

0G: Towards Data Availability 2.0

0G Labs
zg@zglabs.org

June 12, 2024
DRAFT Version

Abstract

The data availability problem stems from the demand for off-chain verification of executed states, which in turn arises from the trade-off between scalability and security of blockchain systems. The increasing prominence of Layer 2 networks and decentralized AI platforms has made data availability even more crucial and has also made its scalability the primary challenge at present. Although some existing projects, e.g., Celestia, EigenDA, and Avail, have taken initial steps to tackle this challenge, their scalability is far from sufficient to meet the enormous demands anticipated in the foreseeable future. 0G proposes a new design of the data availability system which can be infinitely scalable and reliable in practice. It builds the data availability layer directly on top of a decentralized storage system and addresses the scalability issue by piling an arbitrary number of independent consensus networks sharing the same security and minimizing the data transfer volume required for broadcast in each of them. Its general decentralized storage design further enables it to support a wide variety of data availability scenarios not limited to Layer 2 networks, but also inclusive of decentralized AI infrastructures.

Contents

1	Introduction	3
2	Overview of the System Design	6
3	Design of the 0G Storage	8

4	Log System Protocol	9
4.1	Protocol Overview	10
4.2	Storage Granularity	11
4.3	Data Flow	12
5	Incentive Mechanism	12
5.1	Storage Request Pricing	13
5.2	Proof of Random Access	13
5.2.1	Fairness for Small Miners	14
5.2.2	Disincentivize Storage Outsourcing	14
5.2.3	Disincentivize Distributed Mining	15
5.2.4	Formal Definition for PoRA Mechanism	16
5.3	Mining Reward	17
5.4	How to Balance the Number of Replicas?	18
5.5	How to Incentivize Data Sharing?	18
6	Key-Value Runtime	20
7	Transactional Processing on Key-Value Store	21
8	Roadmap	23

1 Introduction

The trade-off between scalability and security in blockchain systems has given rise to a demand for off-chain verification of executed states. This further introduces the data availability problem [1]. Specifically, the off-chain verifier, usually the light client, needs to be able to access the entire transaction history data to verify the execution of the transactions. Currently, a major application scenario for data availability is the Layer 2 networks of Ethereum [2]. The blocks containing executed transactions in Layer 2 networks need to be published and stored somewhere for light client to conduct further verification. Similar requirements also exist in decentralized AI infrastructures where the results of training or inference tasks on devices in decentralized networks need to be further verified due to the demands of users or system incentives. For example, in ORA/OpML [3] scenario where an optimistic mechanism is employed, it requires participants to provide fraud proofs for specific AI tasks during a challenge window and the proofs may need to contain the data and models used in those tasks. Some incentive mechanisms may also require randomly choosing and verifying old historical tasks to give rewards accordingly, and hence to achieve a good trade-off between verification cost and effectiveness.

Public Layer 1 blockchain like Ethereum can be an option to provide the service for data availability usage, but it brings significant, sometimes unacceptable, extra costs to Layer 2 applications (e.g., \$140 for one OP [4] block with 218KB). It is also extremely inefficient and unscalable with a data throughput at about 83 KBps and a finality time of 12 minutes. Recently, Celestia [5] was proposed to provide a cheaper solution by separating the data availability module from Ethereum to become a shared platform. It is itself a consensus-based distributed ledger that is specifically used to serve data availability requests. Although it is cheaper due to the lower gas price for storing transaction data, it is still far from a scalable solution because every data block of the data availability request has to be broadcast in its consensus network to get the multi-signed certification from all the validators. This prevents it from effectively utilizing aggregated network bandwidth, resulting in an overall data throughput unable to surpass the bottleneck of 10 MBps, which makes it hard to support the huge amount of data availability requests brought by the proliferation of Layer 2s, Layer 3s,

and decentralized AI platforms. Furthermore, Celestia does not have a scalable storage solution for data blobs with availability requirements, meaning it must rely on a massive data availability sampling done by large amounts of light clients who each store slices of all data. This further slows down the system due to the enormous bandwidth consumption and forces its block time to more than 10 seconds.

In contrast, EigenDA [6] provides a data availability solution with horizontal scalability in design. Instead of employing a consensus layer, it relies on a quorum of data availability nodes that hold a security assumption where the majority of nodes are honest. Each data availability node is an EigenLayer [7] restaker that stakes its ETH to participate in the quorum protocol. The protocol does not involve a well-designed incentive mechanism for storing data blocks. Instead, it introduces a slashing-based mechanism to punish the node that breaks its promise on the data storage and access service. However, this brings to the node the risk of unexpected loss of the staked assets since it is hard to distinguish between malicious data withholding and an accidental failure (probably caused by a DDoS attack). This greatly reduces the willingness of nodes to participate which limits the actual scalability that it can achieve. In addition, although the data blocks can be maintained in the nodes with availability, without a general decentralized storage system, it is hard to maintain the availability of data index, and hence cannot achieve the available retrieval.

One important factor in a quorum-based system is the quorum construction. In EigenDA, the process of constructing the quorums is too arbitrary. The data availability node operator itself decides which quorum to join and the clients choose by themselves which quorum to use for data availability service. In this case, it is easy for malicious clients to collude with data availability nodes to fool users, jeopardizing security. Furthermore, the aggregated signatures of the data availability nodes in a quorum need to be submitted to the Ethereum smart contract for further verification to confirm the data availability property, which can also become a significant bottleneck for large data throughput.

