who made the contract offer. That way farmers and renters can find any number of potential partners, and buy or sell storage space on a broad permissionless market.

To find a partner, a node can sign an incomplete contract and publish it to the network. Other nodes on the network can subscribe to certain types of contracts (i.e. types they might be interested in) and respond to these published offers. This model is called publish-subscribe or pub/sub. Nodes can easily determine what contracts they're interested in, and forward on contracts to other nodes they think might be interested.

Together, the contracting system and the network form what we call the *Storj Protocol*. It's a description of how nodes on the network behave, how nodes communicate with other nodes, how contracts get negotiated and executed, and everything else necessary to buy and sell storage space on a distributed system. Anyone can implement the Storj Protocol in any way they please.

Storj Toolset

The protocol contains all the tools necessary to securely make storage contracts, but it's missing a lot of things. It functions, but it's not useful yet. To be useful to a renter, the system needs availability, bandwidth, and any number of other commitments in the form of a Service Level Agreement (SLA). The farming software needs management features to avoid using excessive resources and automation features to effectively deploy to multiple hosts. Instead of trying to fit all these features into the core protocol, Storj opted to address them in an additional software layer. To make this network useful and easy to interact with, Storj is releasing two tools: *Storj Share* and *Bridge*.

StorjShare

StorjShare is the reference farming client. It allows users to easily setup and run a farm on any machine. StorjShare is available as a command-line interface (CLI) for more advanced users and to enable automation. The CLI allows the user to set parameters like the amount of storage space to share, storage location, and a payment address. It also handles contract negotiation, audit responses, and all other network communications.

Storj is also releasing a StorjShare graphical user interface (GUI) in order to streamline the farming process for our non-technical users. Anyone can download the StorjShare GUI, fill in a few fields, and join the network. The GUI is a wrapper around the CLI which does all the heavy lifting. After initial setup, the user rarely needs to interact with the StorjShare GUI. They should be able to set it up, minimize it, and let it run in the background.

If the user opts in to data collection, StorjShare will also collect system telemetry. This data might include hard drive capacity and level of utilization as well as information about network connection quality. The telemetry data gets sent back to Storj Labs so that Storj developers can use it to improve the network and our software. In the future, StorjShare could even enable people to opt in to special services and programs.

Bridge

To help renters use the network, Storj also created Bridge. Bridge is designed to be deployed to a production server to handle contract negotiation, auditing, payments, availability, and a number of other needs. Bridge exposes these services and, by extension, storage resources via an Application Programming Interface (API) and client. The client is designed to be integrated into other apps, so that any application can use a Bridge server to store data on the Storj network without having to be a part of the network.

As its name implies, Bridge is a centralized bridge into the decentralized Storj network. Its goal is to allow traditional applications to interact with the Storj network like they would with any other object store. It distills all the complexities of p2p communication and storage contract negotiation to push and pull

requests. Unlike most object stores, Bridge doesn't deal directly in objects, but rather in references to objects. It stores pointers to the locations of the objects on the distributed network, as well as the information necessary to audit those objects. Ideally, no data will transit through Bridge, but rather will be transmitted directly to farmers on the network.

The Bridge Client handles all the client-side work to use the network effectively. It encrypts files as they enter the network, preserving privacy and security. To ensure availability, it shards files, applies erasure coding, and spreads the shards across multiple farmers. The client then communicates with the Bridge to manage each of the shard locations on the network, and helps the user manage their encryption keys locally. While the initial implementation of the Bridge Client is a Node.js package, the end goal is to have it available in many other languages.

The Storj API

Storj core service is an object store, similar to Amazon S3. This object store is managed by a set of public Bridge nodes. We maintain the infrastructure to negotiate contracts, manage payments, audits, etc. Our customers interact with our Bridges via the Bridge Client, and never even have to know they're using a distributed network. The API is designed for usability, so everything complicated gets handled behind the scenes to provide a smooth, extensible development experience.

