Independent Study - Decentralized File Systems

Shoeb Siddiqui, Sujit Gujar

July 2020

Contents

1

Motivation and Objective

Decentralized File System (DFS) seems to be a promising sub-domain of decentralization. It seems to be a relevant and prudent research interest due to its potential utility in the coming times (for content delivery and personal archives) and large investment and development.

The objective of this study is to understand the importance, fundamentals, and operation of DFS. The focus will be on blockchain-like aspects of the DFS, if any, and, most notably, the incentive mechanisms that underpin the practicality and adoption potential of these DFS.[1]

2

Resources and Papers

Auxiliary Resources:
Pre-liminaries:

- Distributed Hash Table
- Sybil resistant DHT

Articles:
Non-specific:

- What is Decentralized Storage? (IPFS,FileCoin, Sia, Storj & Swarm)
- Decentralised data storage in a blockchain future
- Why is decentralized and distributed file storage critical for a better web?
- Decentralized Storage by Consensys

Formal resources :
Non-Specific:

1. Survey paper - When Blockchain Meets Distributed File Systems: An Overview, Challenges, and Open Issues
2. Survey paper - Blockchain-based decentralized storage networks: A survey

Specific:

1. IPFS - Content Addressed, Versioned, P2P File System :
   - IPFS Non-Peer Reviewed Yellow Paper
2. Filecoin: A Decentralized Storage Network :

[1] These are our understandings and any errors are with the authors of this report and we are positive about fixing them.

- Filecoin Whitepaper from Filecoin website or similarly Filecoin Whitepaper from Coinlist website
- Filecoin website

3. Siacoin :

- Siacoin Whitepaper
- Siacoin website

4. Storj :

- Storj Whitepaper link or similarly Storj Whitepaper
- Storj website

5. Swarm :

- Swarm incentive mechanism: Swap, Swear and Swindle technical report
- Swap, Swear and Swindle ppt
- Swarm Documentation
- Swarm website

6. Metadisk :

- Metadisk Whitepaper or Metadisk Whitepaper v1.01

7. PPio :

- PPIO Whitepaper
- PPIO website

8. Maidsafe :

- Maidsafe Whitepaper
- Safecoin: The Decentralised Network Token Whitepaper
- MaidSafe Distributed File System Whitepaper
- Maidsafe website

IPFS related published papers: The following papers were not verified, and the references were taken verbatim from the survey paper When Blockchain Meets Distributed File Systems: An Overview, Challenges, and Open Issues. However, even a cursory search yields several IPFS based papers published in highly reputed conferences.

1. E. Nyaletey, R. M. Parizi, Q. Zhang, and K.-K. R. Choo, "BlockIPFS- blockchain-enabled interplanetary file system for forensic and trusted data traceability," in 2019 IEEE International Conference on Blockchain (Blockchain), 2019, pp. 18–25.
2. Y. Chen, H. Li, K. Li, and J. Zhang, "An improved P2P file system scheme based on IPFS and blockchain," in Proc. of IEEE International Conference on Big Data (Big Data), 2017, pp. 2652–2657.
3. R. Norvill, B. B. F. Pontiveros, R. State, and A. Cullen, "IPFS for reduction of chain size in Ethereum," in Proc. of IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2018, pp. 1121–1128.
4. M. Steichen, B. Fiz, R. Norvill, W. Shbair, and R. State, "Blockchain- based, decentralized access control for IPFS," in Proc. of IEEE International Conference on iThings, GreenCom, CPSCom and SmartData, 2018, pp. 1499–1506.
5. N. Nizamuddin, H. R. Hasan, and K. Salah, "IPFS-blockchain-based authenticity of online publications," in International Conference on Blockchain. Springer, 2018, pp. 199–212.
6. M. Naz, F. A. Al-zahrani, R. Khalid, N. Javaid, A. M. Qamar, M. K. Afzal, and M. Shafiq, "A secure data sharing platform using blockchain and interplanetary file system," Sustainability, vol. 11, no. 24, p. 7054, 2019.

3

Preliminaries

This section explains the preliminary concepts and notions required to appreciate Decentralized File Systems (DFS).

A DFS is essentially just file storage. Typical centralized file storage is only physically present only on one device, and the data can be accessed and controlled only locally by the device administrator. DFS stores data on multiple devices. Each device does not store the entire database, but only a part of it. However, for reliability, there may be multiple devices that store copies of the same data.

3.1 Distributed Hash Tables

Since each device only hosts a part of the database, any query must essentially *find* the device(s) that the data is present on. Distributed Hash Tables (DHTs) are widely used to coordinate and maintain metadata about the peer-to-peer system. DHTs are used to route requests to the device that has the data.

Key points to note:

- Hash table - a set of *key-value* pairs, *key* - the file (either the filename or file content, i.e., content addressable) and *value* - its location.
- DHT - distributed system - provides a lookup system similar to Hash Tables - participating nodes can effectively retrieve value given key.
- Advantage of DHT - node churn involves minimal workaround re-distributing keys; Can scale to an extremely large number of nodes.
- Can be used to build more complex services.
- Properties:
  - DHT aim - Autonomy and Decentralization, Crash Fault tolerance, Scalability.
  - Change in node membership - involves limited work - any one node needs to coordinate with only a few other nodes.
  - Byzantine Fault Tolerance is not a primary goal.
  - Deals with traditional distributed systems issues - load balancing, data integrity, and performance.
- Structure: Foundation - abstract keyspace, such as the set of 160-bit strings. Keyspace partitioning scheme splits ownership of keyspace amongst participating nodes. Overlay network connects nodes, allowing them to find the owner of any given key.

Once these components are in place, a typical use of the DHT for storage and retrieval might proceed as follows. Suppose the keyspace is the set of 160-bit strings. To index a file - key k = SHA-1 hash(filename) - message put(k, filedata) sent to any node. The message is forwarded until it reaches the owner of key k, who stores key and filedata. Any node can then retrieve filedata using message get(k) sent to any node. The message will be routed through the overlay to the node responsible for k, which will reply with the stored filedata.