To better address the scalability issue of data availability, we propose 0G (0G in short), the first data availability system with a built-in general-purpose storage layer that is infinitely scalable and decentralized. The scalability of 0G comes from the following architecture design points. First,

the basic setup of the storage layer of OG consists of a storage network connecting with a separate consensus network. Each data block written into the storage network accompanies a transaction on the consensus network to record the commitment and the ordering of the data. To scale the data throughput infinitely, the storage network is organized in a partitioned way and connects to an arbitrary number of consensus networks that run in parallel and independently. The data requests from different independent applications may be written into different storage partitions and their data commitments can be recorded into different consensus networks simultaneously. All the consensus networks share the same set of validators with the same staking status so that they keep the same level of security. Each storage node actively participates in a mining process by submitting proof of accessibility for a specific piece of data to a smart contract deployed on a consensus network. Once the proof is validated by the smart contract, the storage node gets rewarded accordingly. This incentive-based mechanism rewards the nodes for contributions rather than punishing them for misbehaviors, so it can better encourage nodes to participate in the maintenance of the network, and hence can promote the network to achieve better scalability in practice.

Secondly, OG embraces the idea of separating the workflow of data availability into both the Data Publishing Lane and the Data Storage Lane. The large volume of data transfers happens on the Data Storage Lane that is supported by the storage layer which achieves horizontal scalability through well-designed partitioning, while the Data Publishing Lane guarantees the data availability property by checking the aggregated signatures of the corresponding Storage Nodes on the consensus network, which only requires tiny data flowing through the consensus protocol to avoid the broadcasting bottleneck. Note that, recently, there have been debates within the Ethereum community regarding data availability, tending to narrowly define it as data publication. However, this can be misleading. In fact, data storage is an integral part of data availability because it must answer the question of where the data is published. Without a well-designed and reliable storage layer, systems like Celestia have to inject entire data blocks into the consensus system and store them on full nodes.

The Storage Nodes are important for OG data availability service and must also be consensus network validators. Similar to EigenDA, the data

availability of 0G is based on a majority honesty assumption in the quorum of Storage Nodes with staking. Differently, the data availability quorums of 0G are constructed randomly by the consensus system through verifiable random function (VRF) which theoretically guarantees the same distribution of the honest participants as in the validator set of the entire consensus network, so that the data availability client cannot collude with them. In other words, as subsets of all the validators, the quorums share the same security property as the entire consensus network. The aggregated signatures of the quorum will be submitted to the consensus of 0G for data availability confirmation, which can be orders of magnitude faster and more efficient than Ethereum. Combined with the multi-consensus design of 0G with infinite scalability, the consensus protocol will not become the bottleneck of the data throughput. In addition, as with EigenDA, the data availability client needs to conduct erasure coding for the data to have it split into chunks. In practice, this process is very costly and can be the major bottleneck for the throughput of a single client. 0G provides a GPU-accelerated solution for the erasure coding process to significantly speed it up.

Last but not least, 0G storage is also designed as a general storage system with multiple stacks of abstractions and structures including an append-only log layer for archiving unstructured data and a key-value layer for managing mutable and structured data. This allows 0G to support reliable data indexing and a greater variety of data availability scenarios.

2 Overview of the System Design

0G system consists of a data availability layer (0G DA) on top of a decentralized storage system (0G Storage). There are one or multiple separate consensus networks that are part of both the 0G DA and the 0G Storage. For 0G Storage, each consensus network is responsible for determining the ordering of the uploaded data blocks, realizing the storage mining verification and the corresponding incentive mechanism through smart contract. For 0G DA, the consensus is in charge of guaranteeing the data availability property of each data block via verifying the aggregated signatures from the corresponding data availability quorum.

Figure 1 illustrates the architecture of the 0G system when deploying a single consensus network. When a data block enters the 0G DA, it is first

erasure coded and organized into multiple consecutive chunks through erasure coding. The merkle root as a commitment of the encoded data block is then submitted to the consensus layer to keep the order of the data entering the system. The chunks are then dispersed to different Storage Nodes in a quorum in 0G Storage. The Storage Nodes periodically participate the mining process by interacting with the consensus network to accrue rewards from the system. The DA client then collects the aggregated signatures from the quorum and submits the signatures to the consensus for verification.

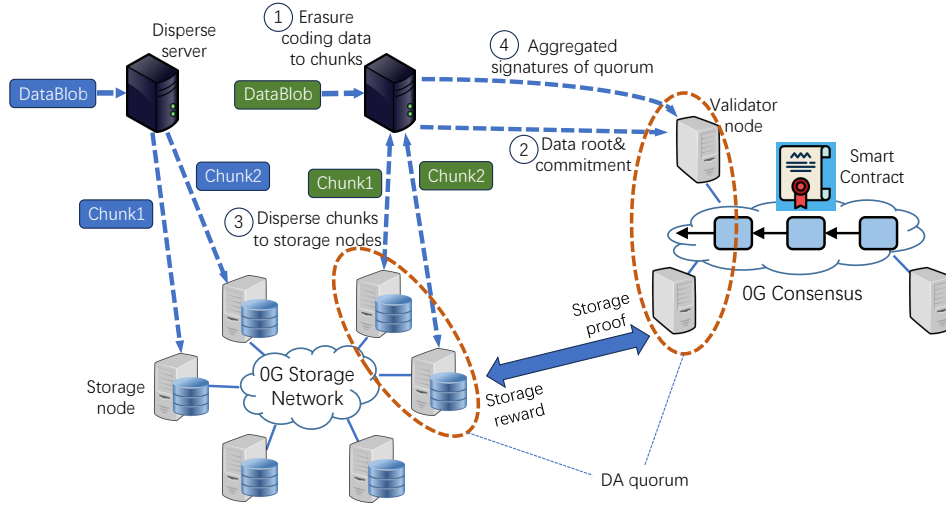


Figure 1: The architecture of the 0G system.