Storj uses its extensive knowledge of the network to provide the top-notch quality of service. The decisions their Bridge makes are based on historical interactions with countless farmers. Storj uses data about their performance, as well as their self-reported telemetry data, to intelligently distribute data across the network. Storj optimizes for high uptime as well as fast retrieval.

Account management features are available via the Bridge Client, or via Storj's beautiful webapp. Again, the function of the GUI is to make the experience as smooth as possible. Storj believes that user experience (UX) is tragically neglected by most developer-oriented services, and because of this, they're designing the product with simplicity and usability in mind. Instead of offering a wide range of

cloud computing and storage services, they aim to offer a single elegant user experience.

With the Storj API, they're trying to build the ideal tool for developers like us: people who care about time-to-prototype, high-quality code, and rapid iteration. They want to provide tools and support to small teams, rapidly-scaling products, and individual developers. They understand that every developer actually works for himself/herself, and the projects he/she cares about. This is a labor of love for us. Storj wants to build an object store that gets out of the way so that developers can concentrate on building the projects they love.

You can learn more about Storj from [Storj Labs](#).

[Storj Reddit](#) | [Storj Twitter](#) | [Storj Github](#)

5. MaidSafe



Status: Live

Explanation:

The SAFE Network is an autonomous distributed network for data storage and communications. It provides Secure Access For Everyone (SAFE). Data stored on the network has extremely high availability, durability, privacy and security. The network scales efficiently and the security of data stored on the network increases as the network grows.

Why SAFE Network?

The existing server-client-based internet gives ownership of data to whoever operates the servers, rather than the people creating the data. The operators can restrict, modify, remove or sell that data with no recourse from the user that created it. Poor acceptance and availability of federated protocols to distribute user data on terms that favour the creators has motivated the creation of the SAFE Network.

Clients storing data on the SAFE Network are protected by default with strong encryption and can control access through a flexible permissions layer.

Clients retrieving data from the SAFE Network are protected by a secure routing and addressing system.

Clients benefit from secure defaults, including built-in end-to-end encryption and secure authentication.

The network is comprised of a graph of independently operated nodes (called vaults) that validate, store and deliver data. Vault operators can contribute to the retention of network data and network performance by supplying disc space and bandwidth for use by the network. Vault operators may join or leave the network at any time without affecting the security of data stored on the network.

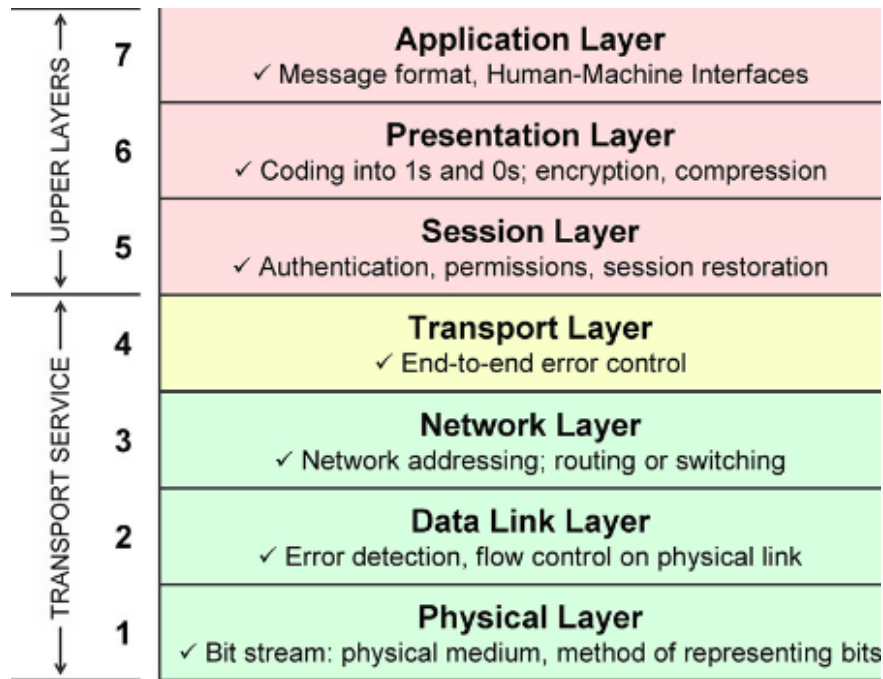
Network tokens called Safecoin are distributed to vault operators by the network for providing these resources. The tokens may then be used to purchase network storage space for their own use or to utilise other resources on the network. This motivates benevolent behaviour of vault operators and protects the network against malicious behaviour.