  - Keyspace partitioning: Most DHTs use some variant of *consistent hashing* or *rendezvous hashing* to map keys to nodes. Both have the essential property that removing or adding one node changes only the set of keys owned by the nodes with adjacent IDs and leaves all other nodes unaffected.
    * Consistent hashing: Defines abstract notion of distance. Each node assigned a single key called an identifier. Node owns all keys for which its identifier is the closest.
    * Rendezvous hashing: Also called *Highest Random Weight* hashing. n servers. Each client hashes each server id with key k. The server with the highest resultant hash is mapped to key k.
    * Locality preserving hashing: similar keys assigned to similar objects (and mapped to closer nodes in some sense). Efficient execution of range queries. Keys and load are not uniformly distributed over the keyspace and peers.
  - Overlay network: Each node maintains a set of links to other nodes (its neighbors or routing table). Together, these links form the overlay network. A node picks its neighbors according to a certain structure, called the network's topology.

    DHT topology property: for any key k, each node either has a node ID that owns k or has a link to a node whose node ID is closer to k, in terms of some keyspace distance definition. Easy to route a message using key-based routing - "forward the message to the neighbor whose ID is closest to the key" greedy algorithm (not necessarily globally optimal).

    Two important constraints - low maximum route length, for high performance, and low maximum node degree, for low maintenance overhead. Low route length requires a high node degree. DHTs construct navigable small-world network topologies, which trade-off route length vs. network degree. The most common choice, O(log n) degree/route length, is

not optimal, but such topologies typically allow more flexibility in the choice of neighbors. Many DHTs use that flexibility to pick neighbors close in terms of latency in the underlying physical network.

Aside from routing, there exist many algorithms that exploit the structure of the overlay network for sending a message to all nodes, or a subset of nodes, in a DHT.

IPFS yellow paper, discusses three prominent DHTs:

1. Kademlia DHT
   - Efficient lookup - log(n) nodes contacted on average.
   - Low coordination overhead.
   - Attack resistance by preferring long-lived nodes.
   - Wide usage in P2P.
2. Coral DHT
   - Stores addresses to peers who can provide the data blocks.
   - Doesn't ignore "far" nodes that have the data and doesn't force "near" nodes to store it unnecessarily; XOR-distance is considered.
   - Uses $get\_any\_values(key)$ instead of $get\_values(key)$, as nodes needs only a single peer.
   - Avoids hotspots by distributing only a subset of values to the "nearest" nodes.
   - Uses hierarchy of clusters, for "finding nearby data without querying distant nodes, reducing lookup latency.
3. S/Kademlia DHT: Extends Kademlia to protect against malicious attacks.
   - Schemes (like PoW) to secure NodeId generation and prevent Sybill attacks (make Sybill expensive).
   - Nodes lookup values over disjoint paths - ensures honest nodes can connect to each other even with a large fraction of adversaries present. Success rate of 0.85 with 0.5 adversarial fraction.

3.2 Block Exchanges - BitTorrent

P2P file-sharing system - coordinates networks of untrusting peers to cooperate in distributing file pieces to each other.

- Quasi tit-for-tat strategy - rewards contributing nodes - punishes solely leeching nodes
- Peers track availability - prioritize sending rarest pieces first - takes the load off seeds
- Standard tit-for-tat strategy is vulnerable to exploitative bandwidth sharing strategies. Propshare - different peer bandwidth allocation strategy - better resists exploitative strategies - improves the performance of swarms.

3.3 Version Control Systems - Git

This section is taken verbatim from the IPFS yellow paper. Version Control Systems provide facilities to model files changing over time and distribute different versions efficiently. The popular version control system Git provides a powerful Merkle DAG object model that captures changes to a filesystem tree in a distributed-friendly way.

- Immutable objects represent Files (blob), Directories (tree), and Changes (commit).
- Objects are content-addressed by the cryptographic hash of their contents.
- Links to other objects are embedded, forming a Merkle DAG. This provides many useful integrity and work-flow properties.
- Most versioning metadata (branches, tags, etc.) are simply pointer references, and thus inexpensive to create and update.
- Version changes only update references or add objects.
- Distributing version changes to other users is simply transferring objects and updating remote references.

Merkle Directed Acyclic Graph – similar but a more general construction than a Merkle Tree. Deduplicated, does not need to be balanced, and non-leaf nodes contain data. Here, deduplication, I believe, refers to the fact that a node in DAG can be referenced multiple other nodes, whereas, in a tree, all nodes are only referenced once; to create a tree where the same content is referred to by multiple nodes, multiple duplicate nodes with the same content would need to be created; this duplication is not required in a DAG, hence deduplicated. Also, DAGs are not required to have a single *root* node, whereas trees do.

3.4 Self-Certified Filesystems - SFS

The Self-certifying File System (SFS) addresses the issue of key management in cryptographic filesystems and proposes separating key management from file system security.

- SFS introduced a technique for building Self-Certified Filesystems.

- Remote filesystems addressed using: $/sfs/[Location] : [HostID]$, where $[Location]$ is server network address (IP address or DNS Name) and $[HostID] = hash(public\_key \| Location)$
- The name of an SFS file system certifies its server. The client can verify that he is actually talking to the legitimate server using the public key offered by the server.
- Once the client has verified the server, a secure channel is established (using the now known public key of the server), and the actual file access takes place.
- All SFS instances share a global namespace where name allocation is cryptographic, not gated by any centralized body.

4

IPFS - Content Addressed, Versioned, P2P File System

4.1 Introduction

The motivation behind IPFS:

- HTTP is the most successful "distributed system of files" ever deployed.
- Industry has gotten away with using HTTP this long because moving small files around is relatively cheap, even small organizations with lots of traffic.
- We are entering a new era of data distribution with new challenges, which can be boiled down to lots of data, quickly accessible everywhere with reliability.
- Pressed by critical features and bandwidth concerns, we have already given up HTTP for different data distribution protocols. The next step is making them part of the Web itself.