When deployed with multiple POS-based consensus networks, each validator participates in the maintenance of all the consensus networks by using a shared staking status. The shared staking status can be recorded in a smart contract on one of the multiple consensus networks or a blockchain network outside of the 0G consensus network set, e.g. Ethereum. Assuming C_0 is the consensus network that maintains the shared staking status, the token T_0 on C_0 would be staked for POS voting. When token T_i is produced on network C_i ($i \neq 0$) via incentives, it can be mapped to T_0 on C_0 by burning T_i on C_i through a secure cross-chain bridge channel. When a validator maintains the POS protocol in any network C_i , it always uses the amount of staked T_0 on C_0 as its voting power in C_i . Therefore, all the network C_i share the same level of POS security. By using Ethereum as network

C_0 , it brings the benefit of incarnating the token T_0 as a standard ERC20 token and also enables the easy integration with existing popular restaking framework like EigenLayer [7]. Note that, the sources of the shared staking status may also come from the high value tokens (e.g., BTC and ETH) in restaking frameworks like EigenLayer and Babylon [8], so that to make 0G consensus reach the Bitcoin or Ethereum level of security. As illustrated in Figure 2, this shared staking design introduces the unlimited scalability to 0G system.

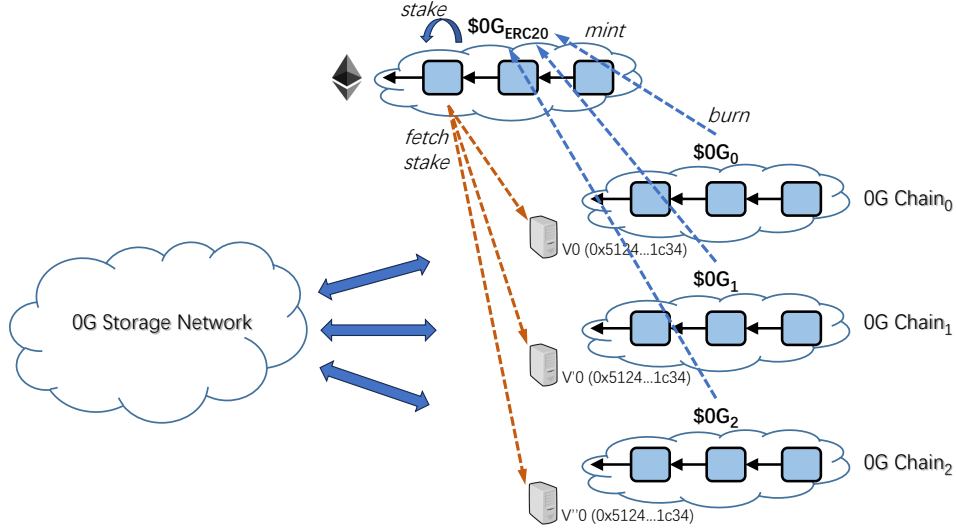


Figure 2: The infinite scalability of the 0G system.

3 Design of the 0G Storage

0G Storage employs layered design targeting to support different types of decentralized applications. Figure 3 shows the overview of the full stack layers of 0G Storage.

The lowest is a log layer that is a decentralized system consisting of multiple Storage Nodes that form a storage network with a built-in incentive mechanism that rewards data storage. The ordering of the uploaded data is guaranteed by a sequencing mechanism that provides log-based semantics and abstraction. This layer is used to store unstructured raw data for permanent persistency.

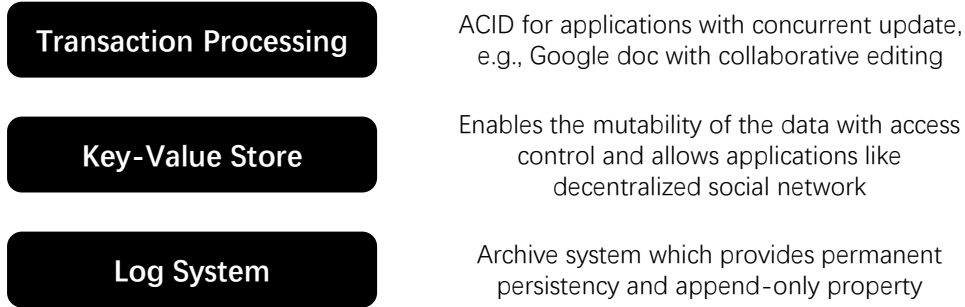


Figure 3: Full stack solution of 0G Storage.

On top of the log layer, 0G Storage provides a Key-Value store runtime to manage structured data with mutability. Multiple key-value store nodes share the underlying log system, putting the structured key-value data into the log entry and appending it to the log system. They play the log entries in the shared log to construct the consistent state snapshot of the key-value store. The throughput and latency of the key-value store are bounded by the log system, so that the efficiency of the log layer is critical to the performance of the entire system. The key-value store can associate access control information with the keys to manage the update permission for the data. This enables the applications like social network, e.g., decentralized Twitter, which requires the maintenance for the ownership of the messages created by the users.

