Filecoin: A Decentralized Storage Network

Protocol Labs

July 19, 2017

Abstract

The internet is in the middle of a revolution: centralized proprietary services are being replaced with decentralized open ones; trusted parties replaced with verifiable computation; brittle location addresses replaced with resilient content addresses; inefficient monolithic services replaced with peer-to-peer algorithmic markets. Bitcoin, Ethereum, and other blockchain networks have proven the utility of decentralized transaction ledgers. These public ledgers process sophisticated smart contract applications and transact crypto-assets worth tens of billions of dollars. These systems are the first instances of internet-wide Open Services, where participants form a decentralized network providing useful services for pay, with no central management or trusted parties. IPFS has proven the utility of content-addressing by decentralizing the web itself, serving billions of files used across a global peer-to-peer network. It liberates data from silos, survives network partitions, works offline, routes around censorship, and gives permanence to digital information.

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. Conversely, clients spend Filecoin hiring miners to store or distribute data. As with Bitcoin, Filecoin miners compete to mine blocks with sizable rewards, but Filecoin mining power is proportional to active storage, which directly provides a useful service to clients (unlike Bitcoin mining, whose usefulness is limited to maintaining blockchain consensus). This 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 that anybody in the world can rely on. The network achieves robustness by replicating and dispersing content, while automatically detecting and repairing replica failures. Clients can select replication parameters to protect against different threat models. The protocol's cloud storage network also provides security, as content is encrypted end-to-end at the client, while storage providers do not have access to decryption keys. Filecoin works as an incentive layer on top of IPFS [1], which can provide storage infrastructure for any data. It is especially useful for decentralizing data, building and running distributed applications, and implementing smart contracts.

This work:

(a) Introduces the Filecoin Network, gives an overview of the protocol, and walks through several components in detail.
(b) Formalizes *decentralized storage network* (DSN) schemes and their properties, then constructs Filecoin as a DSN.
(c) Introduces a novel class of *proof-of-storage* schemes called *proof-of-replication*, which allows proving that any replica of data is stored in physically independent storage.
(d) Introduces a novel useful-work consensus based on sequential *proofs-of-replication* and storage as a measure of power.
(e) 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.
(f) Discusses use cases, connections to other systems, and how to use the protocol.

Note: Filecoin is a work in progress. Active research is under way, and new versions of this paper will appear at https://filecoin.io. For comments and suggestions, contact us at research@filecoin.io.

Contents

List of Figures

1 Introduction

Filecoin is a protocol token whose blockchain runs on a novel proof, called *Proof-of-Spacetime*, where blocks are created by miners that are storing data. Filecoin protocol provides a data storage and retrieval service via a network of independent storage providers that does not rely on a single coordinator, where: (1) clients pay to store and retrieve data, (2) Storage Miners earn tokens by offering storage (3) Retrieval Miners earn tokens by serving data.

1.1 Elementary Components

The Filecoin protocol builds upon four novel components.

1. **Decentralized Storage Network (DSN)**: We provide an abstraction for network of independent storage providers to offer storage and retrieval services (in Section 2). Later, we present the Filecoin protocol as an incentivized, auditable and verifiable DSN construction (in Section 4).

2. **Novel *Proofs-of-Storage***: We present two novel *Proofs-of-Storage* (in Section 3): (1) *Proof-of-Replication* allows storage providers to prove that data has been *replicated* to its own uniquely dedicated physical storage. Enforcing unique physical copies enables a verifier to check that a prover is not deduplicating multiple copies of the data into the same storage space; (2) *Proof-of-Spacetime* allows storage providers to prove they have stored some data throughout a specified amount of time.

3. **Verifiable Markets**: We model storage requests and retrieval requests as orders in two decentralized verifiable markets operated by the Filecoin network (in Section 5). Verifiable markets ensure that payments are performed when a service has been correctly provided. We present the Storage Market and the Retrieval Market where miners and clients can respectively submit storage and retrieval orders.

4. **Useful *Proof-of-Work***: We show how to construct a useful *Proof-of-Work* based on *Proof-of-Spacetime* that can be used in consensus protocols. Miners do not need to spend wasteful computation to mine blocks, but instead must store data in the network.

1.2 Protocol Overview

- The Filecoin protocol is a *Decentralized Storage Network* construction built on a blockchain and with a native token. Clients spend tokens for storing and retrieving data and miners earn tokens by storing and serving data.

- The Filecoin DSN handle storage and retrieval requests respectively via two *verifiable markets*: the Storage Market and the Retrieval Market. Clients and miners set the prices for the services requested and offered and submit their orders to the markets.

- The markets are operated by the Filecoin network which employs *Proof-of-Spacetime* and *Proof-of-Replication* to guarantee that miners have correctly stored the data they committed to store.

- Finally, miners can participate in the creations of new blocks for the underlining blockchain. The influence of a miner over the next block is proportional to the amount of their storage currently in use in the network.

A sketch of the Filecoin protocol, using nomenclature defined later within the paper, is shown in Figure 1 accompanied with an illustration in Figure 2.

1.3 Paper organization

The remainder of this paper is organized as follows. We present our definition of and requirements for a theoretical DSNscheme in Section 2. In Section 3 we motivate, define, and present our *Proof-of-Replication* and *Proof-of-Spacetime* protocols, used within Filecoin to cryptographically verify that data is continuously

stored in accordance with deals made. Section 4 describes the concrete instantiation of the Filecoin DSN, describing data structures, protocols, and the interactions between participants. Section 5 defines and describes the concept of Verifiable Markets, as well as their implementations, the Storage Market and Retrieval Market. Section 6 motivates and describes the use of the *Proof-of-Spacetime* protocol for demonstrating and evaluating a miner's contribution to the network, which is necessary to extend the blockchain and assign the block reward. Section 7 provides a brief description of Smart Contracts within the Filecoin We conclude with a discussion of future work in Section 8.

Filecoin Protocol Sketch

Network

at each epoch t in the ledger $\mathcal{L}$:

1. for each new block:
   (a) check if the block is in the valid format
   (b) check if all transactions are valid
   (c) check if all orders are valid
   (d) check if all proofs are valid
   (e) check if all pledges are valid
   (f) discard block, if any of the above fails
2. for each new order $\mathcal{O}$ introduced in t
   (a) add $\mathcal{O}$ to the Storage Market's orderbook.
   (b) if $\mathcal{O}$ is a *bid*: lock $\mathcal{O}$.funds
   (c) if $\mathcal{O}$ is an *ask*: lock $\mathcal{O}$.space
   (d) if $\mathcal{O}$ is a *deal*: run Put.AssignOrders
3. for each $\mathcal{O}$ in the Storage Market's orderbook:
   (a) check if $\mathcal{O}$ has expired (or canceled):
   - remove $\mathcal{O}$ from the orderbook
   - return unspent $\mathcal{O}$.funds
   - free $\mathcal{O}$.space from AllocTable

   (b) if $\mathcal{O}$ is a *deal*, check if the expected proofs exist by running Manage.RepairOrders:
   - if one missing, penalize the $\mathcal{M}$'s pledge collateral
   - if proofs are missing for more than Δ_{fault} epochs, cancel order and re-introduce it to the market
   - if the piece cannot be retrieved and reconstructed from the network, cancel order and re-fund the client

Client

at any time:

1. submit new storage orders via Put.AddOrders
   (a) find matching orders via Put.MatchOrders
   (b) send file to the matched miner $\mathcal{M}$
2. submit new retrieval orders via Get.AddOrders
   (a) find matching orders via Get.MatchOrders
   (b) create a payment channel with $\mathcal{M}$

on receiving $\mathcal{O}_{\text{deal}}$ from Storage Miners $\mathcal{M}$

1. sign $\mathcal{O}_{\text{deal}}$
2. submit the signed $\mathcal{O}_{\text{deal}}$ to the blockchain via Put.AddOrders

on receiving (p_i) from Retrieval Miners $\mathcal{M}$:

1. verify that (p_i) is valid and it was requested
2. send a micropayment to $\mathcal{M}$

Storage Mine

at any time:

1. renew expired pledges via Manage.PledgeSector
2. pledge new storage via Manage.PledgeSector
3. submit a new ask order via Put.AddOrder

at each epoch t:

1. for each $\mathcal{O}_{\text{ask}}$ in the orderbook:
   (a) find matched orders via Put.MatchOrders
   (b) start a new deal by contacting the matching client
2. for each sector pledged:
   (a) generate proof of storage via Manage.ProveSector
   (b) if time to post the proof (every Δ_{proof} epochs), submit it to the blockchain

on receiving piece p from client $\mathcal{C}$:

1. check if the piece is of the size specified in the order $\mathcal{O}_{\text{bid}}$
2. create $\mathcal{O}_{\text{deal}}$ and sign it and send it to $\mathcal{C}$
3. store the piece in a sector
4. if the sector is full, run Manage.SealSector

Retrieval Mine

at any time:

1. gossip *ask* orders to the network
2. listen to *bid* orders from the network

on retrieval request from $\mathcal{C}$:

1. start payment channel with $\mathcal{C}$
2. split data in multiple parts
3. only send parts if payments are received

Figure 1: Sketch of the Filecoin Protocol.

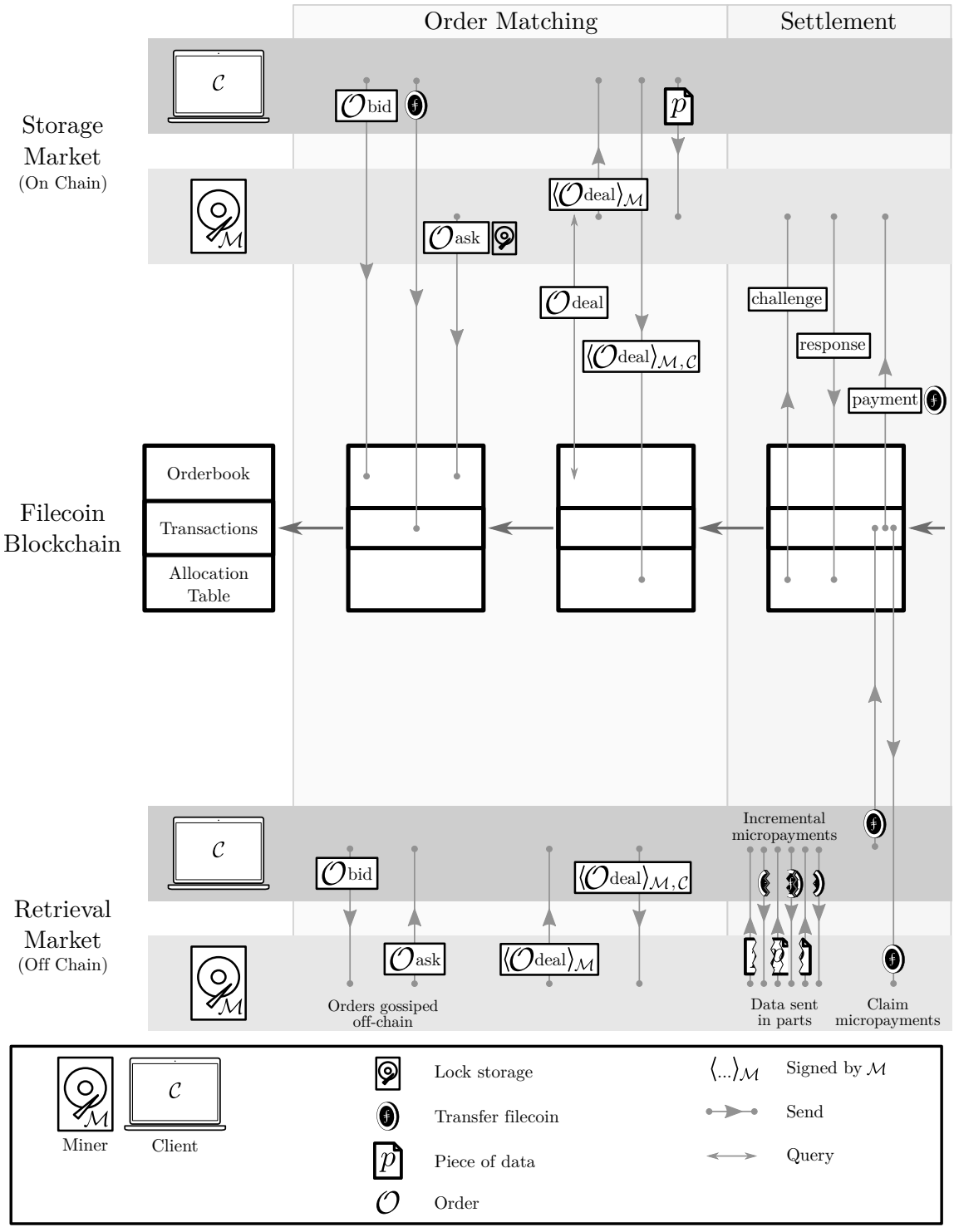

Figure 2: Illustration of the Filecoin Protocol, showing an overview of the Client-Miner interactions. The Storage and Retrieval Markets shown above and below the blockchain, respectively, with time advancing from the Order Matching phase on the left to the Settlement phase on the right. Note that before micropayments can be made for retrieval, the client must lock the funds for the microtransaction.