IPFS could push the web to new horizons, where publishing valuable information does not impose hosting it on the publisher but upon those interested, where users can trust the content they receive without trusting the peers they receive it from, and where old but important files do not go missing.

IPFS use-cases include global filesystem, personal sync folder, data sharing system, versioned package manager, root/boot filesystem of a VM, database, linked (and encrypted) communications platform, integrity checked/encrypted CDN, web CDN, Permanent Web. IPFS implementations target: an IPFS library to import in your own applications, command-line tools to manipulate objects directly, mounted file systems, using FUSE, or as kernel modules.

IPFS Salient Points:

- IPFS is similar to the Web, but IPFS could be seen as a single BitTorrent swarm.
- IPFS provides a high throughput content-addressed block storage model, with content-addressed hyperlinks, forming a generalized Merkle DAG, upon which one can build versioned file systems, blockchains, and even a Permanent Web.
- IPFS is peer-to-peer; no nodes are privileged. IPFS nodes store IPFS objects in local storage. Nodes connect to each other and transfer objects. These objects represent files and other data structures. IPFS has no single point of failure, and nodes do not need to trust each other.

- IPFS combines a distributed hash table, an incentivized block exchange, version control systems, and a self-certifying namespace. The contribution of IPFS is simplifying, evolving, and connecting proven techniques into a single cohesive system, greater than the sum of its parts.

4.2 IPFS Sub-Protocols

The IPFS protocol is divided into a stack of sub-protocols responsible for different functionality.

Identities Manage node identity generation and verification. Nodes are identified by a NodeId, the cryptographic hash of a public-key, created with S/Kademlia's static crypto puzzle. Users are free to instantiate a "new" node identity on every launch, though that loses accrued network benefits.

Network Manages connections to other peers, uses various underlying network protocols. Configurable. IPFS can use any network; it does not rely on or assume access to IP. The IPFS network stack features:

- Transport: IPFS can use any transport protocol.
- Reliability: IPFS can provide reliability if underlying networks do not provide it.
- Connectivity: IPFS also uses ICE NAT traversal techniques.
- Integrity: optionally checks the integrity of messages using a hash checksum.
- Authenticity: optionally checks the authenticity of messages using HMAC with the sender's public key.

Routing Maintains information to locate specific peers and objects. Responds to both local and remote queries. IPFS nodes require a routing system that can find (a) other peers' network addresses and (b) peers who can serve particular objects. IPFS achieves this using a DSHT based on S/Kademlia and Coral. The IPFS DHT makes a distinction for values stored based on their size. Small values are stored directly on the DHT. For values larger, the DHT stores references, which are the NodeIds of peers who can serve the block.

Block Exchange A novel block exchange protocol (BitSwap) that governs efficient block distribution. Modeled as a market, weakly incentivizes data replication. BitSwap peers are looking to acquire a set of blocks (want_list) and have another set of blocks to offer in exchange (have_list). In the base case, BitSwap nodes have to provide direct value to each other in the form of blocks. In some cases, nodes must work for their blocks. In the case that a node has nothing that its peers want (or nothing at all), it seeks the pieces its peers want, with lower priority than what the node wants itself. This incentivizes nodes to cache and disseminate rare pieces, even if they are not interested in them directly.

BitSwap Credit The protocol must also incentivize nodes to seed when they do not need anything in particular, as they might have the blocks others want. Thus, BitSwap nodes send blocks to their peers optimistically, expecting the debt to be repaid. However, leeches (free-loading nodes that never share) must be protected against. A simple credit-like system solves the problem:

1. Peers track their balance (in bytes verified) with other nodes.
2. Peers send blocks to debtor peers probabilistically, according to a function that falls as debt increases.

If a node decides not to send to a peer, the node subsequently ignores the peer for an ignore_cooldown timeout.

BitSwap Strategy The strategy of a node can be represented as the probability that a node will send to a requester, given metrics on the requester and its relationship with the node. The choice of the function that determines this probability should aim to:

1. maximize the trade performance for the node and the whole exchange
2. prevent freeloaders from exploiting and degrading the exchange
3. be effective with and resistant to other, unknown strategies
4. be lenient to trusted peers

One choice of function that works in practice is a sigmoid, scaled by a debt ratio, $r = \frac{bytesSent}{bytesRecv+1}$, and the probability of sending to a debtor, $P(send|r) = 1 - \frac{1}{1+exp(6-3r)}$. This function drops off quickly as the nodes' debt ratio surpasses twice the established credit.

The debt ratio is a measure of trust: lenient to nodes that have exchanged lots of data and merciless to unknown nodes. This (a) provides Sybill resistance, (b) protects previously successful trade relationships, and (c) eventually chokes relationships that have deteriorated until they improve.

BitSwap Ledger BitSwap nodes keep ledgers accounting the transfers with other nodes. When activating a connection, BitSwap nodes exchange their ledger information. If it does not match exactly, the ledger is reinitialized from scratch. It is possible for malicious nodes to purposefully "lose" the Ledger, hoping to erase debts; In this case, the partner node is free to count it as misconduct and refuse to trade. Only the current ledger entries are useful. Nodes are also free to garbage collect ledgers as necessary.

BitSwap Specification Sketch of the lifetime of a peer connection:

1. Open: peers send ledgers until they agree.
2. Sending: peers exchange want_lists and blocks.
3. Close: peers deactivate a connection.
4. Ignored: a peer is ignored (for the duration of a timeout) if a node's strategy avoids sending

If a transmission verification fails, the sender is either malfunctioning or attacking the receiver. The receiver is free to refuse further trades.