The network utilises SHA3-256 identifiers for vaults and data in combination with XOR distances between these identifiers to anonymise and globally distribute all data and traffic.

Much of the existing internet infrastructure is improved by the SAFE Network, including Addressing, Domain Name System, Transport Layer Security, Packet Routing, server software such as http web servers and imap mail servers, authentication layers such as oauth

and openid; these are all superseded by secure-by-default modules that combine to make the SAFE Network operable.

SAFE operates on existing physical internet infrastructure, but replaces all layers of the network from there up. It primarily targets OSI layers 3 to 7.



The OSI Model

Client Operations

Clients can upload and download data from the SAFE Network. This section outlines how these operations are conducted.

Resource Identifiers

Clients that wish to download data from the SAFE Network require software that can translate SAFE resource identifiers to endpoints on the SAFE Network, much like how browsers translate http URLs to endpoints on a server. Downloading data requires no special permissions or access, just software that can locate and interpret the data on the network.

Resources are stored on the network as content addressable resources. The identifier for these resources are SHA3–256 hashes of the resource content. This 256 bit identifier is used to retrieve a resource from the network (thus acting similarly to an IP address), allowing the client to specify which part of the network may be able to serve their request.

The 256 bit resource identifier may be represented in a human-friendly form using the built-in SAFE DNS, such as

`safe://www.userX/video.mp4` . This can be converted to the 256 bit identifier for the file by a lookup on the SAFE Network using software that can interpret SAFE DNS records.

Self Encryption

Resources on the SAFE Network are never more than 1 MB each. Clients working with files larger than 1 MB will have their data automatically split into 1 MB chunks that are then distributed across the network.

This means a typical file on the network consists of several parts: *chunks* which are individual 1 MB portions after the file is split, and a *datamap* which stores the identifier of each portion of the file. The network sees the datamap as just another chunk.

The client keeps a record of the resource identifier for the datamap. Thus, by initially retrieving that single resource (ie the datamap) the entire file can be retrieved despite being spread over many individual resources. This allows any resource to be addressed by a single resource identifier.

The datamap also acts as an encryption key for the chunks it refers to. Chunks are encrypted by that key, so vaults cannot read individual chunks to gain an insight to part of the original. This is known as Self Encryption.

Additionally, the file may be encrypted by the client before being uploaded using the encryption option built-in to the client software. This uses a unique secure key derived from the authentication the user has with the network. This means the key to decrypt the file never leaves the client and is never exposed to the network, allowing

extremely secure storage of data which cannot be decrypted by any vault on the network, even with access to the datamap.

Splitting the file via self encryption has many benefits.

- chunks on their own are not useful, so all chunks are fungible and of equal value to the network
- chunks for a file can be downloaded in parallel by clients, increasing performance
- chunks have unique and widely distributed names, so cannot easily be correlated with each other
- wide distribution of chunks greatly reduces the chance of data loss when individual vaults go offline
- blacklisting based on content is not possible

Immutable Data

Resource identifiers are determined by the content of that resource. This means two identical copies of a resource will have the same identifier (eg the same photo kept on my phone and my laptop). This comes with several benefits:

- resource identifiers are universal, unique and permanent (unlike urls on the current internet)
- resources cannot be duplicated on the network, increasing storage efficiency
- caching rules for resources are extremely simple and efficient
- encrypted resources are unique to the key that encrypts them, so cannot be decrypted by the vaults that store them

Network Traversal

When a client connects to the network they are allocated a session identifier. The session identifier determines which vault acts as their entry point to the network.