2 De nition of a Decentralized Storage Network

We introduce the notion of a *Decentralized Storage Network* (DSN) scheme. DSNs aggregate storage offered by multiple independent storage providers and self-coordinate to provide data storage and data retrieval to clients. Coordination is decentralized and does not require trusted parties: the secure operation of theses systems is achieved through protocols that coordinate and verify operations carried out by individual parties. DSNs can employ different strategies for coordination, including Byzantine Agreement, gossip protocols, or CRDTs, depending on the requirements of the system. Later, in Section 4, we provide a construction for the Filecoin DSN.

Definition 2.1. A DSN scheme Π is a tuple of protocols run by storage providers and clients:

$$(\mathsf{Put}, \mathsf{Get}, \mathsf{Manage})$$

- $\mathsf{Put}(\mathsf{data}) \to \mathsf{key}$: Clients execute the Put protocol to *store* data under a unique identifier key.
- $\mathsf{Get}(\mathsf{key}) \to \mathsf{data}$: Clients execute the Get protocol to *retrieve* data that is currently stored using key.
- $\mathsf{Manage}()$: The network of participants coordinates via the Manage protocol to: control the available storage, audit the service offered by providers and repair possible faults. The Manage protocol is run by storage providers often in conjunction with clients or a network of auditors[1].

A DSN scheme Π must guarantee *data integrity* and *retrievability* as well as tolerate *management* and *storage faults* defined in the following sections.

2.1 Fault tolerance

2.1.1 Management faults

We define management faults to be 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 liveness and safety of the system.

For example, consider a DSN scheme Π, where the Manage protocol requires Byzantine Agreement (BA) to audit storage providers. In such protocol, the network receives proofs of storage from storage providers and runs BA to agree on the validity of these proofs. If the BA tolerates up to f faults out of n total nodes, then our DSN can tolerate $f < n/2$ faulty nodes. On violations of these assumptions, audits can be compromised.

2.1.2 Storage faults

We define storage faults to be 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. The parameters f and m depend on protocol implementation; protocol designers can fix f and m or leave the choice to the user, extending $\mathsf{Put}(\mathsf{data})$ into $\mathsf{Put}(\mathsf{data}, f, m)$. A Get execution on stored data is successful if there are fewer than f faulttorage

prNo-35228(some2(pa)-ss.) 22.8722(pa)-(and)-401(57TJ/F8(run)]TJ -378.6ts)-394(03(de)-388(r)5 -1dplleme)-333(4(0

2.2.1 Data Integrity

This property requires that no bounded adversary $\mathcal{A}$ can convince clients to accept altered or falsified data at the end of a Get execution.

Definition 2.2. A DSN scheme Π provides *data integrity* if: for any successful Put execution for some data d under key k, there is no computationally-bounded adversary $\mathcal{A}$ that can convince a client to accept d', for $\mathsf{d}' \neq \mathsf{d}$ at the end of a Get execution for identifier k.

2.2.2 Retrievability

This property captures the requirement that, given our fault-tolerance assumptions of Π, if some data has been successfully stored in Π and storage providers continue to follow the protocol, then clients can eventually retrieve the data.

Definition 2.3. A DSN scheme Π provides *retrievability* if: for any successful Put execution for data under key, there exists a successful Get execution for key for which a client retrieves data.[2].

2.2.3 Other Properties

DSNs can provide other properties specific to their application. We define three key properties required by the Filecoin DSN: *public verifiability*, *auditability*, and *incentive-compatibility*.

Definition 2.4. A DSN scheme Π is *publicly verifiable* if: 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 key and does not have access to data.

Definition 2.5. A DSN scheme Π is *auditable*, if it 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.

Definition 2.6. A DSN scheme Π is *incentive-compatible*, if: 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.

[2]This definition does not guarantee every Get to succeed: if every Get eventually returns data, then the scheme is *fair*.

3 *Proof-of-Replication* and *Proof-of-Spacetime*

In the Filecoin protocol, storage providers must convince their clients that they stored the data they were paid to store; in practice, storage providers will generate *Proofs-of-Storage* (PoS) that the blockchain network (or the clients themselves) verifies.

In this section we motivate, present and outline implementations for the *Proof-of-Replication* (PoRep) and *Proof-of-Spacetime* (PoSt) schemes used in Filecoin.

3.1 Motivation

Proofs-of-Storage (PoS) schemes such as *Provable Data Possession* (PDP) [2] and *Proof-of-Retrievability* (PoR) [3, 4] schemes allow a user (i.e. the verifier $\mathcal{V}$) who outsources data $\mathcal{D}$ to a server (i.e. the prover $\mathcal{P}$) to repeatedly check if the server is still storing $\mathcal{D}$. The user can verify the integrity of the data outsourced to a server in a very efficient way, more efficiently than downloading the data. 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*.

- *Sybil Attacks*: Malicious miners could pretend to store (and get paid for) more copies than the ones physically stored by creating multiple Sybil identities, but storing the data only once.
- *Outsourcing Attacks*: Malicious miners could commit to store more data than the amount they can physically store, relying on quickly fetching data from other storage providers.
- *Generation Attacks*: Malicious miners could claim to be storing a large amount of data which they are instead efficiently generating on-demand using a small program. If the program is smaller than the purportedly stored data, this inflates the malicious miner's likelihood of winning a block reward in Filecoin, which is proportional to the miner's storage currently in use.

3.2 Proof-of-Replication

Proof-of-Replication (PoRep) is a novel *Proof-of-Storage* which allows a server (i.e. the prover $\mathcal{P}$) to convince a user (i.e. the verifier $\mathcal{V}$) that some data $\mathcal{D}$ has been *replicated* to its own uniquely dedicated physical storage. Our scheme is an interactive protocol, where the prover $\mathcal{P}$: (a) commits to store n distinct *replicas* (physically independent copies) of some data $\mathcal{D}$, and then (b) convinces the verifier $\mathcal{V}$, that $\mathcal{P}$ is indeed storing each of the replicas via a challenge/response protocol. To the best of our knowledge, PoRep improves on PoR and PDP schemes, preventing *Sybil Attacks*, *Outsourcing Attacks*, and *Generation Attacks*.

Note. For a formal definition, a description of its properties, and an in-depth study of *Proof-of-Replication*, we refer the reader to [5].

Definition 3.1. (Proof-of-Replication) A PoRep scheme enables an efficient prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that $\mathcal{P}$ is storing a *replica* $\mathcal{R}$, a physical independent copy of some data $\mathcal{D}$, unique to $\mathcal{P}$. A PoRep protocol is characterized by a tuple of polynomial-time algorithms:

$$(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$$

- $\mathsf{PoRep.Setup}(1^\lambda, \mathcal{D}) \to \mathcal{R}, \mathcal{S}_\mathcal{P}, \mathcal{S}_\mathcal{V}$, where $\mathcal{S}_\mathcal{P}$ and $\mathcal{S}_\mathcal{V}$ are scheme-specific setup variables for $\mathcal{P}$ and $\mathcal{V}$, λ is a security parameter. PoRep.Setup is used to generate a replica $\mathcal{R}$, and give $\mathcal{P}$ and $\mathcal{V}$ the necessary information to run PoRep.Prove and PoRep.Verify. Some schemes may require the prover or interaction with a third party to compute PoRep.Setup.

- $\mathsf{PoRep.Prove}(\mathcal{S}_\mathcal{P}, \mathcal{R}, c) \rightarrow \pi^c$, where c is a random challenge issued by a verifier $\mathcal{V}$, and π^c is a proof that a prover has access to $\mathcal{R}$ a specific *replica* of $\mathcal{D}$. PoRep.Prove is run by $\mathcal{P}$ to produce a π^c for $\mathcal{V}$.
- $\mathsf{PoRep.Verify}(\mathcal{S}_\mathcal{V}, c, \pi^c) \rightarrow \{0,1\}$, which checks whether a proof is correct. PoRep.Verify is run by $\mathcal{V}$ and convinces $\mathcal{V}$ whether $\mathcal{P}$ has been storing $\mathcal{R}$.

3.3 Proof-of-Spacetime

Proof-of-Storage schemes allow a user to check if a storage provider is storing the outsourced data at the time of the challenge. *How can we use* PoS *schemes to prove that some data was being stored throughout a period of time?* A natural answer to this question is to require the user to repeatedly (e.g. every minute) send challenges to the storage provider. However, the communication complexity required in each interaction can be the bottleneck in systems such as Filecoin, where storage providers are required to submit their proofs to the blockchain network.

To address this question, we introduce a new proof, *Proof-of-Spacetime*, where 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.

Definition 3.2. (Proof-of-Spacetime) A PoSt scheme enables an efficient prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that $\mathcal{P}$ is storing some data $\mathcal{D}$ for some time t. A PoSt is characterized by a tuple of polynomial-time algorithms:

$$(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$$

- $\mathsf{PoSt.Setup}(1^\lambda, \mathcal{D}) \rightarrow \mathcal{S}_\mathcal{P}, \mathcal{S}_\mathcal{V}$, where $\mathcal{S}_\mathcal{P}$ and $\mathcal{S}_\mathcal{V}$ are scheme-specific setup variables for $\mathcal{P}$ and $\mathcal{V}$, λ is a security parameter. PoSt.Setup is used to give $\mathcal{P}$ and $\mathcal{V}$ the necessary information to run PoSt.Prove and PoSt.Verify. Some schemes may require the prover or interaction with a third party to compute PoSt.Setup.
- $\mathsf{PoSt.Prove}(\mathcal{S}_\mathcal{P}, \mathcal{D}, c, t) \rightarrow \pi^c$, where c is a random challenge issued by a verifier $\mathcal{V}$, and π^c is a proof that a prover has access to $\mathcal{D}$ for some time t. PoSt.Prove is run by $\mathcal{P}$ to produce a π^c for $\mathcal{V}$.
- $\mathsf{PoSt.Verify}(\mathcal{S}_\mathcal{V}, c, t, \pi^c) \rightarrow \{0,1\}$, which checks whether a proof is correct. PoSt.Verify is run by $\mathcal{V}$ and convinces $\mathcal{V}$ whether $\mathcal{P}$ has been storing $\mathcal{D}$ for some time t.

3.4 Practical PoRep and PoSt

We are interested in practical PoRep and PoSt constructions that can be deployed in existing systems and do not rely on trusted parties or hardware. We give a construction for PoRep (see *Seal-based Proof-of-Replication* in [5]) that requires a very slow sequential computation Seal to be performed during Setup to generate a replica. The protocol sketches for PoRep and PoSt are presented in Figure 4 and the underlying mechanism of the proving step in PoSt is illustrated in Figure 3.