Objects A Merkle DAG of content-addressed immutable objects with links used to represent arbitrary data structures, e.g., file hierarchies and communication systems. IPFS builds a Merkle DAG, where links between objects are cryptographic hashes of the targets embedded in the sources. This is a generalization of the Git data structure. Merkle DAGs provide IPFS content addressing, tamper resistance, and deduplication. IPFS Merkle DAG is flexible. The only requirements are that object references be (a) content addressed and (b) encoded in the format specified. IPFS grants applications complete control over the data field; applications can use any custom data format they choose, which IPFS may not understand. The separate in-object link table allows IPFS to:

- List all object references in an object.
- Resolve string path lookups, such as foo/bar/baz. Given an object, IPFS resolves the first path component to a hash in the object's link table, fetches that second object, and repeats with the next component. Thus, string paths can walk the Merkle DAG no matter what the data formats are.
- Resolve all objects referenced recursively.

A raw data field and a common link structure are the necessary components for constructing arbitrary data structures on top of IPFS. IPFS Merkle DAG allows systems to use IPFS as a transport protocol for more complex applications.

Paths IPFS objects can be traversed with a string path API. The path format is: /ipfs/hash-of-object/name-path-to-object. The /ipfs prefix allows mounting. The second path component (first within IPFS) is the hash of an object. There is no global root. We simulate the root with content addressing. All objects are always accessible via their hash.

Local Objects IPFS clients require some local storage, an external system on which to store and retrieve local raw data for the objects IPFS manages. Ultimately, all blocks available in IPFS are in some node's local storage.

Object Pinning Nodes who wish to ensure the survival of particular objects can do so by pinning the objects. This ensures local storage of the object. Pinning can be done recursively to pin down all linked descendent objects as well. Useful to persist files, including references. This also makes IPFS a Web where links are permanent, and Objects can ensure the survival of others they point to.

Publishing Objects Anyone can publish an object by simply adding its key to the DHT, adding themselves as a peer, and giving other users the object's path.

Object-level Cryptography IPFS is equipped to handle object-level cryptographic operations. An encrypted or signed object is wrapped in a special frame that allows encryption or verification of the raw bytes. Cryptographic operations change the object's hash, defining a different object. It is possible to have a parent object encrypted under one key and a child under another or not at all. This secures links to shared objects.

Files Versioned file system hierarchy inspired by Git. IPFS also defines a set of objects for modeling a versioned filesystem on top of the Merkle DAG. Different from Git to introduce:

- fast size lookups
- large file deduplication
- embedding commits into trees

Conversion between IPFS and Git Objects is possible, losslessly using Git objects.
IPFS Object Model:

- Block/Blob: An addressable variable-sized unit of data; represents a file. They have no links.
- List: A ordered collection of blocks or other lists; represents a large file.
- Tree: A collection of blocks, lists, or other trees; represents a directory, a map of names to hashes.
- Commit: Represents a snapshot in the version history of an object.

Version Control As long as a single commit and all the child objects it references are accessible, all preceding versions are retrievable, and the full history of the filesystem changes can be accessed.

Filesystem Paths IPFS objects can be traversed with a string path API. Trees are restricted to have no data in order to represent them as directories. Commits can either be represented as directories or be hidden from the filesystem entirely.

Splitting Files into Lists and Blob IPFS offers options to split large files: Rabin fingerprints, rsync, and user-specified.

Path Lookup Performance Retrieving each object requires looking up its key in the DHT, connecting to peers, and retrieving its blocks. The overhead is mitigated by tree caching and flattened trees.

Naming A self-certifying mutable name system. Mutable naming is a way to retrieve the mutable state at the same path. Without it, all communication of new content must happen off-band, sending IPFS links.

Self-Certified Names Naming scheme from SFS gives a way to construct self-certified names in a cryptographically assigned global namespace that are mutable.

- Assign every user a mutable namespace at: /ipns/NodeId=hash(node.PubKey)
- User can publish an object to this path signed by her private key.
- When other users retrieve the object, they can check the signature matches the public key and NodeId. This verifies the authenticity of the Object published by the user, achieving mutable state retrival.

The ipns (InterPlanetary Name Space) is a separate prefix. It is not a content-addressed object; it relies on the Routing system. The process is (1) publish the object as a regular immutable IPFS object, (2) publish its hash on the Routing system as a metadata value: routing.setValue(NodeId, <ns-object-hash>). It is advised to publish a commit object so that clients may be able to find old names.

Human-Friendly Names IPNS is not very user friendly, as it exposes long hash values as names. IPFS improves the user-friendliness of IPNS with 1) Peer Links, 2) DNS Records, 3) Pronounceable Identifiers, and 4) Name Shortening Services.

5

Filecoin: A Decentralized Storage Network

5.1 Introduction

Filecoin is a decentralized storage network that turns cloud storage into an algorithmic market. The market runs on a blockchain with a native protocol token (also called "Filecoin"), which miners earn by providing storage to clients. Clients spend Filecoin hiring miners to store or distribute data. Miners compete to mine blocks with sizable rewards. Mining power is proportional to active storage. It creates a powerful incentive for miners to amass as much storage as they can and rent it out to clients. The protocol weaves these amassed resources into a self-healing storage network. Robustness is achieved by replicating and dispersing content, automatically detecting and repairing replica failures. Clients can select replication parameters to protect against different threat models. Content is encrypted. Filecoin works as an incentive layer on top of IPFS

This paper: introduces Filecoin Network, formalizes decentralized storage network (DSN), constructs Filcoin as a DSN, introduces a novel useful-work consensus based on sequential proofs-of-replication and storage as a measure of power, formalizes verifiable markets, and constructs two markets, a Storage Market and a Retrieval Market, which govern how data is written to and read from Filecoin, respectively, discusses use cases, connections to other systems, and how to use the protocol.