0G Storage further provides transactional semantics on the key-value store runtime to enable concurrent updates for the keys from multiple users who have the write access permission. The total order of the log entries guaranteed by the underlying log system provides the foundation for the concurrency control of the transactional executions on top of the key-value store. With this capability, 0G Storage can support decentralized applications like collaborative editing and even database workloads.

4 Log System Protocol

The log layer of 0G Storage provides decentralized storage service via a permissionless network of Storage Nodes. These Storage Nodes collaboratively serve archived data, where each node optionally specifies which portion of data it keeps in local storage.

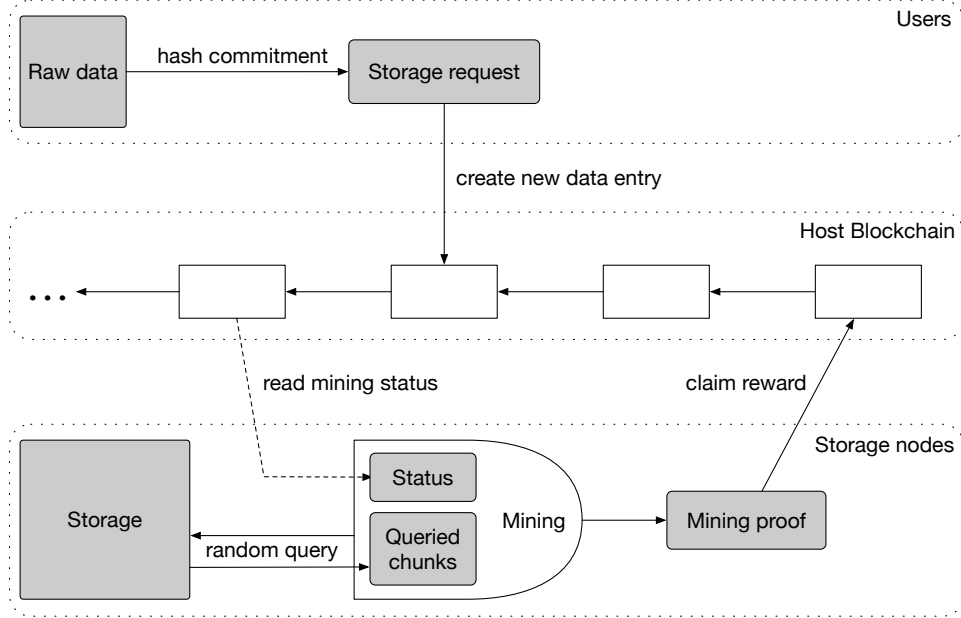


Figure 4: Overview of 0G Storage protocol. Data creation and reward distribution are fully decoupled: users directly submit storage requests to the 0G Storage contract deployed on host blockchain; Storage Nodes claim rewards by proving to the contract that they have the ability to answer random queries to archived data. 0G tokens are handled in another isolated contract, which enables non-blocking token transfer.

4.1 Protocol Overview

As a decentralized storage infrastructure, 0G Storage prefers to improve existing blockchain ecosystems rather than build yet another “storage service” blockchain from scratch.

The storage state of 0G Storage network is maintained in a smart contract deployed on an existing blockchain. This contract is called the 0G Storage *contract* and the underlying blockchain is called the *host blockchain* or *host chain* for short.

The design of 0G Storage network fully decouples data creation, reward distribution, and token circulation.

The 0G Storage contract is responsible for data storage request processing, data entry creation, and reward distribution. Data storage requests are submitted by users who wish to store data in the 0G Storage network, where each request includes necessary metadata such as data size and com-

mitments, and it comes along with the payment for storage service. Data entries are created for accepted data requests, keeping record of stored data while reward distribution is handled independently through a mining process. Storage Nodes submit mining proofs to the 0G Storage contract to claim rewards for maintaining the 0G Storage network.

The token circulation of 0G is fully embedded into the host chain ecosystem, as an ERC-20 token maintained by another contract called the *ZG ledger*. This embedding design brings significant advantages:

- **Simplicity:** there is no need to maintain a full-fledged consensus protocol, which reduces complexity and enables 0G Storage to focus on decentralized storage service.
- **Safety:** the consensus is outsourced to the host blockchain, and hence inherits the security of the host blockchain. Typically the more developed host blockchain would provide a stronger safety guarantee than a newly built blockchain.
- **Accessibility:** every smart contract on the host blockchain is able to access the original state of 0G directly, without relying on some trusted off-chain notary. This difference is essential comparing to the projection of an external ledger managed by a third-party.
- **Composability:** 0G tokens (ZG) can always be transferred directly on the host blockchain, like any other ERC-20 tokens. This is much more convenient than typical layer-2 ledgers, where transactions are first processed by layer-2 validators and then committed to the host chain after a significant latency. This feature empowers 0G Storage stronger composability as a new lego to the ecosystem.

4.2 Storage Granularity

The log layer of 0G Storage is updated (append-only) at the granularity of *log entries*, where every entry is created by a storage-request transaction sent to the 0G Storage contract. The log layer is akin to a filesystem, with every log entry corresponding to a file.

The log system operates at the level of fixed-size *sectors*, with each sector storing 256 B of data. To avoid one sector being shared by distinct log entries, every log entry must be padded to multiple sectors.