3.4.1 Cryptographic building blocks

Collision-resistant hashing. We use a collision resistant hash function $\mathsf{CRH} : \{0,1\}^* \rightarrow \{0,1\}^{O(\lambda)}$. We also use a collision resistant hash function MerkleCRH, which divides a string in multiple parts, construct a binary tree and recursively apply CRH and outputs the root.

zk-SNARKs. Our practical implementations of PoRep and PoSt rely on zero-knowledge Succinct Non-interactive ARguments of Knowledge (zk-SNARKs) [6, 7, 8]. Because zk-SNARKs are *succinct*, proofs are very short and easy to verify. More formally, let L be an NP language and C be a decision circuit for L. A trusted party conducts a one-time setup phase that results in two public keys: a proving key pk and a verification key vk. The proving key pk enables any (untrusted) prover to generate a proof π attesting that

$x \in L$ for an instance x of her choice. The non-interactive proof π is both *zero-knowledge* and *proof-of-knowledge*. Anyone can use the verification key vk to verify the proof π; in particular zk-SNARK proofs are publicly verifiable: anyone can verify π, without interacting with the prover that generated π. The proof π has constant size and can be verified in time that is linear in $|x|$.

A zk-SNARK for circuit satisfiability is a triple of polynomial-time algorithms

$$(\mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$$

- $\mathsf{KeyGen}(1^\lambda, C) \rightarrow (\mathsf{pk}, \mathsf{vk})$. On input security parameter λ and a circuit C, KeyGen probabilistically samples pk and vk. Both keys are published as public parameters and can be used to prove/verify membership in L_C.
- $\mathsf{Prove}(\mathsf{pk}, x, w) \rightarrow \pi$. On input pk and input x and witness for the NP-statement w, the *prover* Prove outputs a non-interactive proof π for the statement $x \in L_C$.
- $\mathsf{Verify}(\mathsf{vk}, x, \pi) \rightarrow \{0, 1\}$. On input vk, an input x, and a proof π, the *verifier* Verify outputs 1 if $x \in L_C$.

We refer the interested reader to [6, 7, 8] for formal presentation and implementation of zk-SNARK systems. Generally these systems require the KeyGen operation to be run by a trusted party; novel work on Scalable Computational Integrity and Privacy (SCIP) systems [9] shows a promising direction to avoid this initial step, hence the above trust assumption.

3.4.2 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 $\mathcal{D}$ 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 $\mathsf{PoRep.Setup}$ to take substantially longer than the time expected for responding to a challenge. For a more formal definition of the Seal operation see [5]. The above operation can be realized with $\mathsf{Seal}^\tau_{\mathsf{AES-256}}$, and τ such that $\mathsf{Seal}^\tau_{\mathsf{AES-256}}$ takes 10-100x longer than the honest challenge-prove-verify sequence. Note that it is important to choose τ such that running $\mathsf{Seal}^\tau_{\mathsf{BC}}$ is distinguishably more expensive than running Prove with random access to $\mathcal{R}$.

3.4.3 Practical PoRep construction

This section describes the construction of the PoRep protocol and includes a simplified protocol sketch in Figure 4; implementation and optimization details are omitted.

Creating a Replica. The Setup algorithm generates a replica via the Seal operation and a proof that it was correctly generated. The prover generates the replica and sends the outputs (excluding $\mathcal{R}$) to the verifier.

Setup
- INPUTS:
  - prover key pair $(\mathsf{pk}_\mathcal{P}, \mathsf{sk}_\mathcal{P})$
  - prover SEAL key $\mathsf{pk}_{\mathsf{SEAL}}$
  - data $\mathcal{D}$
- OUTPUTS: replica $\mathcal{R}$, Merkle root rt of $\mathcal{R}$, proof π_{SEAL}

Proving Storage. The Prove algorithm generates a proof of storage for the replica. The prover receives a random challenge, c, from the verifier, which determines a specific leaf $\mathcal{R}_c$ in the Merkle tree of $\mathcal{R}$ with root rt; the prover generates a proof of knowledge about $\mathcal{R}_c$ and its Merkle path leading up to rt.

Prove
- INPUTS:
  - prover *Proof-of-Storage* key $\mathsf{pk}_{\mathsf{POS}}$
  - replica $\mathcal{R}$
  - random challenge c
- OUTPUTS: a proof π_{POS}

Verifying the Proofs. The Verify algorithm checks the validity of the proofs of storage given the Merkle root of the replica and the hash of the original data. Proofs are publicly verifiable: nodes in the distributed system maintaining the ledger and clients interested in particular data can verify these proofs.

Verify
- INPUTS:
  - prover public key, $\mathsf{pk}_{\mathcal{P}}$
  - verifier SEAL and POS keys $\mathsf{vk}_{\mathsf{SEAL}}$, $\mathsf{vk}_{\mathsf{POS}}$
  - hash of data $\mathcal{D}$, $h_{\mathcal{D}}$
  - Merkle root of replica $\mathcal{R}$, rt
  - random challenge, c
  - tuple of proofs, $(\pi_{\mathsf{SEAL}}, \pi_{\mathsf{POS}})$
- OUTPUTS: bit b, equals 1 if proofs are valid

3.4.4 Practical PoSt construction

This section describes the construction of the PoSt protocol and includes a simplified protocol sketch in Figure 4; implementation and optimization details are omitted. The Setup and Verify algorithm are equivalent to the PoRep construction, hence we describe here only Prove.

Proving space and time. The Prove algorithm generates a *Proof-of-Spacetime* for the replica. The prover receives a random challenge from the verifier and generate *Proofs-of-Replication* in sequence, using the output of a proof as an input of the other for a specified amount of iterations t (see Figure 3).

Prove
- INPUTS:
  - prover PoSt key $\mathsf{pk}_{\mathsf{POST}}$
  - replica $\mathcal{R}$
  - random challenge c
  - time parameter t
- OUTPUTS: a proof π_{POST}

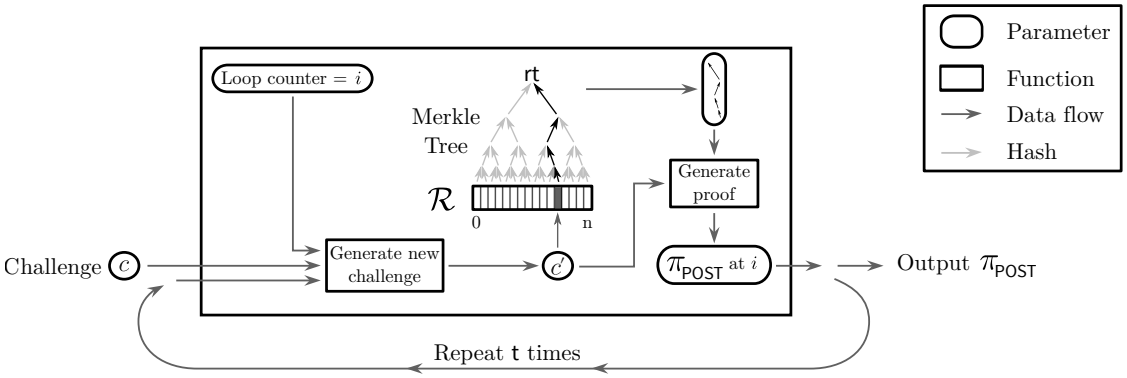

Figure 3: Illustration of the underlying mechanism of PoSt.Prove showing the iterative proof to demonstrate storage over time.

3.5 Usage in Filecoin

The Filecoin protocol employs *Proof-of-Spacetime* to audit the storage offered by miners. To use PoSt in Filecoin, we modify our scheme to be non-interactive since there is no designated verifier, and we want any member of the network to be able to verify. Since our verifier runs in the public-coin model, we can extract randomness from the blockchain to issue challenges.

Filecoin PoRep protocol

Setup

- INPUTS:
  - prover key pair $(\mathsf{pk}_{\mathcal{P}}, \mathsf{sk}_{\mathcal{P}})$
  - prover SEAL key $\mathsf{pk}_{\mathsf{SEAL}}$
  - data $\mathcal{D}$
- OUTPUTS: replica $\mathcal{R}$, Merkle root rt of $\mathcal{R}$, proof π_{SEAL}

1) Compute $h_{\mathcal{D}} := \mathsf{CRH}(\mathcal{D})$
2) Compute $\mathcal{R} := \mathsf{Seal}^{\tau}(\mathcal{D}, \mathsf{sk}_{\mathcal{P}})$
3) Compute $\mathsf{rt} := \mathsf{MerkleCRH}(\mathcal{R})$
4) Set $\vec{x} := (\mathsf{pk}_{\mathcal{P}}, h_{\mathcal{D}}, \mathsf{rt})$
5) Set $\vec{w} := (\mathsf{sk}_{\mathcal{P}}, \mathcal{D})$
6) Compute $\pi_{\mathsf{SEAL}} := \mathsf{SCIP.Prove}(\mathsf{pk}_{\mathsf{SEAL}}, \vec{x}, \vec{w})$
7) Output $\mathcal{R}$, rt, π_{SEAL}

Prove

- INPUTS:
  - prover *Proof-of-Storage* key $\mathsf{pk}_{\mathsf{POS}}$
  - replica $\mathcal{R}$
  - random challenge c
- OUTPUTS: a proof π_{POS}

1) Compute $\mathsf{rt} := \mathsf{MerkleCRH}(\mathcal{R})$
2) Compute $\mathsf{path} :=$ Merkle path from rt to leaf $\mathcal{R}_c$
3) Set $\vec{x} := (\mathsf{rt}, c)$
4) Set $\vec{w} := (\mathsf{path}, \mathcal{R}_c)$
5) Compute $\pi_{\mathsf{POS}} := \mathsf{SCIP.Prove}(\mathsf{pk}_{\mathsf{POS}}, \vec{x}, \vec{w})$
6) Output π_{POS}

Verify

- INPUTS:
  - prover public key, $\mathsf{pk}_{\mathcal{P}}$
  - verifier SEAL and POS keys $\mathsf{vk}_{\mathsf{SEAL}}$, $\mathsf{vk}_{\mathsf{POS}}$
  - hash of data $\mathcal{D}$, $h_{\mathcal{D}}$
  - Merkle root of replica $\mathcal{R}$, rt
  - random challenge, c
  - tuple of proofs, $(\pi_{\mathsf{SEAL}}, \pi_{\mathsf{POS}})$
- OUTPUTS: bit b, equals 1 if proofs are valid

1) Set $\vec{x_1} := (\mathsf{pk}_{\mathcal{P}}, h_{\mathcal{D}}, \mathsf{rt})$
2) Compute $b_1 := \mathsf{SCIP.Verify}(\mathsf{vk}_{\mathsf{SEAL}}, \vec{x_1}, \pi_{\mathsf{SEAL}})$
3) Set $\vec{x_2} := (\mathsf{rt}, c)$
4) Compute $b_2 := \mathsf{SCIP.Verify}(\mathsf{vk}_{\mathsf{POS}}, \vec{x_2}, \pi_{\mathsf{POS}})$
5) Output $b_1 \wedge b_2$

Filecoin PoSt protocol

Setup

- INPUTS:
  - prover key pair $(\mathsf{pk}_{\mathcal{P}}, \mathsf{sk}_{\mathcal{P}})$
  - prover POST key pair $\mathsf{pk}_{\mathsf{POST}}$
  - some data $\mathcal{D}$
- OUTPUTS: replica $\mathcal{R}$, Merkle root rt of $\mathcal{R}$, proof π_{SEAL}

1) Compute $\mathcal{R}$, rt, $\pi_{\mathsf{SEAL}} := \mathsf{PoRep.Setup}(\mathsf{pk}_{\mathcal{P}}, \mathsf{sk}_{\mathcal{P}}, \mathsf{pk}_{\mathsf{SEAL}}, \mathcal{D})$
2) Output $\mathcal{R}$, rt, π_{SEAL}

Prove

- INPUTS:
  - prover PoSt key $\mathsf{pk}_{\mathsf{POST}}$
  - replica $\mathcal{R}$
  - random challenge c
  - time parameter t
- OUTPUTS: a proof π_{POST}