Connection between IPFS and Filecoin As per docs.ipfs.io: Filecoin and IPFS are two separate, complementary protocols, both created by Protocol Labs. IPFS allows peers to store, request, and transfer verifiable data with each other, while Filecoin is designed to provide a system of persistent data storage. Under Filecoin's incentive structure, clients pay to store data at specific levels of redundancy and availability, and miners earn payments and rewards by continuously storing data and cryptographically proving it. IPFS addresses and moves content, while Filecoin is an incentive layer to persist data.

5.2 Decentralized Storage Network

A DSN scheme is essentially:

- Put(data) - key: Clients execute the Put protocol to store data under a unique identifier key.
- Get(key) - data: Clients execute the Get protocol to retrieve data that is currently stored using a key.
- Manage(): The network of participants coordinates via the Manage protocol to: control the available storage, audit the service offered by providers, and repair possible faults.

A DSN scheme must guarantee:

- Fault tolerance
  - Management faults: They are byzantine faults caused by participants in the Manage protocol. A DSN scheme relies on the fault tolerance of its underlining Manage protocol. Violations on the faults tolerance assumptions for management faults can compromise the liveness and safety of the system.

  - Storage faults: They are byzantine faults that prevent clients from retrieving the data: i.e., Storage Miners lose their pieces, Retrieval Miners stop serving pieces. A successful Put execution is (f, m)-tolerant if it results in its input data being stored in m independent storage providers (out of n total), and it can tolerate up to f byzantine providers. A Get execution on stored data is successful if there are fewer than f faulty storage providers.

- Properties
  - Data integrity: This property requires that no bounded adversary A can convince clients to accept altered or falsified data at the end of a Get execution.
  - Retrievability: This property captures the requirement that, given our fault-tolerance assumptions, if some data has been successfully stored and storage providers continue to follow the protocol, then clients can eventually retrieve the data.
  - Public verifiable: For each successful Put, the network of storage providers can generate a proof that the data is currently being stored. The Proof-of-Storage must convince any efficient verifier, which only knows the key and does not have access to data.
  - Auditable: Generates a verifiable trace of operation that can be checked in the future to confirm storage was indeed stored for the right duration of time.
  - Incentive-compatible: Storage providers are rewarded for successfully offering storage and retrieval service or penalized for misbehaving, such that the storage providers' dominant strategy is to store data.

5.3 Proof-of-Replication and Proof-of-Spacetime

Proofs-of-Storage (PoS) schemes such as Provable Data Possession (PDP) and Proof-of-Retrievability (PoR) schemes allow a user (i.e., the verifier V) who outsources data D to a server (i.e., the prover P) to repeatedly check if the server is still storing D. The server generates probabilistic proofs of possession by sampling a random set of blocks and transmits a small constant amount of data in a challenge/response protocol with the user. PDP and PoR schemes only guarantee that a prover had possession of some data at the time of the challenge/response. In Filecoin, we need stronger guarantees to prevent three types of attacks that malicious miners could exploit to get rewarded for storage they are not providing: Sybil attack, outsourcing attacks, generation attacks.

Both PoRep and PoSt use Collision-resistant hashing and zk-SNARKS (with its Setup-Prove-Verify framework).

Proof-of-Replication (PoRep) is a novel Proof-of-Storage that allows a server (i.e., the prover P) to convince a user (i.e., the verifier V) that some data D has been replicated to its own uniquely dedicated physical storage. Our scheme is an interactive protocol, where the prover P: (a) commits to store n distinct replicas (physically independent copies) of some data D, and then (b) convinces the verifier V that P is indeed storing each of the replicas via a challenge/response protocol. The replica is an independent physical copy of some data D, unique to P.

Proof-of-Spacetime (PoSt) is essentially PoRep repeatedly performed, with the previous output being used as one of the inputs, the other input being the seed from the blockchain. A verifier can check if a prover is storing her/his outsourced data for a range of time. The intuition is to require the prover to (1) generate sequential Proofs-of-Storage (in our case Proof-of-Replication) as a way to determine time (2) recursively compose the executions to generate a short proof.

Seal operation The role of the Seal operation is to (1) force replicas to be physically independent copies by requiring provers to store a pseudo-random permutation of D unique to their public key, such that committing to store n replicas results in dedicating disk space for n independent replicas (hence n times the storage size of a replica) and (2) to force the generation of the replica during PoRep. Setup to take substantially longer than the time expected for responding to a challenge. Point 2 is to ensure that another replica of the data isn't used to generate the replica queried.

5.4 Filecoin : A DSN Construction

Any user can participate as a Client, a Storage Miner, and/or a Retrieval Miner.

- Clients pay to store data and to retrieve data in the DSN, via Put and Get requests.
- Storage Miners provide data storage to the network. Storage Miners participate in Filecoin by offering their disk space and serving Put requests. To become Storage Miners, users must pledge their storage by depositing collateral proportional to it. Storage Miners respond to Put requests by committing to store the client's data for a specified time. Storage Miners generate Proofs-of-Spacetime and submit them to the blockchain to prove to the Network that they are storing the data through time. In case of invalid or missing proofs, Storage Miners are penalized and

lose part of their collateral. Storage Miners are also eligible to mine new blocks, and in doing so, they hence receive the mining reward for creating a block and transaction fees for the transactions included in the block.

- Retrieval Miners provide data retrieval to the Network. Retrieval Miners participate in Filecoin by serving data that users request via Get. Unlike Storage Miners, they are not required to pledge, commit to store data, or provide proofs of storage. It is natural for Storage Miners to also participate as Retrieval Miners. Retrieval Miners can obtain pieces directly from clients or from the Retrieval Market.

Guarantees and Requirements: The following are the intuitions on how the Filecoin DSN achieves integrity, retrievability, public verifiability, and incentive-compatibility.