The mining process of 0G Storage requires proving data accessibility to random challenge queries. To maximize the competitive edge of SSD storage, the challenge queries are set to the level of 256 KB *chunks*, *i.e.* 1024 sectors. As every challenge query requires the miner to prove accessibility to a whole chunk of data, Storage Nodes would maintain data at the granularity of chunks.

4.3 Data Flow

In 0G Storage, committed data are organized sequentially. Such a sequence of data is called a *data flow*, which can be interpreted as a list of data entries or equivalently a sequence of fixed-size data sectors. Thus, every piece of data in 0G can be indexed conveniently with a universal offset. This offset will be used to sample challenges in the mining process of PoRA.

The default data flow is called the “main flow” of 0G, and it incorporates all new log entries (unless otherwise specified) in an append-only manner.

There are also specialized flows that only accept some category of log entries, *e.g.* data related to a specific application. The most significant advantage of specialized flows is a consecutive addressing space, which may be crucial in some use cases. Furthermore, a specialized flow can apply customized storage price, which is typically significantly higher than the floor price of the default flow, and hence achieves better data availability and reliability.

5 Incentive Mechanism

This section describes the incentive mechanism design of the 0G Storage, which consists of two types of actors: users and miners (a.k.a. Storage Nodes). Users pay tokens (ZG) to create data entries in the log and add data to the network, while miners provide data services and receive tokens (ZG) as a reward from the network. The payment from users to miners is mediated by the 0G network, as the service is sustained by the whole network rather than any specific miner.

0G Storage implements storage service in a “pay once, storage forever” manner. Users pay a one-shot storage endowment for each created data entry, and thereafter the endowment is used to incentivize miners who maintain

that data entry.

The storage endowment is maintained per data entry, and a miner is only eligible for storage rewards from data entries that they have access to. The total storage reward paid for a data entry is independent of the popularity of that data entry. For instance, a popular data entry stored by many miners will be frequently mined, but the reward is amortized amongst those miners. On the other hand, a less popular data entry that is rarely mined would have its storage reward accumulate and hence provide a higher payoff to miners who store this rare data entry.

5.1 Storage Request Pricing

The cost of each 0G Storage request is composed of two parts: a) fee and b) storage endowment. The fee part is paid to host chain miners/validators for invoking the 0G contract to process storage requests and add new data entries into the log, which is priced like any other smart contract invocation transaction. In what follows we focus on the storage endowment part, which supports the perpetual reward to 0G Storage miners who serve the corresponding data.

Given a data storage request SR with specific amount of endowment $SR_{endowment}$ and size of committed data SR_{data_size} (measured in number of 256 B sectors), the unit price of SR is calculated as follows:

$$SR_{unit_price} = SR_{endowment} / SR_{data_size} \quad (1)$$

This unit price SR_{unit_price} must exceed a globally specified lower bound to be added to the log, otherwise the request will be pending until when the lower bound decreases below SR_{unit_price} (in the meantime miners will most likely not store this unpaid data). Users are free to set a higher unit price SR_{unit_price} , which would motivate more Storage Nodes mining on that data entry and hence lead to better data availability.

5.2 Proof of Random Access

The 0G network adopts a Proof of Random Access (PoRA) mechanism to incentivize miners to store data. By requiring miners to answer randomly produced queries to archived data chunks, the PoRA mechanism establishes

the relation between mining proof generation power and data storage. Miners answer the queries repeatedly and computes an output digest for each loaded chunk until find a digest satisfies the mining difficulty (i.e., has enough leading zeros). PoRA will stress the miners' disk I/O and reduce their capability in responding user queries. So 0G Storage adopts intermittent mining, in which a mining epoch starts with a block generation at a specific block height on the host chain and stops when a valid PoRA is submitted to the 0G Storage contract.

In a strawman design, a PoRA iteration consists of a computing stage and a loading stage. In the computing stage, a miner computes a random recall position (the universal offset in the flow) based on an arbitrary picked random nonce and a mining status read from the host chain. In the loading stage, a miner loads the archived data chunks at the given recall position, and computes output digest by hashing the tuple of mining status and the data chunks. If the output digest satisfies the target difficulty, the miner can construct a legitimate PoRA consists of the chosen random nonce, the loaded data chunk and the proof for the correctness of data chunk to the mining contract.

In the following, we introduce some major considerations in improving the fairness in PoRA mining and formalize the PoRA mining mechanism.

5.2.1 Fairness for Small Miners

When the storage size of the 0G Storage network significantly exceeds the storage capacity of a single machine, a single machine will take almost all the time in finding an available recall position. This makes the PoRA becomes proof of work. To make the PoRA mining process be friendly to small miners with a single machine, the mining range is limited to a threshold 8 TB. When the size of archived data chunks exceeds 8 TB, a miner must specify a mining range over the data flow sequence in size of 8 TB. For large miners have enough machines to store all the data, it can mine on different data ranges concurrently.

5.2.2 Disincentivize Storage Outsourcing

To promote the whole network storing enough replicas of data chunks, 0G Storage limits the reward share of a single miner with a single storage replica.

PoRA mechanism seals the data for each miner in different ways, challenges the accessing of sealed data, and prevents mining with data chunks from others. If a miner outsources the data storage and queries the archived data chunks in PoRA mining, it must seal the answering data to compute the output digest, which is a large cost. A miner can seal and unseal tens of MB data chunks by a single thread, so it can still serving the user queries efficiently. As reading data from SSDs can reach up to 7 GB/s, the data sealing is a heavy task for PoRA mining. As long as the cost imposed by data sealing exceeds the cost of purchasing disks and synchronizing data, the miners intend to store a replica of data chunks instead of sealing data on PoRA mining.