1) Set $\pi_{\mathsf{POST}} := \perp$
2) Compute $\mathsf{rt} := \mathsf{MerkleCRH}(\mathcal{R})$
3) For $i = 0...t$:
   a) Set $c' := \mathsf{CRH}(\pi_{\mathsf{POST}} || c || i)$
   b) Compute $\pi_{\mathsf{POS}} := \mathsf{PoRep.Prove}(\mathsf{pk}_{\mathsf{POS}}, \mathcal{R}, c')$
   c) Set $\vec{x} := (\mathsf{rt}, c, i)$
   d) Set $\vec{w} := (\pi_{\mathsf{POS}}, \pi_{\mathsf{POST}})$
   e) Compute $\pi_{\mathsf{POST}} := \mathsf{SCIP.Prove}(\mathsf{pk}_{\mathsf{POST}}, \vec{x}, \vec{w})$
4) Output π_{POST}

Verify

- INPUTS:
  - prover *public key* $\mathsf{pk}_{\mathcal{P}}$
  - verifier SEAL and POST keys $\mathsf{vk}_{\mathsf{SEAL}}$, $\mathsf{vk}_{\mathsf{POST}}$
  - hash of some data $h_{\mathcal{D}}$
  - Merkle root of some replica rt
  - random challenge c
  - time parameter t
  - tuple of proofs $(\pi_{\mathsf{SEAL}}, \pi_{\mathsf{POST}})$
- OUTPUTS: bit b, equals 1 if proofs are valid

1) Set $\vec{x_1} := (\mathsf{pk}_{\mathcal{P}}, h_{\mathcal{D}}, \mathsf{rt})$
2) Compute $b_1 := \mathsf{SCIP.Verify}(\mathsf{vk}_{\mathsf{SEAL}}, \vec{x_1}, \pi_{\mathsf{SEAL}})$
3) Set $\vec{x_2} := (\mathsf{rt}, c, t)$
4) Compute $b_2 := \mathsf{SCIP.Verify}(\mathsf{vk}_{\mathsf{POST}}, \vec{x_2}, \pi_{\mathsf{POST}})$
5) Output $b_1 \wedge b_2$

Figure 4: *Proof-of-Replication* and *Proof-of-Spacetime* protocol sketches. Here CRH denotes a collision-resistant hash, $\vec{x}$ is the NP-statement to be proven, and $\vec{w}$ is the witness.

4 Filecoin: a DSN Construction

The Filecoin DSN is a decentralized storage network that is *auditable*, *publicly verifiable* and designed on *incentives*. Clients pay a network of miners for data storage and retrieval; miners offer disk space and bandwidth in exchange of payments. Miners receive their payments only if the network can *audit* that their service was correctly provided.

In this section, we present the Filecoin DSN construction, based on the DSN definition and *Proof-of-Spacetime*.

4.1 Setting

4.1.1 Participants

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 loose 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.

4.1.2 The Network, $\mathcal{N}$

We personify all the users that run Filecoin full nodes as one single abstract entity: *The Network*. The Network acts as an intermediary that runs the Manage protocol; informally, at every new block in the Filecoin blockchain, full nodes manage the available storage, validate pledges, audit the storage proofs, and repair possible faults.

4.1.3 The Ledger

Our protocol is applied on top of a ledger-based currency; for generality we refer to this as the *Ledger*, $\mathcal{L}$. At any given time t (referred to as *epoch*), all users have access to $\mathcal{L}_t$, the ledger at epoch t, which is a sequence of *transactions*. The ledger is append-only[3]. The Filecoin DSN protocol can be implemented on any ledger that allows for the verification of Filecoin's proofs; we show how we can construct a ledger based on *useful work* in Section 6.

4.1.4 The Markets

Demand and supply of storage meet at the two Filecoin Markets: Storage Market and Retrieval Market. The markets are two decentralized exchanges and are explained in detail in Section 5. In brief, clients and miners set the prices for the services they request or provide by submitting orders to the respective markets. The exchanges provide a way for clients and miners to see matching offers and initiate deals. By running the Manage protocol, the Network guarantees that miners are rewarded and clients are charged if the service requested has been successfully provided.

[3] $t < t'$ implies that $\mathcal{L}_t$ is a prefix of $\mathcal{L}'_t$

4.2 Data Structures

Pieces. A *piece* is some part of data that a client is storing in the DSN. For example, data can be deliberately divided into many pieces and each piece can be stored by a different set of Storage Miners.

Sectors. A *sector* is some disk space that a Storage Miner provides to the network. Miners store pieces from clients in their sectors and earn tokens for their services. In order to store pieces, Storage Miners must pledge their sectors to the network.

AllocationTable. The AllocTable is a data structure that keeps track of pieces and their assigned sectors. The AllocTable is updated at every block in the ledger and its Merkle root is stored in the latest block. In practice, the table is used to keep the state of the DSN, allowing for quick look-ups during proof verification. For more details, see Figure 5.

Orders. An *order* is a statement of intent to request or offer a service. Clients submit *bid* orders to the markets to request a service (resp. Storage Market for storing data and Retrieval Market for retrieving data) and Miners submit *ask* orders to offer a service. The order data structures are shown in Figure 10. The Market Protocols are detailed in Section 5.

Orderbook. Orderbooks are sets of orders. See the Storage Market orderbook in Section 5.2.2 and Retrieval Market orderbook in Section 5.3.2 for details.

Pledge. A *pledge* is a commitment to offer storage (specifically a *sector*) to the network. Storage Miners must submit their pledge to the ledger in order to start accepting orders in the Storage Market. A pledge consists of the size of the pledged sector and the collateral deposited by the Storage Miner (see Figure 5 for more details).

Data Structures

Pledge

pledge := $\langle \mathsf{size}, \mathsf{coll} \rangle_{\mathcal{M}_i}$

- size, the size of the sector being pledged.
- coll, the collateral specific to this pledge that $\mathcal{M}_i$ deposits.

Orderbook

OrderBook: $(\mathcal{O}^1..\mathcal{O}^n)$

- $\mathcal{O}^i$, currently valid *deal*, *ask*, *bid* orders.

Allocation

allocTable: $\{\mathcal{M}_1 \rightarrow (\mathsf{allocEntry}..\mathsf{allocEntry}), \mathcal{M}_2..\}$

allocEntry: (sid, orders, last, missing)

- sid, sector id
- $\mathcal{O}^i$, currently valid *deal*, *ask*, *bid* orders.
- orders, set of orders $\{\mathcal{O}_{\mathsf{deal}}..\mathcal{O}_{\mathsf{deal}}\}$
- last, last proof of storage in the ledger $\mathcal{L}$
- missing, counter for missing proofs

Figure 5: Data Structures in a DSN scheme

4.3 Protocol

In this Section, we give an overview of the Filecoin DSN by describing the operations performed by the *clients*, the *Network* and the *miners*. We present the methods of the Get and the Put protocol in Figure 7 and the Manage protocol in Figure 8. An example protocol execution is shown in Figure 6. The overall Filecoin Protocol is presented in Figure 1.

4.3.1 Client Cycle

1. **Put**: *Client stores data in Filecoin.*

   Clients can store their data by paying Storage Miners in Filecoin tokens. The Put protocol is described in detail in Section 5.2.

   A client initiates the Put protocol by submitting a *bid* order to the Storage Market orderbook (by submitting their order to the blockchain). When a matching *ask* order from miners is found, the client sends the piece to the miner. Both parties sign a *deal* order and submit it to the Storage Market orderbook.

   Clients should be able to decide the amount of physical replicas of their pieces either by submitting multiple orders (or specifying a replication factor in the order). Higher redundancy results in a higher tolerance of storage faults.

2. **Get**: *Client retrieves data from Filecoin.*

   Clients can retrieve any data stored in the DSN by paying Retrieval Miners in Filecoin tokens. The Get protocol is described in detail in Section 5.3.

   A client initiates the Get protocol by submitting a *bid* order to the Retrieval Market orderbook (by gossiping their order to the network). When a matching *ask* order from miners is found, the client receives the piece from the miner. When received, both parties sign a *deal* order and submit it to the blockchain to confirm that the exchange succeeded.

4.3.2 Mining Cycle (for Storage Miners)

We give an informal overview of the mining cycle.

1. **Pledge**: *Storage Miners pledge to provide storage to the Network.*

   Storage Miners pledge their storage to the network by depositing collateral via a *pledge* transaction in the blockchain, via Manage.PledgeSector. The collateral is deposited for the time intended to provide the service, and it is returned if the miner generates proofs of storage for the data they commit to store. If some proofs of storage fail, a proportional amount of collateral is lost.

   Once the pledge transaction appears in the blockchain, miners can offer their storage in the Storage Market: they set their price and add an ask order to the market's orderbook.

   Manage.PledgeSector
   - INPUTS:
     - current allocation table allocTable
     - pledge request pledge
   - OUTPUTS: allocTable′

2. **Receive Orders**: *Storage Miners get storage requests from the Storage Market.*

   Once the pledge transaction appears in the blockchain (hence in the AllocTable), miners can offer their storage in the Storage Market: they set their price and add an ask order to the market's orderbook via Put.AddOrders.

   Put.AddOrders
   - INPUTS: list of orders $\mathcal{O}^1..\mathcal{O}^n$
   - OUTPUTS: bit b, equals 1 if successful

   Check if their orders are matched with a corresponding *bid* order from a client, via Put.MatchOrders.

   Put.MatchOrders
   - INPUTS:
     - the current Storage Market OrderBook
     - query order to match $\mathcal{O}^q$
   - OUTPUTS: matching orders $\mathcal{O}^1..\mathcal{O}^n$

Once orders are matched, clients send their data to the Storage Miners. When receiving the piece, miners run Put.ReceivePiece. When the data is received, both the miner and the client sign a *deal* order and submit it to the blockchain.

Put.ReceivePiece
- INPUTS:
  - signing key for $\mathcal{M}_j$.
  - current orderbook OrderBook
  - *ask* order $\mathcal{O}_{\mathsf{ask}}$
  - *bid* order $\mathcal{O}_{\mathsf{bid}}$
  - piece p
- OUTPUTS: *deal* order $\mathcal{O}_{\mathsf{deal}}$ signed by $\mathcal{C}_i$ and $\mathcal{M}_j$

3. **Seal**: *Storage Miners prepare the pieces for future proofs.*

   Storage Miners' storage is divided in sectors, each sector contains pieces assigned to the miner. The Network keeps track of each Storage Miners' sector via the allocation table. When a Storage Miner sector is filled, the sector is *sealed*. Sealing is a slow, sequential operation that transforms the data in a sector into a replica, a unique physical copy of the data that is associated to the public key of the Storage Miner. Sealing is a necessary operation during the *Proof-of-Replication* as described in Section 3.4.

   Manage.SealSector
   - INPUTS:
     - miner public/private key pair $\mathcal{M}$
     - sector index j
     - allocation table allocTable
   - OUTPUTS: a proof π_{SEAL}, a root hash rt

4. **Prove**: *Storage Miners prove they are storing the committed pieces.*

   When Storage Miners are assigned data, they must repeatedly generate proofs of replication to guarantee they are storing the data (for more details, see Section 3). Proofs are posted on the blockchain and the Network verifies them.

   Manage.ProveSector
   - INPUTS:
     - miner public/private key pair $\mathcal{M}$
     - sector index j
     - challenge c
   - OUTPUTS: a proof π_{POS}

4.3.3 Mining Cycle (for Retrieval Miners)

We give an informal overview of the mining cycle for Retrieval Miners.

1. **Receive Orders**: *Retrieval Miners get data requests from the Retrieval Market.*

   Retrieval Miners announce their pieces by gossiping their *ask* orders to the network: they set their price and add an ask order to the market's orderbook.

   Get.AddOrders
   - INPUTS: list of orders $\mathcal{O}^1..\mathcal{O}^n$
   - OUTPUTS: none

   Then, Retrieval Miners check if their orders are matched with a corresponding *bid* order from a client.

Get.MatchOrders
- INPUTS:
  - the current Retrieval Market OrderBook
  - query order to match $\mathcal{O}^q$
- OUTPUTS: matching orders $\mathcal{O}^1..\mathcal{O}^n$

2. **Send**: *Retrieval Miners send pieces to the client.*

   Once orders are matched, Retrieval Miners send the piece to the client (see Section 5.3 for details). When the piece is received, both the miner and the client sign a *deal* order and submit it to the blockchain.