- Achieving Integrity: Pieces are named after their cryptographic hash. After a Put request, clients only need to store this hash to retrieve the data via Get and to verify the integrity of the content received.
- Achieving Retrievability: In a Put request, clients specify the replication factor and the type of erasure coding desired, specifying in this way the storage to be (f, m)-tolerant. The assumption is that given m Storage Miners storing the data, a maximum of f faults are tolerated. By storing data in more than one Storage Miner, a client can increase the chances of recovery in case Storage Miners go offline or disappear.
- Achieving Public Verifiability and Auditability: Storage Miners are required to submit their proofs of storage to the blockchain. Any user in the network can verify the validity of these proofs without having access to the outsourced data. Since the proofs are stored on the blockchain, they are a trace of operation that can be audited at any time.
- Achieving Incentive Compatibility: Informally, miners are rewarded for the storage they are providing. When miners commit to store some data, then they are required to generate proofs. Miners that skip proofs are penalized (by losing part of their collateral) and not rewarded for their storage.
- Achieving Confidentiality: Clients that desire for their data to be stored privately must encrypt their data before submitting them to the network.

5.5 Filecoin Storage and Retrieval Markets

Verifiable Market A verifiable Market is a protocol with two phases: order matching and settlement.

Verifiable Market Protocol:

- Order matching
  - Participants add buy orders and sell orders to the orderbook.
  - When two orders match, involved parties jointly create a deal order that commits the two parties to the exchange and propagate it to the network by adding it to the orderbook.
- Settlement
  - The network ensures that the transfer of goods or services has been executed correctly by requiring sellers to generate cryptographic proofs for their exchange/service.
  - On success, the network processes the payments and clears the orders from the orderbook.

Storage Market Requirements:

- In-chain orderbook: It is important that: (1) Storage Miners orders are public so that the lowest price is always known to the network and clients can make an informed decision on their orders, (2) client orders must always be submitted to the orderbook, even when they accept the lowest price, in this way the market can react to the new offer. Hence, we require orders to be added in the clear to the Filecoin blockchain in order to be added to the orderbook.
- Participants commit their resources: We require both parties to commit to their resources as a way to avoid disservice: to avoid Storage Miners not providing the service and to avoid clients not having available funds. In order to participate in the Storage Market, Storage Miners must pledge, depositing collateral proportional to their amount of storage in DSN. In this way, the Network can penalize Storage Miners that do not provide proofs of storage for the pieces they committed to store. Similarly, clients must deposit the funds specified in the order, guaranteeing in this way commitment and availability of funds during settlement.
- Self-organization to handle faults: Orders are only settled if Storage Miners have repeatedly proved that they have stored the pieces for the duration of the agreed-upon time period. The Network must be able to verify the existence and the correctness of these proofs and act according to the Repair rules outlined.

The Storage Market Protocol:

- Order Matching: Clients and Storage Miners submit their orders to the orderbook by submitting a transaction to the blockchain. When orders are matched, the client sends the piece to the Storage Miner, and both parties sign a deal order and submit it to the orderbook.

- Settlement: Storage Miners seal their sectors, generate proofs of storage for the sector containing the piece, and submit them to the blockchain regularly (step 3b); meanwhile, the rest of the network must verify the proofs generated by the miners and repair possible faults.

Retrieval Market Retrieval Miners are not required to store pieces through time or generate proofs of storage. Any user in the network can become a Retrieval Miner by serving pieces in exchange for Filecoin tokens. Retrieval Miners can obtain pieces by receiving them directly from clients, by acquiring them from the Retrieval Market, or by storing them from being a Storage Miner.

Requirements:

- Off-chain orderbook: Clients must be able to find Retrieval Miners that are serving the required pieces and directly exchange the pieces after settling on the pricing. This means that the orderbook cannot be run via the blockchain - since this would be the bottleneck for fast retrieval requests - instead, participants will have only a partial view of the OrderBook. Hence, we require both parties to gossip their orders.

- Retrieval without trusted parties: The impossibility results on fair exchange remind us that it is impossible for two parties to perform an exchange without trusted parties. In the Storage Market, the blockchain network acts as a (decentralized) trusted party that verifies the storage provided by the Storage Miners. In the Retrieval Market, Retrieval Miners and clients exchange data without the network witnessing the exchange of files. We go around this result by requiring the Retrieval Miner to split their data into multiple parts, and for each part sent to the client, they receive a payment. In this way, if the client stops paying or the miner stops sending data, either party can halt the exchange. Note that for this to work, we must assume that there is always one honest Retrieval Miner.

- Payments channels: Clients are interested in retrieving the pieces as soon as they submit their payments; Retrieval Miners are interested in only serving the pieces if they are sure of receiving payment. Validating payments via a public ledger can be the bottleneck of a retrieval request; hence we must rely on efficient off-chain payments. The Filecoin blockchain must support payment channels that enable rapid, optimistic transactions and use the blockchain only in case of disputes. In this way, Retrieval Miners and Clients can quickly send the small payments required by our protocol. Future work includes the creation of a network of payment channels.