More specifically, every 4KB data chunk will sealed by the miner ID and other context data. Suppose `seal_seed` is a 32-byte digest of the miner ID and the context data, figure 5 defines the seal process.

```

Input  : seal_seed, unsealed_data
Output: sealed_data
1 Regard unsealed_data in length of 4 KB as an array of 32-byte elements  $\vec{d}$ 
2  $h \leftarrow \text{seal\_seed}$ 
3 for  $i$  from 0 to 127 do
4    $\vec{d}[i] \leftarrow \vec{d}[i] \text{ XOR } h$ 
5    $h \leftarrow \text{Keccak256}(\vec{d}[i])$ 
6 Regard  $\vec{d}$  as a 4-KB data chunk sealed_data

```

Figure 5: Seal 4 KB data chunks

5.2.3 Disincentivize Distributed Mining

As a mining mechanism for incentivizing data storage, a single storage replica should produce a similar hashrate for fairness. In PoRA mining process, the hashrate is bottlenecked by the storage Input/Output (I/O). So we expected all the miners to have similar storage I/O per unit of storage. However, a mining farm can significantly increase its I/O compared to normal miners by storing data chunks in a distributed system with in-memory filesystem. As the DDR memory has a significantly larger bandwidth compared to SSD, the mining farm can produce much more hashrate while contributing negligible to the data storage.

OG Storage prevents this behavior by imposing a large amount of data transfer from the computing stage to the loading stage to encourage miners to complete two stages on the same machine. So if the computing stage and loading stage are separated by different machines, the hashrate is bounded by the network bandwidth, which is usually smaller than the storage I/O bandwidth. In the computing stage, a random scratchpad will be generated coupled with the recall position. It has the same length as the data chunk to be loaded. Before computing the hash of the loaded data chunk, the data chunk should be mixed with the scratchpad by an XOR operation.

However, there is a dilemma between the read bandwidth and the transaction fee for submitting PoRA to the host chain. Since loading data chunks from local disks should be more efficient than downloading them via the network, the loading stage should leverage the fast SSD read speeds in sequential access to maximize the data chunk loading bandwidth. According to our benchmarks, randomly loading 256-KB data chunks reaches 80% read speed of sequential access. However, when submitting the data chunk to the host chain, a 256-KB transaction is too large and will consume a large amount of transaction fee. Reducing the size of the data chunk could mitigate this issue, but would impair the read speed.

PoRA mechanism adopts “batched data loading” to resolve this dilemma. In each iteration, a miner loads a 256-KB data chunk from storage, divides it into 4-KB data chunks, and computes the hash for each 4-KB data chunk individually. If one of the 64 outputs matches the target quality, the miner finds a valid answer. So only the 4-KB data chunk contributing to this answer is required to be submitted to the host chain.

5.2.4 Formal Definition for PoRA Mechanism

Precisely, the mining process has the following steps:

1. Register the `miner_id` on the mining contract;
2. For each mining epoch, repeat the following steps:
 - (a) Wait for the layer-1 blockchain to release a block at a given epoch height;
 - (b) Get the block hash `block_hash` of this block and the relevant context (including `merkle_root`, `data_length`, `context_digest`) at this

- time;
- (c) Compute the number of minable entries $n = \lfloor \text{data_length}/256\text{KB} \rfloor$.
 - (d) For each iteration, repeat the following steps:
 - i. Pick a random 32-byte nonce;
 - ii. Decide the mining range parameters `start_position` and `mine_length`; `mine_length` should be equal to $\min(8\text{TB}, n \cdot 256\text{KB})$;
 - iii. Compute the recall position τ and the scratchpad \vec{s} by the algorithm in figure 6;
 - iv. Load the 256-kilobyte sealed data chunk \vec{d} started from the position of $h \cdot 256\text{KB}$;
 - v. Compute $\vec{w} = \vec{d} \text{ XOR } \vec{s}$ and divide \vec{w} into 64 4-kilobyte pieces;
 - vi. For each piece \vec{v} , compute the Blake2b hash of the tuple (`miner_id`, `nonce`, `context_digest`, `start_position`, `mine_length`, \vec{v});
 - vii. If one of the Blake2b hash outputs is smaller than a target value, the miner found a legitimate PoRA solution.

Input : `miner_id`, `nonce`, `context_digest`, `start_position`, `mine_length`
Output: Recall position τ and the scratchpad \vec{s}

```

1  $h \leftarrow \text{Blake2b}(\text{miner\_id}, \text{nonce}, \text{context\_digest}, \text{start\_position}, \text{mine\_length})$ 
2 Initialize an empty array  $\vec{s}$  with 4096 64-byte elements
3 for  $i \in \{x \in \mathbb{Z} \mid 0 \leq x < 4095\}$  do
4    $h \leftarrow \text{Blake2b}(h)$ 
5    $\vec{s}[i] \leftarrow h$ 
6  $r \leftarrow \text{Keccak256}(h)$ 
7  $n \leftarrow \text{mine\_length}/256\text{KB}$ 
8  $\tau \leftarrow \text{start\_position} + r \bmod n$ 

```

Figure 6: Computing the recall position and the scratchpad

5.3 Mining Reward

OG Storage creates pricing segments every 8 GB of data chunks over the data flow. Each pricing segment is associated with an *Endowment Pool* and a *Reward Pool*. The *Endowment Pool* collects the storage endowments of all

the data chunks belonging to this pricing segment and releases a fixed ratio of balance to the *Reward Pool* every second. The rate of reward release is set at 4% per year.