Put.SendPiece
- INPUTS:
  - an *ask* order $\mathcal{O}_{\mathsf{ask}}$
  - a *bid* order $\mathcal{O}_{\mathsf{bid}}$
  - a piece p
- OUTPUTS: a *deal* order $\mathcal{O}_{\mathsf{deal}}$ signed by $\mathcal{M}_i$

4.3.4 Network Cycle

We give an informal overview of the operations run by the network.

1. **Assign**: *The Network assigns clients' pieces to Storage Miners' sectors.*

   Clients initiate the Put protocol by submitting a bid order in the Storage Market[4].

   When ask and bid orders match, the involved parties jointly commit to the exchange and submit a *deal* order in the market. At this point, the Network *assigns* the data to the miner and makes a note of it in the allocation table.

Manage.AssignOrders
- INPUTS:
  - deal orders $\mathcal{O}^1_{\mathsf{deal}}..\mathcal{O}^n_{\mathsf{deal}}$
  - allocation table allocTable
- OUTPUTS: updated allocation table allocTable′

2. **Repair**: *The Network finds faults and attempt to repair them.*

   All the storage allocations are public to every participant in the network. At every block, the Network checks if the required proofs for each assignment are present, checks that they are valid, and acts accordingly:

   - if any proof is missing or invalid, the network penalizes the Storage Miners by taking part of their collateral,
   - if a large amount of proofs are missing or invalid (defined by a system parameter Δ_{fault}), the network considers the Storage Miner *faulty*, settles the order as failed and reintroduces a new order for the same piece into the the market,
   - if every Storage Miner storing this piece is faulty, then the piece is lost and the client gets refunded.

[4]Storage orders are submitted via the blockchain, see Section 5.

Manage.RepairOrders
- INPUTS:
  - current time t
  - current ledger $\mathcal{L}$
  - table of storage allocations allocTable
- OUTPUTS: orders to repair $\mathcal{O}^1_{\mathsf{deal}}..\mathcal{O}^n_{\mathsf{deal}}$, updated allocation table allocTable

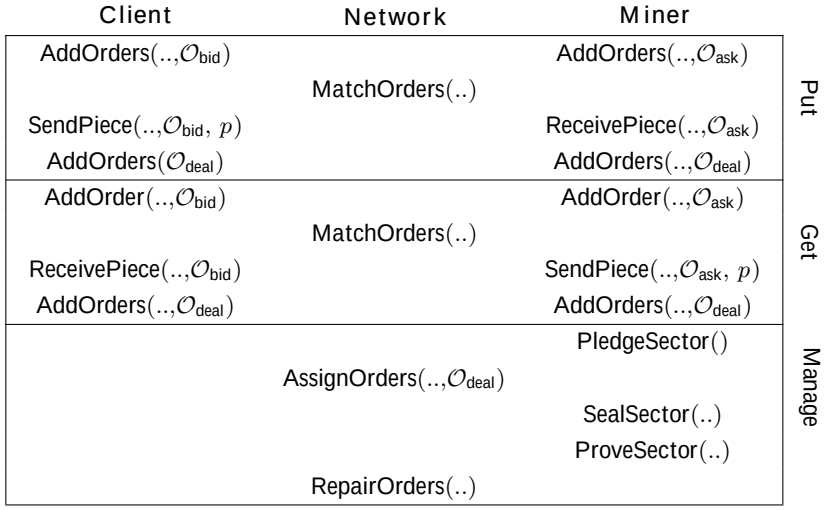

Figure 6: Example execution of the Filecoin DSN, grouped by party and sorted chronologically by row

4.4 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 (π_{SEAL}, π_{POST}) 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.

Put Protocol

Market

AddOrders

- INPUTS: list of orders $\mathcal{O}^1..\mathcal{O}^n$
- OUTPUTS: bit b, equals 1 if successful

1) Set $\mathsf{tx}_{\mathsf{order}} := (\mathcal{O}^1, .., \mathcal{O}^n)$
2) Submit $\mathsf{tx}_{\mathsf{order}}$ to $\mathcal{L}$
3) Wait for $\mathsf{tx}_{\mathsf{order}}$ to be included in $\mathcal{L}$
4) Output 1 on success, 0 otherwise

MatchOrders

- INPUTS:
  - the current Storage Market OrderBook
  - query order to match $\mathcal{O}^q$
- OUTPUTS: matching orders $\mathcal{O}^1..\mathcal{O}^n$

1) Match each $\mathcal{O}^i$ in OrderBook such that:
   a) If $\mathcal{O}^q$ is an *ask* order:
      i) Check if $\mathcal{O}^i$ is *bid* order
      ii) Check $\mathcal{O}^i$.price $\geq \mathcal{O}^q$.price
      iii) Check $\mathcal{O}^i$.size $\leq \mathcal{O}^q$.space
   b) If $\mathcal{O}^q$ is a *bid* order:
      i) Check if $\mathcal{O}^i$ is *ask* order
      ii) Check $\mathcal{O}^i$.price $\leq \mathcal{O}^q$.price
      iii) Check $\mathcal{O}^i$.space $\geq \mathcal{O}^q$.size
2) Output matched orders $\mathcal{O}^1...\mathcal{O}^n$

Exchange

SendPiece

- INPUTS:
  - an *ask* order $\mathcal{O}_{\mathsf{ask}}$
  - a *bid* order $\mathcal{O}_{\mathsf{bid}}$
  - a piece p
- OUTPUTS: a *deal* order $\mathcal{O}_{\mathsf{deal}}$ signed by $\mathcal{M}_i$

1) Get identity of $\mathcal{M}_i$ from $\mathcal{O}_{\mathsf{ask}}$ signature
2) Send ($\mathcal{O}_{\mathsf{ask}}$,$\mathcal{O}_{\mathsf{bid}}$,$p$) to $\mathcal{M}_i$
3) Receive $\mathcal{O}_{\mathsf{deal}}$ signed by $\mathcal{M}_i$
4) Check if $\mathcal{O}_{\mathsf{deal}}$ is valid according to Definition 5.2
5) Output $\mathcal{O}_{\mathsf{deal}}$

ReceivePiece

- INPUTS:
  - signing key for $\mathcal{M}_j$.
  - current orderbook OrderBook
  - *ask* order $\mathcal{O}_{\mathsf{ask}}$
  - *bid* order $\mathcal{O}_{\mathsf{bid}}$
  - piece p
- OUTPUTS: *deal* order $\mathcal{O}_{\mathsf{deal}}$ signed by $\mathcal{C}_i$ and $\mathcal{M}_j$

1) Check if $\mathcal{O}_{\mathsf{bid}}$ is valid:
   a) Check if $\mathcal{O}_{\mathsf{bid}}$ is in OrderBook
   b) Check if $\mathcal{O}_{\mathsf{bid}}$ is not referenced by other active $\mathcal{O}_{\mathsf{deal}}$
   c) Check if $\mathcal{O}_{\mathsf{bid}}$.size is equal to $|p|$
   d) Check if $\mathcal{O}$ is signed by $\mathcal{M}_i$
2) Store p locally
3) Set $\mathcal{O}_{\mathsf{deal}}$:= $\langle\ \mathcal{O}_{\mathsf{ask}}, \mathcal{O}_{\mathsf{deal}}, \mathcal{H}(p)\ \rangle_{\mathcal{M}_i}$
4) Get identity of $\mathcal{C}_j$ from $\mathcal{O}_{\mathsf{bid}}$
5) Send $\mathcal{O}_{\mathsf{deal}}$ to $\mathcal{C}_j$
6) Output $\mathcal{O}_{\mathsf{deal}}$

Get Protocol

Market

AddOrders

- INPUTS: list of orders $\mathcal{O}^1..\mathcal{O}^n$
- OUTPUTS: none

1) Gossip $\mathcal{O}^1..\mathcal{O}^n$ to the network

MatchOrders

- INPUTS:
  - the current Retrieval Market OrderBook
  - query order to match $\mathcal{O}^q$
- OUTPUTS: matching orders $\mathcal{O}^1..\mathcal{O}^n$

1) Match each $\mathcal{O}^i$ in OrderBook such that:
   a) Check $\mathcal{O}^i$.piece is equal to $\mathcal{O}^q$.piece
   b) If $\mathcal{O}^q$ is an *ask* order:
      i) Check if $\mathcal{O}^i$ is *bid* order
      ii) Check $\mathcal{O}^i$.price $\geq \mathcal{O}^q$.price
   c) If $\mathcal{O}^q$ is a *bid* order:
      i) Check if $\mathcal{O}^i$ is *ask* order
      ii) Check $\mathcal{O}^i$.price $\leq \mathcal{O}^q$.price
2) Output matched orders $\mathcal{O}^1...\mathcal{O}^n$

Exchange

SendPiece

- INPUTS:
  - an *ask* order $\mathcal{O}_{\mathsf{ask}}$
  - a *bid* order $\mathcal{O}_{\mathsf{bid}}$
  - a piece p
- OUTPUTS: a *deal* order $\mathcal{O}_{\mathsf{deal}}$ signed by $\mathcal{C}_i$

1) Create $\mathcal{O}_{\mathsf{deal}}$:
   a) Set $\mathcal{O}_{\mathsf{deal}}$.ask := $\mathcal{O}_{\mathsf{ask}}$
   b) Set $\mathcal{O}_{\mathsf{deal}}$.bid := $\mathcal{O}_{\mathsf{deal}}$
2) Get identity of $\mathcal{C}_i$ from $\mathcal{O}_{\mathsf{bid}}$ signature
3) Setup a micropayment channel with $\mathcal{C}_i$
4) For each block of data p_j of p:
   a) Set π_j to be a merkle path from $\mathcal{H}(p)$ to p_j
   b) Send ($\mathcal{O}_{\mathsf{deal}}$,$p_j$, π_j) to $\mathcal{C}_i$
   c) Receive $\langle\ \mathcal{O}_{\mathsf{deal}}, \mathrm{j}\ \rangle_{\mathcal{C}_i}$
5) Output $\mathcal{O}_{\mathsf{deal}}$

ReceivePiece

- INPUTS:
  - a client's key $\mathcal{C}_j$
  - an *ask* order $\mathcal{O}_{\mathsf{ask}}$
  - a *bid* order $\mathcal{O}_{\mathsf{bid}}$
  - merkle tree hash of p in the orders h_p
- OUTPUTS: a piece p

1) Create $\mathcal{O}_{\mathsf{deal}}$:
   a) Set $\mathcal{O}_{\mathsf{deal}}$.ask := $\mathcal{O}_{\mathsf{ask}}$
   b) Set $\mathcal{O}_{\mathsf{deal}}$.bid := $\mathcal{O}_{\mathsf{bid}}$
2) Get identity of $\mathcal{M}_i$ from $\mathcal{O}_{\mathsf{ask}}$ signature
3) Set up a micropayment channel with $\mathcal{M}_i$ (or re-using an existing one)
4) When receiving ($\mathcal{O}_{\mathsf{deal}}$, p_j, π_j) from $\mathcal{M}_i$:
   a) Check if $\mathcal{O}_{\mathsf{deal}}$ is valid and matches $\mathcal{O}_{\mathsf{ask}}$ and $\mathcal{O}_{\mathsf{bid}}$
   b) Check if π_j is a valid merkle-path with root hash h_p
   c) Send $\langle\ \mathcal{O}_{\mathsf{deal}}, j\ \rangle_{\mathcal{C}_i}$
5) Output p

Figure 7: Description of the Put and Get Protocols in the Filecoin DSN

Manage **Protocol**

Network

AssignOrders

- INPUTS:
  - deal orders $\mathcal{O}^1_{\mathsf{deal}}..\mathcal{O}^n_{\mathsf{deal}}$
  - allocation table allocTable
- OUTPUTS: updated allocation table allocTable′

1) Copy allocTable in allocTable′
2) For each order $\mathcal{O}^i_{\mathsf{deal}}$:
   a) Check if $\mathcal{O}^i_{\mathsf{deal}}$ is valid according to Definition 5.2
   b) Get $\mathcal{M}_j$ from $\mathcal{O}^i_{\mathsf{deal}}$ signature
   c) Add details from $\mathcal{O}^i_{\mathsf{deal}}$ to allocTable′
3) Output allocTable′

RepairOrders

- INPUTS:
  - current time t
  - current ledger $\mathcal{L}$
  - table of storage allocations allocTable
- OUTPUTS: orders to repair $\mathcal{O}^1_{\mathsf{deal}}..\mathcal{O}^n_{\mathsf{deal}}$, updated allocation table allocTable

1) For each allocEntry in allocTable:
   a) If $t <$ allocEntry.last $+ \Delta_{\mathsf{proof}}$: skip
   b) Update allocEntry.last$= t$
   c) Check if π is in $\mathcal{L}_{t-\Delta_{\mathsf{proof}}:t}$ and PoSt.Verify(π)
   d) On success: update allocEntry.missing$= 0$
   e) On failure:
      i) update allocEntry.missing++
      ii) penalize collateral from $\mathcal{M}_i$'s pledge
   f) If allocEntry.missing $> \Delta_{\mathsf{fault}}$ then set all the orders from the current sector as failed orders
2) Output failed orders $\mathcal{O}^1_{\mathsf{deal}}..\mathcal{O}^n_{\mathsf{deal}}$ and allocTable′.