When the client requests a resource, the request is sent to their entry point vault, which is then routed via several other vaults on the network before finally reaching the vault storing the data being requested. The data is then passed back along the route to the client.

This routing mechanism traverses a 256 bit namespace progressively via closest XOR distance.

The resource being requested has a unique 256 bit identifier determined by the self encryption process. The vault acting as the entry point for the client also has a unique 256 bit identifier (as do all vaults).

The chunk identifier is XORed with the entry point vault identifier. The resulting value is the XOR distance between the chunk and the vault.

If a neighbouring vault has an identifier with a smaller XOR distance to the chunk, the request is passed to that vault.

This continues until there is no vault with a closer XOR distance to the chunk.

The chunk will be stored in the vault with the identifier closest to the chunk identifier (measured using XOR distance).

That vault checks if it has a copy of the requested chunk and returns the response back along the route the request came from.

This process forms a request chain where each vault only knows the details of the node one step away in the chain. The original requester and the vault storing the chunk are separated by this chain of vaults, making the request anonymous.

The same network traversal happens when storing chunks. Chunks enter the network via their entry point vault, and are then passed on until they reach the closest vault, which then stores the chunk.

Nodes along the route can cache the chunk so if another request for it is made later, perhaps by a different route, it can be responded to

sooner without having to extend the request all the way to the final storage vault.

Messaging

Email and instant messaging are ubiquitous experiences of the current internet. The SAFE Network facilitates messaging which can replace imap / smtp / xmpp servers.

A message may be stored on the network the same way any resource may be stored. To ensure the message remains private, the message may be encrypted prior to sending using the recipients key (that key can also be stored as a resource on the SAFE Network). By the same method, connections between contacts (like the gpg web of trust) may also be represented on the network by resources. This collection of resources form the foundation for a secure messaging platform.

The only remaining step is to notify the recipient when they have received a message.

On the current internet, new messages are typically presented to the user by adding a message to their inbox. Since an inbox is simply a list of messages, this list may be represented as a resource on the network. However, in the case of the SAFE Network, the inbox is created not as immutable data, but as mutable data. This second data type has a fixed identifier which does not depend on the content of the resource. This, combined with a permissions system, allows data at that identifier to be updated.

This data type allows the recipient to be notified of their new message by updating the data located at the identifier for the recipients inbox. The inbox resource can be created with permission for anyone to append new data. The sender of the message appends the identifier of their message to the inbox of the recipient, and the recipient is able to locate the new message.

This messaging system facilitates features such as email and instant messaging, but also introduces scope for other systems based on messaging such as payments, smart contracts, social networking, inter-process signalling, dynamic web content, stun servers for webrtc...

Mutable Data

Mutable data is the second and final data type present on the network. It allows data at a fixed location on the network to be modified.

A strong permissions layer allows the owner of the mutable data to specify who may change it and how they may change it.

The validation of ownership and modification of mutable data is performed by the network using digital signatures. The owner of mutable data can specify which keys are permitted or denied modifications to the data.

Each key is permitted or denied specific actions to the existing content. These actions are either 'update' or 'append'.

By combining clearly defined permissions with cryptographically secure signatures, the modifications to mutable data may be strictly controlled by the owner.

The content of mutable data may point to other mutable data, allowing the creation of chains of mutable data that can be used for many purposes such as version control and branching, verifiable history and data recovery.

Network Operations

There are several operations performed by the network that allow clients to store and retrieve data securely and reliably. These operations give rise to an autonomous self-healing network which is resistant to attack.

Close Group Consensus

Nodes on the network (known as vaults) are primarily responsible for storing chunks of data. The availability of chunks is critical to the success of the network.

Since anyone may add or remove their vaults from the network at any time, the network must be able to detect and respond to malicious

behaviour before it results in the loss of data. This requires enforcement of a set of rules governing acceptable behaviour of vaults. Vaults which don't comply with the rules are rejected from the network and are not responsible for client data.