The mining reward is paid to miners for providing data service. Miners receive mining reward when submit the first legitimate PoRA for a mining epoch to 0G Storage contract.

The mining reward consists of two parts:

- **Base reward:** the base reward, denoted by R_{base} , is paid for every accepted mining proof. The base reward per proof decreases over time.
- **Storage reward:** the storage reward, denoted by $R_{storage}$, is the perpetual reward from storing data. When a PoRA falls in a pricing segment, half of the balance in its *Reward Pool* is claimed as the storage reward.

The total reward for a new mining proof is thus:

$$R_{total} = R_{base} + R_{storage} \quad (2)$$

5.4 How to Balance the Number of Replicas?

Every miner in 0G Storage is required to explicitly claim the range of data entries that he maintains, and then they may only use the claimed data in their mining process. That is, a miner cannot receive mining rewards from unclaimed data, and on the other hand, the mining output is decreased if they fail to answer a challenge within their claimed range. As a result, a miner has no incentive to lie about the range of data used for mining.

Since miners truthfully expose the list of data entries that they maintain, it is easy to observe and estimate the number of replicas for every specific piece of data. Furthermore, recalling that the storage endowment is paid to miners who maintain corresponding pieces of data, miners are incentivized to store rare data and hence balance the number of replicas.

5.5 How to Incentivize Data Sharing?

In a storage network, storing data can be easily incentivized by relating the mining reward to storage size, whereas sharing is much less incentivized as the communication of data must happen outside consensus. Therefore, a

rational node would prefer to share nothing to save bandwidth and power expenses. One might suggest using a mechanism similar to the BitTorrent network, where each node maintains a local ranking of its peers and refuses to share data with leechers. However, such a mechanism is far from optimal due to the mismatch between willingness and capability. This is especially so for a storage network as a newcomer cannot quickly bootstrap (they have nothing to share), whereas a powerful node with sufficient data has much less incentive to increase its ranking score.

OG Storage introduces a built-in royalty mechanism to incentivize data-sharing behavior. The royalty mechanism allows a storage node to receive dividends from the data it shares with others when a new mining proof is mined based on the shared data. Thus, storage nodes have a clear incentive to share data with others, and newcomers get a better kick-start.

The essence of the royalty mechanism is that data providers are rewarded only after the data is shared. Data sharing is hard to incentivize mainly because off-chain sharing behavior is not publicly observable. In particular, a third-party can hardly verify whether some node A sends specific data to another node B through an off-chain channel, especially when both of the two nodes could be lying. However, the OG loyalty mechanism bypasses the problem by switching to another observable event, *i.e.* a new mining proof is generated based on the shared data. If the data sharing did not happen (e.g. the data provider ignored the request without sending data), then the requester would not be able to use that piece of data in its mining proof, and hence there is no reward for the unanswered data request. On the other hand, when the data is shared and by probability queried by the PoRA challenge, a rational miner will include it in the mining proof for a better payoff, which is positive even after deducting the dividends to the data provider.

More specifically, the royalty mechanism works as follows:

- The data requester sends his request to the OG Storage contract to create a royalty record. Each royalty record includes the range of requested data entries, the designated data provider, and the length of the royalty period.
- The data provider, when observing a new request on the host chain, decides on its own whether or not to send requested data to the re-

quester.

- When the requester submits a mining proof π , the 0G contract checks for every PoRA challenge query Q_i in π whether Q_i falls in the range of some valid royalty record. If yes, the data provider gets a certain fraction of the storage reward as a dividend.
- Expired royalty records will be removed.

As a result, data providers are incentivized to store and share data, and the expected royalty dividends are proportional to the amount of shared data. There is no incentive to forge sharing relations since the total storage reward R_i is unchanged with or without data sharing royalty.

The expiration of royalty records is introduced for three reasons: a) in case a request is ignored, the requester can resubmit it to another data provider after the previous request expires; b) a permanent royalty record does not make sense in practice, since the requester can always open a new account and hence renders the record useless; c) cleaning expired royalty records reduces the overhead of maintaining royalty-sharing relations.

Furthermore, the history of data sharing and royalty dividends can be observed onchain, which improves the effectiveness of credit ranking in the 0G Storage network.

6 Key-Value Runtime

0G Storage provides a Key-Value runtime upon the log layer. Each key-value node can access the key-value store state through the runtime interface. The key-value runtime provides the standard interface like `Put()` and `Get()`, and accepts serialized key-value pair from any application-specific structure.

During the normal execution of the key-value store node, it maintains the latest key-value state locally. It updates the value of a key through `Put()` API which composes a log entry containing the updated key-value pair and appends it to the log. The runtime constantly monitors the new log entries in the log and fetches them back to the key-value node, updating the local key-value state according to the log entry contents. In this sense, multiple key-value store nodes essentially synchronize with each other through the shared decentralized log. A user-defined function will be used to deserialize the

raw content in the log entry to the application-specific key-value structure. Applications can use `Get()` API to access the latest value of any given key. To improve the efficiency of the updates for small key-value pairs, the `Put()` allows batched updates with multiple key-value pairs at once. Figure 7 illustrates the architecture of the decentralized key-value store. To manage the access control, the ownership information of each key can also be maintained in the log entries. All the honest key-value nodes follow the same update rule for the keys based on the ownership to achieve state consistency.