Miner

PledgeSector

- INPUTS:
  - current allocation table allocTable
  - pledge request pledge
- OUTPUTS: allocTable′

1) Copy allocTable to allocTable′
2) Set $\mathsf{tx}_{\mathsf{pledge}} :=$ (pledge)
3) Submit $\mathsf{tx}_{\mathsf{pledge}}$ to $\mathcal{L}$
4) Wait for $\mathsf{tx}_{\mathsf{pledge}}$ to be included in $\mathcal{L}$
5) Add new sector of size pledge.size in allocTable′
6) Output allocTable′

SealSector

- INPUTS:
  - miner public/private key pair $\mathcal{M}$
  - sector index j
  - allocation table allocTable
- OUTPUTS: a proof π_{SEAL}, a root hash rt

1) Find all the pieces $p_1..p_n$ in sector $\mathcal{S}_j$ in the allocTable
2) Set $\mathcal{D} := p_1|p_2|..|p_n$
3) Compute $(\mathcal{R}, \mathsf{rt}, \pi_{\mathsf{SEAL}}) :=$ PoSt.Setup$(\mathcal{M}, \mathsf{pk}_{\mathsf{SEAL}}, \mathcal{D})$
4) Output π_{SEAL}, rt

ProveSector

- INPUTS:
  - miner public/private key pair $\mathcal{M}$
  - sector index j
  - challenge c
- OUTPUTS: a proof π_{POS}

1) Find $\mathcal{R}$ for sector j
2) Compute $\pi_{\mathsf{POST}} :=$ PoSt.Prove$(\mathsf{pk}_{\mathsf{POST}}, \mathcal{R}, c, \Delta_{\mathsf{proof}})$
3) Output π_{POST}

Figure 8: Description of the Manage Protocol in the Filecoin DSN

5 Filecoin Storage and Retrieval Markets

Filecoin has two markets: the *Storage Market* and the *Retrieval Market*. The two markets have the same structure but different design. The Storage Market allows Clients to pay *Storage Miners* to store data. The Retrieval Market allows Clients to retrieve data by paying *Retrieval Miners* to deliver the data. In both cases, clients and miners can set their offer and demand prices or accept current offers. The exchanges are run by the *Network* - a personification of the network of full nodes in Filecoin. The network guarantees that miners are rewarded by the clients when providing the service.

5.1 Veri able Markets

Exchange Markets are protocols that facilitate exchange of a specific good or service. They do this by enabling buyers and sellers to conduct transactions. For our purposes, we require exchanges to be *verifiable*: a decentralized network of participants must be able to verify the exchange between buyers and sellers.

We present the notion of *Verifiable Markets*, where no single entity governs an exchange, transactions are transparent, and anybody can participate pseudonymously. Verifiable Market protocols operate the exchange of goods/services in a decentralized fashion: consistency of the orderbooks, orders settlements and correct execution of services are independently verified via the participants - miners and full nodes in the case of Filecoin. We simplify verifiable markets to have the following construction:

Definition 5.1. A *verifiable* Market is a protocol with two phases: order matching and settlement. *Orders* are statements of intent to buy or sell a security, good or service and the *orderbook* is the list of all the available orders.

Veri able Market Protocol

Order matching:

1. Participants add *buy* orders and *sell* orders to the orderbook.
2. 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:

3. 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.
4. On success, the network processes the payments and clears the orders from the orderbook.

Figure 9: Generic protocol for *Verifiable Markets*

5.2 Storage Market

The Storage Market is a *verifiable market* which allows clients (i.e. buyers) to request storage for their data and Storage Miners (i.e. sellers) to offer their storage.

5.2.1 Requirements

We design the Storage Market protocol accordingly to the following 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 informed decision on their orders, (2) client orders must be always 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 clear to the Filecoin blockchain in order to be added to the orderbook.

- **Participants committing 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 to the Storage Market, Storage Miners must pledge, depositing a collateral proportional to their amount of storage in DSN (see Section 4.3.3 for more details). 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 rules outlined in the Repair portion of Subsection 4.3.4.

5.2.2 Datastructures

Put **Orders**. There are three types of orders: *bid* orders, *ask* orders and *deal* orders. Storage Miners create ask orders to add storage, clients create bid orders to request storage, when both parties agree on a price, they jointly create a deal order. The data structures of the orders are shown in detail in Figure 10, and the parameters of the orders are explicitly defined.

Put **Orderbook**. The Orderbook in the Storage Market is the set of currently valid and open *ask*, *bid* and *deal* orders. Users can interact with the orderbook via the methods defined in the Put protocol: AddOrders, MatchOrders as described in Figure 7.

The orderbook is public and every honest user has the same view of the orderbook. At every epoch, new orders are added to the orderbook if new order transactions ($\mathsf{tx}_{\mathsf{order}}$) appear in new blockchain blocks; orders are removed if they are cancelled, expired or settled. Orders are added in blockchain blocks, hence in the orderbook, if they are valid:

Definition 5.2. We define the validity of *bid*, *ask*, *deal* orders:

(Valid *bid* order): A *bid* order from client $\mathcal{C}_i$, $\mathcal{O}_{\mathsf{bid}} := \langle \mathsf{size}, \mathsf{funds}[, \mathsf{price}, \mathsf{time}, \mathsf{coll}, \mathsf{coding}] \rangle_{\mathcal{C}_i}$ is valid if:

- $\mathcal{C}_i$ has at least the amount of funds available in their account.
- time is not set in the past
- The order must guarantee at least a minimum amount[5] of epochs of storage.

(Valid *ask* order): An *ask* order from Storage Miner $\mathcal{M}_i$, $\mathcal{O}_{\mathsf{ask}} := \langle \mathsf{space}, \mathsf{price} \rangle_{\mathcal{M}_i}$ is valid if:

- $\mathcal{M}_i$ has pledged to be a miner and the pledge will not expire before time epochs.
- space must be less than $\mathcal{M}_i$'s available storage: $\mathcal{M}_i$ pledged storage minus the storage committed in the orderbook (in *ask* and *deal* orders).

(Valid *deal* order): A *deal* order $\mathcal{O}_{\mathsf{deal}} := \langle \mathsf{ask}, \mathsf{bid}, \mathsf{ts} \rangle_{\mathcal{C}_i, \mathcal{M}_j}$ is valid if

- ask references an order $\mathcal{O}_{\mathsf{ask}}$ such that: it is in the Storage Market OrderBook, no other deal orders in the Storage Market OrderBook mention it, it is signed by $\mathcal{C}_i$.
- bid references an order $\mathcal{O}_{\mathsf{bid}}$ such that: it is in the Storage Market OrderBook, no other deal orders in the Storage Market OrderBook mention it, it is signed by $\mathcal{M}_j$.
- ts is not set in the future or too far in the past.

Remark. If a malicious client receives a signed deal from a Storage Miner, but never adds it to the orderbook, then the Storage Miner cannot re-use the storage committed in the deal. The field ts prevents this attack because, after ts, the order becomes invalid and cannot be submitted in the orderbook.

[5] This will be a parameter of the system.

Storage Market Orders

***bid* order**

$\mathcal{O}_{\mathsf{bid}} := \langle \mathsf{size}, \mathsf{funds}[, \mathsf{price}, \mathsf{time}, \mathsf{coll}, \mathsf{coding}] \rangle_{\mathcal{C}_i}$

- size, the size of the piece to be stored
- funds, the total amount that client $\mathcal{C}_i$ is depositing
- time, the maximum epoch time for which the file should be stored[a]
- price, the spacetime price in Filecoin[b]
- coll, the collateral specific to this piece that the miner is required to deposit
- coding, the erasure coding scheme for this piece

***ask* order**

$\mathcal{O}_{\mathsf{ask}}$: $\langle$ space, price $\rangle_{\mathcal{M}_i}$

- space, amount of space Storage Miner $\mathcal{M}_i$ is providing in the order
- price, the spacetime price in Filecoin

***deal* order**

$\mathcal{O}_{\mathsf{deal}}$: $\langle$ ask, bid, ts, hash $\rangle_{\mathcal{C}_i, \mathcal{M}_j}$

- ask, a cryptographic reference to $\mathcal{O}_{\mathsf{ask}}$ from $\mathcal{C}_i$
- order, a cryptographic reference to $\mathcal{O}_{\mathsf{bid}}$ from $\mathcal{M}_i$
- ts, timestamp epoch in which the order has been signed by $\mathcal{M}_i$
- hash cryptographic hash of the piece that $\mathcal{M}_j$ will store

[a] If not specified, the piece will be stored until expiration of funds.

[b] If not specified, when a Storage Miner is faulty, the network can re-introduce the order at the current best price.

Retrieval Market Orders

***bid* order**

$\mathcal{O}_{\mathsf{bid}}$: $\langle$ piece, price $\rangle_{\mathcal{C}_i}$

- piece, the index of the piece requested[a]
- price, the price at which $\mathcal{C}_i$ is paying for one retrieval

***ask* order**

$\mathcal{O}_{\mathsf{ask}}$: $\langle$ piece, price $\rangle_{\mathcal{M}_i}$

- piece, the index of the piece requested
- price, the price at which $\mathcal{M}_j$ is serving the piece for

***deal* order**

$\mathcal{O}_{\mathsf{deal}}$: $\langle$ ask, order $\rangle_{\mathcal{C}_i, \mathcal{M}_j}$

- ask, a cryptographic reference to $\mathcal{O}_{\mathsf{ask}}$ from $\mathcal{C}_i$
- order, a cryptographic reference to $\mathcal{O}_{\mathsf{ask}}$ from $\mathcal{C}_i$

[a] Only pieces stored in Filecoin can be requested

Figure 10: Orders data structures for the Retrieval and Storage Markets

5.2.3 The Storage Market Protocol

In brief, the Storage Market protocol is divided in two phases: *order matching* and *settlement*:

- *Order Matching*: Clients and Storage Miners submit their orders to the orderbook by submitting a transaction to the blockchain (step 1). 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 (step 2).
- *Settlement*: Storage Miners seal their sectors (step 3a), 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 (step 3c).

The Storage Market protocol is explained in detail in Figure 11.

5.3 Retrieval Market

The Retrieval Market allows clients to request retrieval of a specific piece and Retrieval Miners to serve it. Unlike Storage Miners, 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.

5.3.1 Requirements

We design the Retrieval Market protocol accordingly to the following 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 participant will have only partial view of the OrderBook. Hence, we require both parties to gossip their orders.
- **Retrieval without trusted parties**: The impossibility results on fair exchange [10] 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 file. We go around this result by requiring the Retrieval Miner to split their data in 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 a 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 which 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 as previously seen in [11, 12].

5.3.2 Data Structures

Get **Orders**. There are three types of orders in the Retrieval Market: clients create *bid* orders $\mathcal{O}_{\mathsf{bid}}$, Retrieval Miners create *ask* orders $\mathcal{O}_{\mathsf{ask}}$, and *deal* orders $\mathcal{O}_{\mathsf{deal}}$, are created jointly when a Storage Miner and a client agree on a deal. The datastructures of the orders is shown in detail on Figure 10.