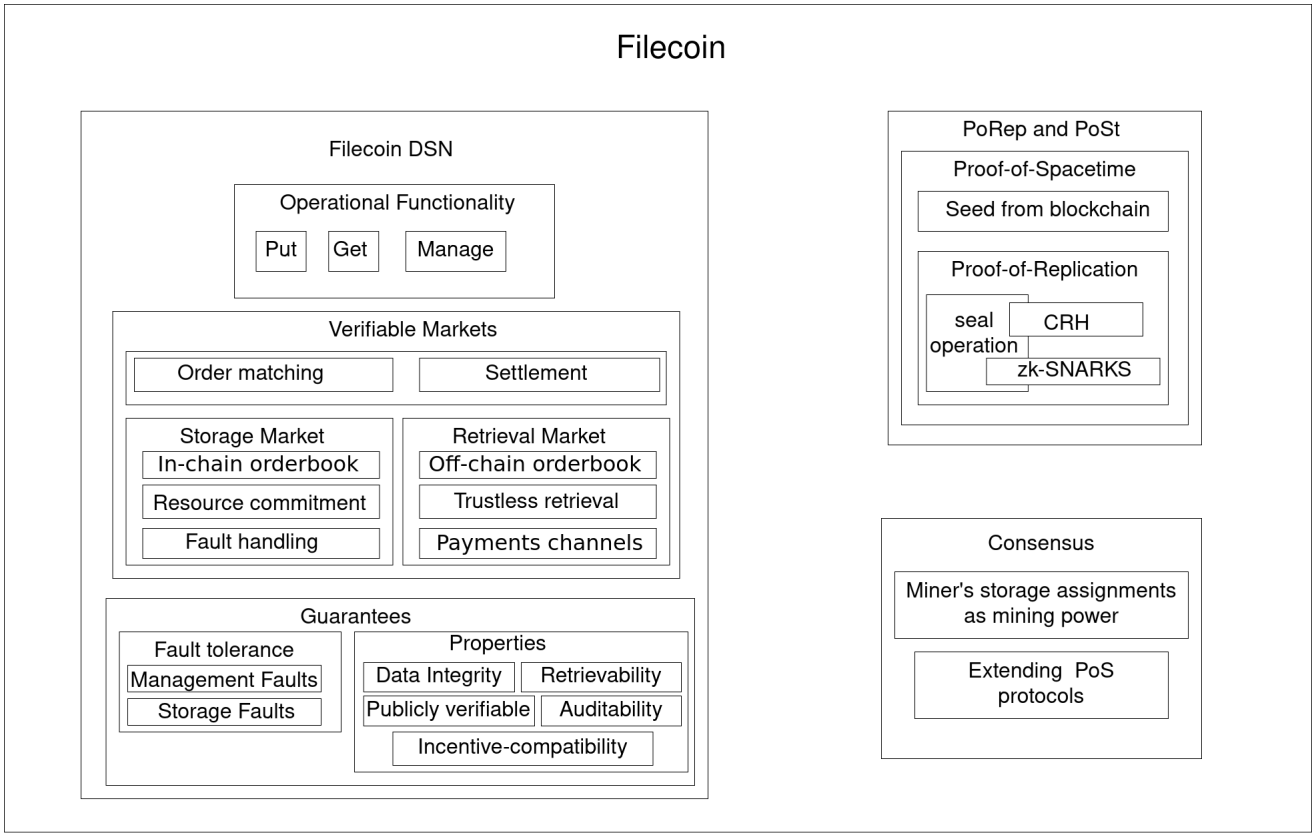
Retrieval Market Protocol

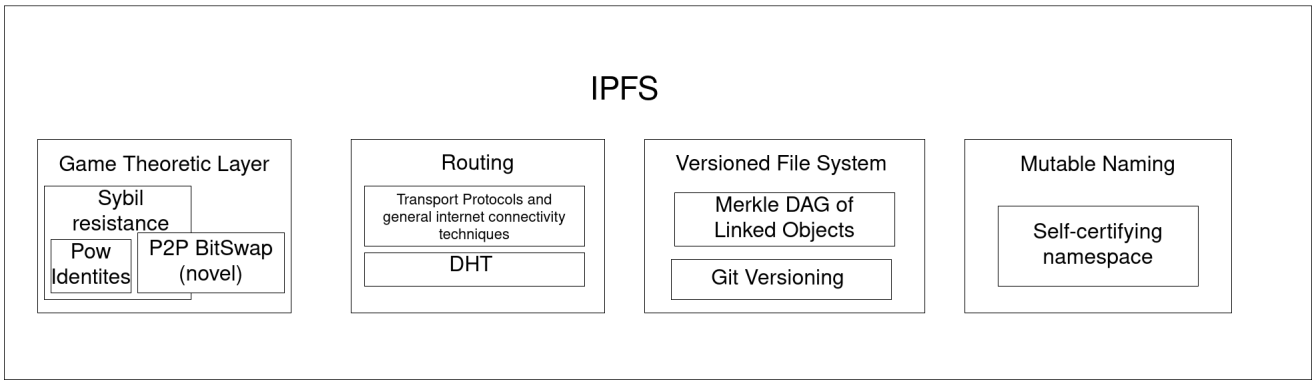
- Order Matching: Clients and Retrieval Miners submit their orders to the orderbook by gossiping their orders. When orders are matched, the client and the Retrieval Miners establish a micropayment channel.

- Settlement: Retrieval Miners send small parts of the piece to the client, and for each piece, the client sends to the miner a signed receipt (step 3). The Retrieval Miner presents the delivery receipts to the blockchain to get their rewards.

5.6 Filecoin Consensus

In Filecoin, the power of a miner at a given time is the sum of the miner's storage assignments. The influence of a miner is the fraction of the miner's power over the total power in the network. Filecoin consensus is implemented by extending Proof-of-Stake consensus protocols, where the stake is replaced with assigned storage.


Filecoin
Filecoin DSN
Operational Functionality
Put
Get
Manage
Verifiable Markets
Order matching
Settlement
Storage Market
In-chain orderbook
Resource commitment
Fault handling
Retrieval Market
Off-chain orderbook
Trustless retrieval
Payments channels
Guarantees
Fault tolerance
Management Faults
Storage Faults
Properties
Data Integrity
Retrievability
Publicly verifiable
Auditability
Incentive-compatibility
PoRep and PoSt
Proof-of-Spacetime
Seed from blockchain
Proof-of-Replication
seal operation
CRH
zk-SNARKS
Consensus
Miner's storage assignments as mining power
Extending PoS protocols


IPFS
Game Theoretic Layer
Sybil resistance
Pow Identites
P2P BitSwap (novel)
Routing
Transport Protocols and general internet connectivity techniques
DHT
Versioned File System
Merkle DAG of Linked Objects
Git Versioning
Mutable Naming
Self-certifying namespace

6

Siacoin

6.1 Introduction

- Sia enables the formation of storage contracts between peers. Contracts are agreements between a storage provider and their client, defining what data will be stored and at what price. They require the storage provider to prove, at regular intervals, that they are still storing their client's data.
- Sia itself stores only the storage contracts formed between parties, defining the terms of their arrangement.
- The host is compensated for every proof they submit and penalized for missing a proof.
- Network consensus can be used to automatically enforce storage contracts. Importantly, this means that clients do not need to personally verify storage proofs.
- Uses erasure coding to store data across multiple hosts, for high availability without excessive redundancy.
- Sia uses an M –of–N multi-signature scheme for all transactions, eschewing the scripting system entirely.
- Sia extends transactions to enable the creation and enforcement of storage contracts. Three extensions are used to accomplish this: contracts, proofs, and contract updates.