The enforcement of the rules occurs through a process called Close Group Consensus.

Vaults form groups which coordinate with each other to reach consensus about the state of the data on the network. Any vault which does not comply with the consensus of the group is rejected and replaced by a different vault.

The rules that groups enforce are that data:

- can be stored
- can be retrieved
- has not been modified

This depends on vaults in the group having:

- space available to store new data
- bandwidth available to relay that data when it's requested
- sufficient participation in achieving consensus with the rest of the group

Vault Naming

All vaults are allocated a random unique 256 bit identifier by the network upon joining or rejoining. Vaults that are close together (measured by the XOR distance between their identifiers) form groups. The vaults in a group work together to form consensus about data on the network so it may be stored and retrieved. Groups are formed in sets of between 8 and 22 close vaults. The more vaults on the network, the greater the number groups on the network.

If a majority of vaults in a group are dishonest, the data in that group is vulnerable to corruption.

This means the group size is a trade off between being large enough that it's difficult to gain control of a majority of vaults in the group and being small enough that consensus can be reached quickly.

The security of the network primarily arises because the attacker *may not choose the identifier for their vault*. They must join and leave repeatedly until the network allocates their vault an identifier in the group they are attempting to control. They must do this for a majority of vaults in a group in order to control consensus for that group.

The difficulty of controlling a group is thus determined by the size of the network, becoming proportionally harder as the network grows in size. The larger the network the more groups there are; the more groups there are the harder it is to join any one specific group.

Overcoming the network-allocated-identifier for new vaults is the primary source of difficulty in trying to abuse the consensus mechanism.

Disjoint Sections

A practical consideration of the formation of groups is the efficiency of inter-group messaging (ie consensus of group membership rather than consensus of individual vault behaviour).

Groups are formed based on the similarity of the leading bits of their identifier (this portion of the identifier is called the Prefix for the Section).

This makes the coordination of vaults leaving or joining the group much simpler.

If groups were to retain a constant membership of 8 vaults, there's a need to reorganizing individual vaults between groups as new vaults join and leave. This may have a cascading effect on nearby groups.

Rather than do this cascading reorganization, groups may vary in size between 8 and 22 vaults. If a group becomes larger than 22 vaults it's split into two new groups, and if it becomes less than 8 vaults it's merged back into the nearest group.

This method of organizing groups is called Disjoint Sections.

Churn

Vaults are established in random locations on the network to prevent attacks on the group consensus mechanism. This security is further strengthened by intermittent relocation of vaults. An attack on the group must happen before relocation happens.

A vault is relocated by the network by allocating it a new random identifier. This causes it to leave its existing group and become part of a new group. Any chunks it was previously storing become the responsibility of a different vault and is automatically handled by the group consensus mechanism. The relocated vault must now store chunks that are closest to its new identifier and continue building consensus with a new group of vaults.

This mechanism is known as churn and is an extension of the same process that happens when new vaults join the network or existing vaults leave.

Farming

The incentive for vault operators to join the network and cooperate toward the goal of secure data storage is motivated by a network token. This token (called safecoin) can be redeemed for resources on the network or used to engage with other resources offered on the network. The motivation for the token is similar to that of blockchains—to ensure cooperation participation is a more rational course of action than uncooperative participation.

Tokens are represented as mutable data on the network. The network defines a total of 2^{32} mutable data resources as safecoin objects, which initially have no owner so do not exist as far as the token economy is concerned.

The network intermittently allocates unowned safecoins to network participants via a Proof Of Resource mechanism, which causes the total number of tokens to grow over time, acting as a bootstrapping mechanism for the token and network storage economy.

The total number of safecoins allocated may also shrink when safecoins are exchanged for network resources. Users must exchange safecoin to store data on the network. This involves submitting safecoin to the network which will then clear the safecoin owner and allocate network storage space for that user to consume. This mechanism is known as coin recycling and provides a counterbalance to the growth of safecoin allocated through Proof Of Resource.