Get **Orderbook**. The Orderbook in the Retrieval Market is the set of valid and open *ask*, *bid* and *deal* orders. Unlike the Storage Market, every user has a different view of the orderbook, since the orders are gossiped in the network and each miner and client only keep track of the orders they are interested in.

Storage Market Protocol

Order Matching

1. Storage Miner $\mathcal{M}_i$ and Client $\mathcal{C}_i$ add orders to the OrderBook:
   (a) $\mathcal{M}_i$ creates $\mathcal{O}_{\mathsf{ask}}{}^1, \mathcal{O}_{\mathsf{ask}}{}^2, ..$ and $\mathcal{C}_j$ creates $\mathcal{O}_{\mathsf{bid}}{}^1, \mathcal{O}_{\mathsf{bid}}{}^2, ...$
   (b) Orders are submitted to the blockchain via Put.addOrders($\mathcal{O}^1, \mathcal{O}^2, ..$)
   (c) On success, the orders are added to the OrderBook, the funds from $\mathcal{C}_j$ are deposited and the space from $\mathcal{M}_i$ is reserved.
2. When orders match, involved parties jointly create $\mathcal{O}_{\mathsf{deal}}$ and add it to the OrderBook:
   (a) $\mathcal{M}_i$ and $\mathcal{C}_j$ independently query the OrderBook via Put.matchOrders($\mathcal{O}$).
   (b) If $\mathcal{M}_i$ and $\mathcal{C}_j$ have matching orders :
      - $\mathcal{C}_j$ sends the piece p to $\mathcal{M}_i$ via Put.SendPiece($\mathcal{O}_{\mathsf{bid}}$, $\mathcal{O}_{\mathsf{ask}}$, p)
      - $\mathcal{M}_i$ receives the piece p from $\mathcal{C}_j$ via Put.ReceivePiece($\mathcal{O}_{\mathsf{bid}}$, $\mathcal{O}_{\mathsf{ask}}$, p).
      - $\mathcal{M}_i$ signs $\mathcal{O}_{\mathsf{deal}}$ and sends it to $\mathcal{C}_j$
   (c) $\mathcal{C}_j$ signs $\mathcal{O}_{\mathsf{deal}}$ and adds it to the OrderBook via Put.addOrders($\mathcal{O}_{\mathsf{deal}}$)

Settlement

3. The Network checks if the Storage Miners are correctly storing the pieces:
   (a) When a Storage Miner fills a sector, they *seal* it (they create a unique replica) via Manage.SealSector and submit the proof π_{SEAL} and rt to the blockchain.
   (b) Storage Miners generate new proofs at every epoch and add them to the Filecoin blockchain every Δ_{proof} epochs via Manage.ProveSectors.
   (c) The Network runs Manage.RepairOrders at every epoch. If proofs are missing or invalid, the network tries to repair in the following ways:
      - if any proofs are missing or invalid, it penalizes the Storage Miners by taking part of their collateral,
      - if a large amount of proofs are missing or invalid for more than Δ_{fault} epochs, it considers the Storage Miner *faulty*, settles the order as failed and reintroduces a new order for the same piece into the the market,
      - if every Storage Miner storing this piece is faulty, then the piece is lost and the client gets refunded.
4. When the time of the order is expired or funds run out, if the service was correctly provided, the Network processes the payments, and removes the orders.

Figure 11: Detailed Storage Market protocol

5.3.3 The Retrieval Market Protocol

In brief, the Retrieval Market protocol is divided in two phases: *order matching* and *settlement*:

- *Order Matching*: Clients and Retrieval Miners submit their orders to the orderbook by gossiping their orders (step 1). When orders are matched, the client and the Retrieval Miners establish a micropayment channel (step 2).
- *Settlement*: Retrieval Miners send a 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 (step 4).

The protocol is explained in details in Figure 12.

Retrieval Market Protocol

Order Matching:

1. Retrieval Miners and Clients add orders to the Get.OrderBook:
   (a) Retrieval Miners $\mathcal{M}_i$ creates *ask* orders $(\mathcal{O}_{\mathsf{ask}}{}^1, \mathcal{O}_{\mathsf{ask}}{}^2, ..)$ and Client $\mathcal{C}_j$ creates *bid* orders $(\mathcal{O}_{\mathsf{bid}}{}^1, \mathcal{O}_{\mathsf{bid}}{}^2, ..)$.
   (b) Both $\mathcal{M}_i$ and $\mathcal{C}_j$ gossip their orders in the Filecoin network via Get.addOrders
   (c) Since there is no *commonly shared* orderbook, when users receive orders, they add them to their own orderbook's view. Differently from the Storage Market, these orders are not binding and no resource is committed (e.g. clients don't do any deposit).
2. When orders match, involved parties jointly create $\mathcal{O}_{\mathsf{deal}}$ and add it to the Get.OrderBook:
   (a) Retrieval Miner $\mathcal{M}_i$ and Client $\mathcal{C}_j$ independently run Get.matchOrders that queries their own current Get.OrderBook view.
   (b) Both $\mathcal{M}_i$ and $\mathcal{C}_j$ sign $\mathcal{O}_{\mathsf{deal}}$ and add it to their Get.OrderBook via Get.addOrders (as described before)
   (c) $\mathcal{C}_i$ and $\mathcal{M}_j$ setup a micropayment channel for $\mathcal{O}_{\mathsf{deal}}$

Settlement:

3. Both parties check whether the piece has been delivered:
   (a) $\mathcal{M}_i$ sends the piece p in parts via Get.SendPiece
   (b) $\mathcal{C}_j$ receives the p in parts and for each part, $\mathcal{C}_j$ acknowledges delivery by sending a micropayment via Get.ReceivePiece
4. When the p has been received by $\mathcal{C}_j$, $\mathcal{M}_j$ can present the micropayments to the network and retrieve the payment, both parties remove their orders from the orderbooks.

Figure 12: Detailed Retrieval Market protocol

6 Useful Work Consensus

The Filecoin DSN protocol can be implemented on top of any consensus protocol that allows for verification of the Filecoin's proofs. In this section, we present how we can bootstrap a consensus protocol based on useful work. Instead of wasteful *Proof-of-Work* computation, the work Filecoin miners do generating *Proof-of-Spacetime* is what allows them to participate in the consensus.

Useful Work. We consider the work done by the miners in a consensus protocol to be *useful*, if the outcome of the computation is valuable to the network, beyond securing the blockchain.

6.1 Motivation

While securing the blockchain is of fundamental importance, Proof-of-Work schemes often require solving puzzles whose solutions are *not reusable* or require a substantial amount *wasteful* computation to find.

- **Non-reusable Work**: Most permissionless blockchains require miners to solve a hard computational puzzle, such as inverting a hash function. Often the solutions to these puzzles are useless and do not have any inherent value beyond securing the network. Can we re-purpose this work for something useful?

  Attempts to re-use work: There have been several attempts to re-use mining power for useful computation. Some efforts require miners to perform a special computation alongside the standard *Proof-of-Work*. Other efforts replace *Proof-of-Work* with useful problems that are still hard to solve. For example, Primecoin re-uses miners' computational power to find new prime numbers, Ethereum requires miners to execute small programs alongside with *Proof-of-Work*, and Permacoin offers archival services by requiring miners to invert a hash function while proving that some data is being archived. Although most of these attempts do perform useful work, the amount of wasteful work is still a prevalent factor in these computations.

- **Wasteful Work**: Solving hard puzzles can be really expensive in terms of cost of machinery and energy consumed, especially if these puzzles solely rely on computational power. When the mining algorithm is embarrassingly parallel, then the prevalent factor to solve the puzzle is computational power. Can we reduce the amount of wasteful work?

  Attempts to reduce waste: Ideally, the majority of a network's resources should be spent on useful work. Some efforts require miners to use more energy-efficient solutions. For example, Spacemint requires miners to dedicate disk space rather than computation; while more energy efficient, theses disks are still "wasted", since they are filled with random data. Other efforts replace hard to solve puzzles with a traditional byzantine agreement based on *Proof-of-Stake*, where stakeholders vote on the next block proportional to their share of currency in the system.

We set out to design a consensus protocol with a useful work based on storing users' data.

6.2 Filecoin Consensus

We propose a *useful work* consensus protocol, where the probability that the network elects a miner to create a new block (we refer to this as the *voting power* of the miner) is proportional to their storage currently in use in relation to the rest of the network. We design the Filecoin protocol such that miners would rather invest in storage than in computing power to parallelize the mining computation. Miners offer storage and re-use the computation for proof that data is being stored to participate in the consensus.

6.2.1 Modeling Mining Power

Power Fault Tolerance. In our technical report [13], we present *Power Fault Tolerance*, an abstraction that re-frames byzantine faults in terms of participants' influence over the outcome of the protocol. Every participant controls some *power* of which n is the total power in the network, and f is the fraction of power

controlled by faulty or adversarial participants.

Power in Filecoin. In Filecoin, the *power* p_i^t of miner $\mathcal{M}_i$ at time t is the sum of the $\mathcal{M}_i$'s storage assignments. The *influence* I_i^t of $\mathcal{M}_i$ is the fraction of $\mathcal{M}_i$'s power over the total power in the network.

In Filecoin, *power* has the following properties:

- *Public*: The total amount of storage currently in use in the network is public. By reading the blockchain, anyone can calculate the storage assignments of each miner - hence anyone can calculate the *power* of each miner and the total amount of power at any point in time.
- *Publicly Verifiable*: For each storage assignment, miners are required to generate *Proofs-of-Spacetime*, proving that the service is being provided. By reading the blockchain, anyone can verify if the power claimed by a miner is correct.
- *Variable*: At any point in time, miners can add new storage in the network by pledging with a new sector and filling the sector. In this way, miners can change their amount of power they have through time.

6.2.2 Accounting for Power with *Proof-of-Spacetime*

Every Δ_{proof} blocks[6], miners are required to submit *Proofs-of-Spacetime* to the network, which are only successfully added to the blockchain if the majority of power in the network considers them valid. At every block, every full node updates the AllocTable, adding new storage assignments, removing expiring ones and marking missing proofs.

The power of a miner $\mathcal{M}_i$ can be calculated and verified by summing the entries in the AllocTable, which can be done in two ways:

- **Full Node Verification**: If a node has the full blockchain log, run the NetworkProtocol from the genesis block to the current block and read the AllocTable for miner $\mathcal{M}_i$. This process verifies every *Proof-of-Spacetime* for the storage currently assigned to $\mathcal{M}_i$.
- **Simple Storage Verification**: Assume a light client has access to a trusted source that broadcasts the latest block. A light client can request from nodes in the network: (1) the current AllocTable entry for miner $\mathcal{M}_i$, (2) a Merkle path that proves that the entry was included in the state tree of the last block, (3) the headers from the genesis block until the current block. In this way, the light client can delegate the verification of the *Proof-of-Spacetime* to the network.

The security of the power calculation comes from the security of *Proof-of-Spacetime*. In this setting, PoSt guarantees that the miner cannot lie about the amount of assigned storage they have. Indeed, they cannot claim to store more than the data they are storing, since this would require spending time fetching and running the slow PoSt.Setup, and they cannot generate proofs faster by parallelizing the computation, since PoSt.Prove is a sequential computation.

6.2.3 Using Power to Achieve Consensus

We foresee multiple strategies for implementing the Filecoin consensus by extending existing (and future) Proof-of-Stake consensus protocols, where stake is replaced with assigned storage. While we foresee improvements in Proof-of-Stake protocols, we propose a construction based on our previous work called Expected Consensus [14]. Our strategy is to elect at every round one (or more) miners, such that the probability of winning an election is proportional to each miner's assigned storage.