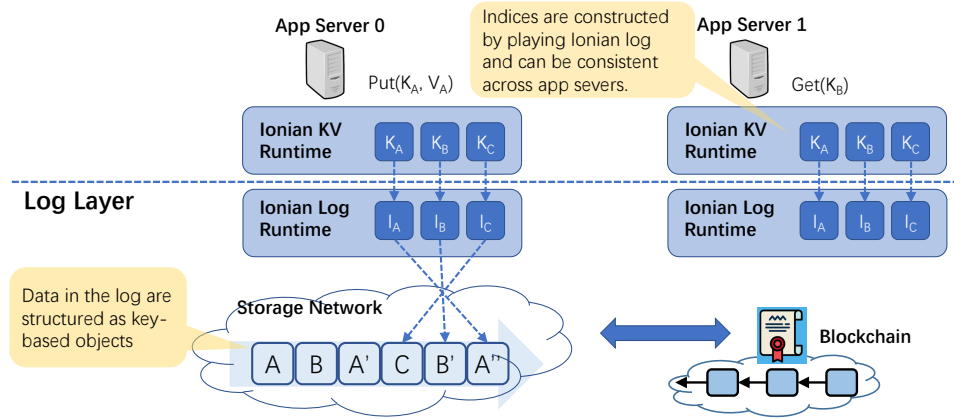


Figure 7: Decentralized Key-Value Store

When a new key-value node just joins the network, it connects to the log layer and plays the log entries from head to tail to construct the latest state of the key-value store. During the log entry playing, an application-specific key-value node can skip irrelevant log entries that do not contain the stream IDs that it cares.

7 Transactional Processing on Key-Value Store

OG Storage employs concurrency control in the key-value runtime to support transactional processing for concurrent operations on multiple keys. This concurrency control mechanism is optimistic and hinges on the overall ordering of log entries which is enforced by the underlying log layer. Figure 8 illustrates the mechanism.

When an application server linking with the OG Storage key-value run-

time starts a transaction using `BeginTx()` interface, it notifies the runtime that the transaction will work on the current state snapshot constructed by playing the log to the current tail. The further key-value operations before the invocation of `EndTx()` update the key-values locally in the server without exposing the updates to the log. When `EndTx()` is invoked, the runtime composes a *commit record* containing the log position the transaction starts from and the read-write set of the transaction. This commit record is then appended to the log.

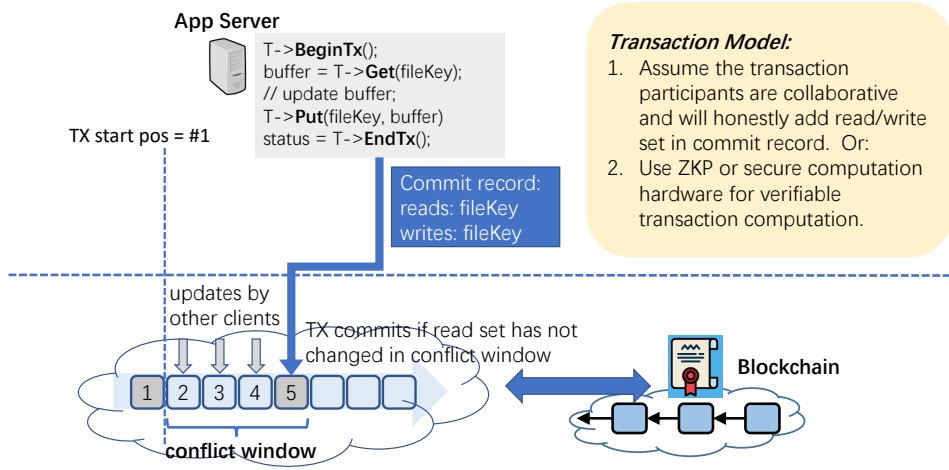


Figure 8: Transactional Processing on 0G Storage KV Store

When an application server with the key-value runtime encounters the commit record while playing the log, it identifies a *conflict window* consisting of all the log entries between the start log position of the transaction and the position of the commit record. The log entries in the conflict window therefore contain the key-value operations concurrent with the transaction submitting the commit record. The runtime further detects whether these concurrent operations contain the updates on the keys that belong to the read set of the transaction. If yes, the transaction is aborted. Otherwise, it is committed successfully.

This transaction model assumes that the transaction participants are collaborative and will honestly compose the commit record with the correct content. Although this assumption in a decentralized environment is too strong, it is still achievable for specific applications. For example, for an

application like Google Docs, a user normally shares access with others who are trusted. In case this assumption cannot hold, the code of the transaction can be stored in OG log and some mechanism of verifiable computation like zero-knowledge proof or hardware with trust execution environment (TEE) can be employed by the transaction executors to detect the validity of the commit record.

8 Roadmap

TBA.

References

- [1] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities, 2019.
- [2] Ethereum White Paper. <https://ethereum.org/en/whitepaper/>, 2020.
- [3] ORA-OpML. <https://docs.ora.io/doc>.
- [4] Optimism. <https://www.optimism.io/>.
- [5] Celestia. <https://celestia.org/>.
- [6] EigenDA. <https://www.blog.eigenlayer.xyz/tag/eigenda/>.
- [7] EigenLayer. <https://www.eigenlayer.xyz/>.
- [8] BabylonChain. <https://babylonchain.io/>.