Safecoin is transferred between users by digitally signing a transfer of ownership for the safecoin mutable data resource to the recipients key. The digital signature is easily verified using the key of the existing owner stored in the mutable data resource. Once the owner is updated the transfer is complete, making safecoin transfers extremely fast, efficient and secure.

Proof Of Resource

When cooperative behaviour from vaults is observed by the network, the vault earns the right to claim ownership of a safecoin. The specific safecoin identifier for each claim is randomly generated by the network.

If the safecoin for that identifier is not currently owned, the network assigns it to the owner of the vault, thus minting a new safecoin into the economy.

If the safecoin for that identifier is already owned no further action is taken.

In this sense, much like blockchain proof of work, safecoin farming has an element of chance that should lead to an even distribution proportional to the resources supplied.

The rate at which safecoin is allocated is adjusted in a similar way to the blockchain difficulty mechanism. The adjustment aims to balance the availability of resources on the network. It encourages the provision of more resources when needed and discourages wasteful excess by reducing reward during times of oversupply.

The safecoin disbursement algorithm has not yet been formally specified or implemented.

The network defines cooperative behaviour as the reliable supply of bandwidth and storage space, and continued involvement in reaching consensus. Unlike blockchain Proof Of Work, where end users receive diminishing returns from further growth in mining capacity, the Proof Of Resource mechanism continues to provide utility to end users proportional to the growth of network resources.

Conclusion

The SAFE Network is an autonomous network for reliable data storage and communications. The combination of a robust data storage and messaging system forms a secure and private alternative to much of the existing infrastructure of the internet.

Data is stored efficiently and reliably on the network using content addressable resource identifiers and self encryption.

Data is retained without corruption and may be retrieved at any time through the use of close group consensus and disjoint sections.

A network allocated token to incentivise the supply of resources within the agreed rules of the system discourages malicious behaviour and ensures the sustainable supply of resources for future clients. The distribution is based on a Proof Of Resource mechanism which is difficult to cheat and exhibits positive externality.

The network improves in speed, security and reliability as more vaults are added, making rapid increases in scale a benefit rather than a concern.

End users benefit from secure-by-default modules and a flexible permissions layer to control access in the manner best suit their needs. The SAFE Network combines many individual modules to create a network with Secure Access For Everyone.

MaidSafe Reddit | MaidSafe Twitter | MaidSafe Github

. . .

That's all. If you find anything missing or wrong here then shoot that in the comments.

Thanks for reading;)



About the Author

Vaibhav Saini is a Co-Founder of ***TowardsBlockchain***, an MIT Cambridge Innovation Center incubated startup.

He works as Senior blockchain developer and has worked on several blockchain platforms including Ethereum, Quorum, EOS, Nano, Hashgraph, IOTA etc.

He is currently a sophomore at IIT Delhi.

Learned something? Press and hold the 🖐️ to say “thanks!” and help others find this article.

Hold down the clap button if you liked the content! It helps me gain exposure .

Want to learn more? Checkout my previous articles.

Setting up your first distributed private storage network on IPFS: Part 1

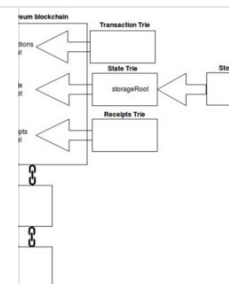
IPFS Private Storage Network Series
medium.com

HackPedia: 16 Solidity Hacks/Vulnerabilities, their Fixes and Rea...

A Complete List of all Solidity Hacks/Vulnerabilities, their Fixes and Real...
hackernoon.com

Getting Deep Into Ethereum: How Data Is Stored In Ethereum?

In this post we will see how states and transactions are stored in Ethereum and ho...
hackernoon.com



EOS 101: Getting started with EOS, Part 1

The only blockchain which has blocktime of less than a second: 0.5 sec!
hackernoon.com



Clap 50 times and follow me on Twitter: @vasa_develop