Expected Consensus. The basic intuition of Expected Consensus EC is to *deterministically*, *unpredictably*, and *secretly* elect a small set of leaders at each epoch. On expectation, the number of elected leaders per

[6] Δ_{proof} is a system parameter.

epoch is 1, but some epochs may have zero or many leaders. Leaders extend the chain by creating a block and propagating it to the network. At each epoch, the chain is extended with one or multiple blocks. In case of a leaderless epoch, an empty block is added to the chain. Although the blocks in chain can be linearly ordered, its data structure is a direct acyclic graph. EC is a probabilistic consensus, where each epoch introduces more certainty over previous blocks, eventually reaching enough certainty that the likelihood of a *different* history is sufficiently small. A block is *committed* if the majority of the participants add their weight on the chain where the block belongs to, by extending the chain or by signing blocks.

Electing Miners. At every epoch, each miner checks if they are elected leader, this is done similarly to previous protocols: CoA [15], Snow White [16], and Algorand [17].

Definition 6.1. (EC Election in Filecoin) A miner $\mathcal{M}_i$ is a leader at time t if the following condition is met:

$$\mathcal{H}\Big(\big\langle t||\mathsf{rand}(t)\big\rangle_{\mathcal{M}_i}\Big)/2^L \leq \frac{p_i^t}{\Sigma_j p_j^t}$$

Where $\mathsf{rand}(t)$ is a public randomness available that can be extracted from the blockchain at epoch t, p_i^t is the power of $\mathcal{M}_i$. Consider the size of $\mathcal{H}(m)$ to be L for any m, $\mathcal{H}$ to be a secure cryptographic hash function and $\langle m\rangle_{\mathcal{M}_i}$ to be a message m signed by $\mathcal{M}_i$, such that:

$$\langle m\rangle_{\mathcal{M}_i} \coloneqq \Big((m), \mathsf{SIG}_{\mathcal{M}_i}\big(\mathcal{H}(m)\big)\Big)$$

In Figure 13, we describe the protocol between a miner (ProveElect) and a network node (VerifyElect).

This election scheme provides three properties: *fairness*, *secrecy* and *public verifiability*.

- *Fairness*: each participant has only one trial for each election, since signatures are deterministic and t and $\mathsf{rand}(t)$ are fixed. Assuming $\mathcal{H}$ is a secure cryptographic hash function, then $\mathcal{H}(\langle t||\mathsf{rand}(t)\rangle_{\mathcal{M}_i})/2^L$ must be a real number uniformly chosen from (0, 1). Hence, the probability for the equation to be true must be $p_i^t/\Sigma_j p_j^t$, which is equal to the miner's portion of power within the network. Because this probability is linear in power, this likelihood is preserved under splitting or pooling power. Note that the random value $\mathsf{rand}(t)$ is not known before time t.
- *Secret*: an efficient adversary that does not own the secret key $\mathcal{M}_i$ can compute the signature with negligible probability, given the assumptions of digital signatures.
- *Public Verifiability*: an elected leader $i \in L^t$ can convince a efficient verifier by showing t, $\mathsf{rand}(t)$, $\mathcal{H}(\langle t||\mathsf{rand}(t)\rangle_i)/2^L$; given the previous point, no efficient adversary can generate a proof without having a winning secret key.

EC Election

Storage Miner at epoch t

ProveElect$(r, t, \mathcal{M}_i) \to \{\perp, \pi_i^t\}$

1. Compute $\mathcal{H}(\langle t||r\rangle_i)/2^L \leq \frac{p_i^t}{\Sigma_j p_j^t}$
   - on success, output $\pi_i^t = \langle t, r\rangle_i$
   - otherwise output $\perp$

Network node on receiving a block at epoch t

VerifyElect$(\pi_t^i, t, \mathcal{M}_i) \to \{\perp|\top\}$

1. Check if π_i^t is a valid signature from user $\mathcal{M}_i$ on t and r
2. Check if p_i^t is the power from $\mathcal{M}_i$ at time t
3. Test if $\mathcal{M}_i$ is elected leader $\mathcal{H}(\pi_i^t)/2^L < \frac{p_i^t}{\Sigma_j p_j^t}$
   - on success, output $\top$
   - otherwise output $\perp$

Figure 13: Leader Election in the Expected Consensus protocol

7 Smart Contracts

Filecoin provides two basic primitives to the end users: Get and Put. These primitives allow clients to store data and retrieve data from the markets at their preferred price. While the primitives cover the default use cases for Filecoin, we enable for more complex operations to be designed on top of Get and Put by supporting a deployment of smart contracts. Users can program new fine-grained storage/retrieval requests that we classify as *File Contracts* as well as generic *Smart Contracts*. We integrate a *Contracts* system (based on [18]) and a *Bridge* system to bring Filecoin storage in other blockchain, and viceversa, to bring other blockchains' functionalities in Filecoin.

We expect a plethora of smart contracts to exist in the Filecoin ecosystem and we look forward to a community of smart-contract developers.

7.1 Contracts in Filecoin

Smart Contracts enable users of Filecoin to write stateful programs that can spend tokens, request storage/retrieval of data in the markets and validate storage proofs. Users can interact with the smart contracts by sending transactions to the ledger that trigger function calls in the contract. We extend the Smart Contract system to support Filecoin specific operations (e.g. market operations, proof verification).

Filecoin supports contracts specific to data storage, as well as more generic smart contracts:

- **File Contracts**: We allow users to program the conditions for which they are offering or providing storage services. There are several examples worth mentioning: (1) contracting miners: clients can specify in advance the miners offering the service without participating in the market, (2) payment strategies: clients can design different reward strategies for the miners, for example a contract can pay the miner incresingly higher through time, another contract can set the price of storage informed by a trusted oracle, (3) ticketing services: a contract could allow a miner to deposit tokens and to pay for storage/retrieval on behalf of their users, (4) more complex operations: clients can create contracts that allow for data update.
- **Smart Contracts**: Users can associate programs to their transactions like in other systems (as in Ethereum [18]) which do not directly depend on the use of storage. We foresee applications such as: decentralized naming systems, asset tracking and crowdsale platforms.

7.2 Integration with other systems

Bridges are tools that aim at connecting different blockchains; while still work in progress, we plan to support cross chain interaction in order to bring the Filecoin storage in other blockchain-based platforms as well as bringing functionalities from other platforms into Filecoin.

- **Filecoin in other platforms**: Other blockchain systems such as Bitcoin [19], Zcash [20] and in particular Ethereum [18] and Tezos, allow developers to write smart contracts; however, these platforms provide very little storage capability and at a very high cost. We plan to provide a *bridge* to bring storage and retrieval support to these platforms. We note that IPFS is already in use by several smart contracts (and protocol tokens) as a way to reference and distribute content. Adding support to Filecoin would allow these systems to guarantee storage of IPFS content in exchange of Filecoin tokens.
- **Other platforms in Filecoin**: We plan to provide *bridges* to connect other blockchain services with Filecoin. For example, integration with Zcash would allow support for sending requests for storing data in privacy.

8 Future Work

This work presents a clear and cohesive path toward the construction of the Filecoin network; however, we also consider this work to be a starting point for future research on decentralized storage systems. In this section we identify and populate three categories of future work. This includes work that has been completed and merely awaits description and publication, open questions for improving the current protocols, and formalization of the protocol.

8.1 On-going Work

The following topics represent ongoing work.

- A specification of the Filecoin state tree in every block.
- Detailed performance estimates and benchmarks for Filecoin and its components.
- A full implementable Filecoin protocol specification.
- A sponsored-retrieval ticketing model where any client C1 can sponsor the download of another client C2 by issuing per-piece bearer-spendable tokens.
- A Hierarchical Consensus protocol where Filecoin subnets can partition and continue processing transactions during temporary or permanent partitions.
- Incremental blockchain snapshotting using SNARK/STARK
- Filecoin-in-Ethereum interface contracts and protocols.
- Blockchain archives and inter-blockchain stamping with Braid.
- Only post *Proofs-of-Spacetime* on the blockchain for conflict resolution.
- Formally prove the realizations of the Filecoin DSN and the novel *Proofs-of-Storage*.

8.2 Open Questions

There are a number of open questions whose answers have the potential to substantially improve the network as a whole, despite the fact that none of them have to be solved before launch.

- A better primitive for the *Proof-of-Replication* Seal function, which ideally is O(n) on decode (not O(nm)) and publicly-verifiable without requiring SNARK/STARK.
- A better primitive for the *Proof-of-Replication* Prove function, which is publicly-verifiable and transparent without SNARK/STARK.
- A transparent, publicly-verifiable *Proof-of-Retrievability* or other Proof-of-Storage.
- New strategies for retrieval in the Retrieval Market (e.g. based on probabilistic payments, zero knowledge contingent payments)
- A better secret leader election for the Expected Consensus, which gives exactly one elected leader per epoch.
- A better trusted setup scheme for SNARKs that allows incremental expansion of public parameters (schemes where a sequence of MPCs can be run, where each additional MPC strictly lowers probability of faults and where the output of each MPC is usable for a system).

8.3 Proofs and Formal Verification

Because of the clear value of proofs and formal verification, we plan to prove many properties of the Filecoin network and develop formally verified protocol specifications in the coming months and years. A few proofs are in progress and more in mind. But it will be hard, long-term work to prove many properties of Filecoin (such as scaling, offline).

- Proofs of correctness for Expected Consensus and variants.
- Proof of correctness for Power Fault Tolerance asynchronous 1/2 impossibility result side-step.
- Formulate the Filecoin DSN in the universal composability framework, describing Get, Put and Manage as ideal functionalities and prove our realizations.
- Formal model and proofs for automatic self-healing guarantees.
- Formally verify protocol descriptions (e.g. TLA+ or Verdi).
- Formally verify implementations (e.g. Verdi).
- Game theoretical analysis of Filecoin's incentives.

Acknowledgements

This work is the cumulative effort of multiple individuals within the Protocol Labs team, and would not have been possible without the help, comments, and review of the collaborators and advisors of Protocol Labs. Juan Benet wrote the original Filecoin whitepaper in 2014, laying the groundwork for this work. He and Nicola Greco developed the new protocol and wrote this whitepaper in collaboration with the rest of the team, who provided useful contributions, comments, review and conversations. In particular David "davidad" Dalrymple suggested the orderbook paradigm and other ideas, Matt Zumwalt improved the structure of the paper, Evan Miyazono created the illustrations and finalized the paper, Jeromy Johnson provided insights while designing the protocol, and Steven Allen contributed insightful questions and clarifications. We also thank all of our collaborators and advisor for useful conversations; in particular Andrew Miller and Eli Ben-Sasson.

Previous version: QmYcf7X6ygKisoVS7EApqY3gxcKW1MigF57zc1cdXjZWrQ

References

[1] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System. 2014.

[2] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609. Acm, 2007.

[3] Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. Acm, 2007.

[4] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107. Springer, 2008.

[5] Protocol Labs. Technical Report: Proof-of-Replication. 2017.

[6] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[7] Nir Bitansky, Alessandro Chiesa, and Yuval Ishai. Succinct non-interactive arguments via linear interactive proofs. Springer, 2013.

[8] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*, pages 90–108. Springer, 2013.

[9] Eli Ben-Sasson, Iddo Bentov, Alessandro Chiesa, Ariel Gabizon, Daniel Genkin, Matan Hamilis, Evgenya Pergament, Michael Riabzev, Mark Silberstein, Eran Tromer, et al. Computational integrity with a public random string from quasi-linear pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 551–579. Springer, 2017.

[10] Henning Pagnia and Felix C Gärtner. On the impossibility of fair exchange without a trusted third party. Technical report, Technical Report TUD-BS-1999-02, Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany, 1999.

[11] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2015.

[12] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *arXiv preprint arXiv:1702.05812*, 2017.

[13] Protocol Labs. Technical Report: Power Fault Tolerance. 2017.

[14] Protocol Labs. Technical Report: Expected Consensus. 2017.

[15] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract] y. *ACM SIGMETRICS Performance Evaluation Review*, 42(3):34–37, 2014.

[16] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. 2016.

[17] Silvio Micali. Algorand: The efficient and democratic ledger. *arXiv preprint arXiv:1607.01341*, 2016.

[18] Vitalik Buterin. Ethereum <https://ethereum.org/>, April 2014. URL `https://ethereum.org/`.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[20] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.