6.2 Transactions and File Contracts

- Inputs and outputs are also paired with a set of spend conditions. Inputs contain the spend conditions themselves, while outputs contain their Merkle root hash.
- Spend conditions are properties that must be met before coins are "unlocked" and can be spent. Output cannot be spent until the time lock has expired, and enough of the specified keys have added their signature.
- The spend conditions are hashed into a Merkle tree. The root hash of this tree is used as the address to which the coins are sent.
- A file contract is an agreement between a storage provider and its client. At the core of a file contract is the file's Merkle root hash. To construct this hash, the file is split into segments of constant size and hashed into a Merkle tree. The root hash, along with the total size of the file, can be used to verify storage proofs.
- Submitting a valid proof during the challenge window triggers an automatic payment to the *valid proof* address (presumably the host). If, at the end of the challenge window, no valid proof has been submitted, coins are instead sent to the *missed proof* address (likely an unspendable address in order to disincentivize DoS attacks). Contracts define a maximum number of proofs that can be missed; if this number is exceeded, the contract becomes invalid.
- If the contract is still valid at the end of the contract duration, it successfully terminates, and any remaining coins are sent to the valid proof address. Conversely, if the contract funds are exhausted before the duration elapses or if the maximum number of missed proofs is exceeded, the contract unsuccessfully terminates, and any remaining coins are sent to the missed proof address.
- Completing or missing a proof results in a new transaction output belonging to the recipient specified in the contract.
- File contracts are also created with a list of *edit conditions*, analogous to the spend conditions of a transaction. If the edit conditions are fulfilled, the contract may be modified. As these modifications can affect the validity of subsequent storage proofs, contract edits must specify a future challenge window at which they will become effective.

6.3 Proofs of Storage

- Storage proof transactions are periodically submitted in order to fulfill file contracts.
- Hosts prove their storage by providing a segment of the original file and a list of hashes from the file's Merkle tree. Each storage proof uses a randomly selected segment.
- If the host is consistently able to demonstrate possession of a random segment, then they are very likely storing the whole file.
- Susceptible to both *Block Withholding Attacks* and *Closed Window Attacks*, but neither of them are powerful attacks.

6.4 Storage Ecosystem

- Each transaction has an arbitrary data field that can be used for any type of information. This arbitrary data provides hosts and clients with a decentralized way to organize themselves. It can be used to advertise available space or files seeking a host or to create a decentralized file tracker.

- A contract requires consent from both the storage provider and their client.

- Hosts are vulnerable to denial of service attacks, which could prevent them from submitting storage proofs or transferring files. It is the responsibility of the host to protect themselves from such attacks.

- The storage proofs contain no mechanism to enforce constant uptime. There are also no provisions that require hosts to transfer files to clients upon request. One might expect, then, to see hosts holding their clients' files hostage and demanding exorbitant fees to download them. However, this attack is mitigated through the use of erasure codes.

- Payment for downloads is expected to be offered through preexisting micropayment channels. Hosts could transfer a small segment of the file and wait to receive a micropayment before proceeding.

- Clients need a reliable method for picking quality hosts. A host could repeatedly form contracts with itself, agreeing to store large "fake" files, such as a file containing only zeros. To mitigate this Sybil attack, clients can require that hosts that announce themselves in the arbitrary data section also include a large volume of time locked coins.

7

Storj

Storj is only functionally decentralized. It only intends to use decentralization for crash fault tolerance and availability. Participants have to trust others, including some entities that are central to the ecosystem. It does not offer a trustless ecosystem. It does not employ blockchains. It is not BFT in anyway.

8

Discussion and Conclusion

In this study, we have explored the domain of Decentralized File Systems (DFS). We have found that DFS operates on two layers: 1) Network and distributed file storage 2) Blockchain-enabled trust-less verifiable ecosystem. For the network and distributed file storage layer, there don't seem to be any mature alternatives to IPFS (besides Ethereum that Swarm uses). The blockchain ecosystem, however, has a few viable options. The game-theoretic aspects of DFS are primarily in the blockchain ecosystem layer.

IPFS and Filecoin are state of the art at the moment. IPFS's infrastructure is very impressive and also modular. Filecoin's use of zksnarks to key the data replica seems to be unique yet relies on micropayment channels and on-chain registry of storage contracts like most other protocols. They both are widely used and are in active development. Some issues, such as DDoS attacks and reliance on honest nodes for data retrieval (data ransom attacks), don't seem to have a simple solution. Other issues, such as Sybil attacks on the proof-of-storage-based influence, can be addressed by using a coinage-like mechanism that Siacoin uses.

Siacoin seems to be, conceptually and architecturally, very immature. Storj is not a trustless system. Metadisk attempted to be the decentralized foundation for Storj, but seems this was given up on (The paper was published in 2014). Swarm, intended to be integrated with ethereum, seems very mature and well developed but does not seem to address collusion to cheat the actual number of data replicas using zksnarks the way filecoin does. PPIO's development seems to either be inactive or very slow (PPIO may be dead). While Maidsafe seems to be under active development, it is conceptually very immature.