Goliath: a horizontally-scalable storage system for crypto.

Liam Zebedee

Abstract

The need for scalable storage systems in crypto infrastructure remains largely unaddressed.

Goliath is a novel horizontally-scalable storage network for crypto inspired by Google's File System (GFS), adapted for decentralized environments. Using Goliath, users can deploy clusters for storing blockchain state and history, decentralized AI models and their outputs, and confidential data stores. Clusters are read-write, permissioned by public keys and ZK proofs, and can grow in size dynamically.

1 Introduction.

1.1 Unbundling Ethereum.

The Ethereum blockchain was invented in 2015, a monolithic bundling of many inventions: Indexes (logs), execution (EVM), sequencing (mempool), consensus (PoW + GHOST), data availability (coinbase). Since then, every aspect of this stack is being torn out and spun into its own protocol.

Overall, Ethereum represents a maturing ecosystem, wherein modules have diverged into more specialized subcomponents. For example, the Lyra Protocol began as a DeFi protocol on Ethereum L1, and then later spun out as an optimistic rollup L2, enabling them to scale with cheaper transcation fees, and then replaced their data availability with Celestia for even more savings passed on to the user.

Curiously, there is an aspect missing from this stack: storage. Every Ethereum node, every Ethereum L2 sequencer, every blockchain requires storage for its apps - both for the raw state of the database (e.g. ERC20 balances) and for the history of the chain itself (what is used to sync). And storage so far has remained the exact same - operators must manually manage and allocate storage via attaching HDD's.

Component	Ethereum (2015)	Ethereum protocols (2024)
Indexes	Logs	Subgraphs, Shadow, Custom indexers
Execution	EVM	Parallel EVM rollups, WebAssembly (Arbitrum AnyTrust), custom ZK-proven VM's (Starknet, Scroll)
Sequencing	Mempool	Decentralized sequencers, threshold-encrypted mempools
Data Availability	Coinbase	Celestia, EigenDA, Danksharding
Security models	Replicated state machines	Optimistic rollups, Validity rollups / ZK-proofs
Economic/validator distribution	Ethereum social layer	Eigenlayer, Symbiotic

Table 1.1: Unbundling the Ethereum monolith.

We have been tracking the development of Ethereum since 2015, and have been interested in innovating in this space. This whitepaper serves as a research exploration into creating something new - a cryptonetwork for scalable storage.

1.2 A storage cryptonetwork.

Crypto infrastructure demands a scalable storage system in many areas of the stack. Syncing a blockchain is extremely expensive and slow, indexing and downloading from many peers from scratch for every node. After a blockchain synchronises, every node must keep a full copy of state on their storage device, ruling out devices with low storage capacities from participating. There is a huge demand for decentralized storage systems that exist independently of blockchains - for example for indexers, for hosting decentralized AI models, confidential data stores [4].

While there have been products, no solution has really succeeded as a modular component in the same way Celestia has for data availability. BitTorrent is designed primarily for *data sharing*, cannot be used to write data, and is not a cryptonetwork where you can pay for storage as $ETH pays for blockspace. Filecoin has succeeded in building an economic model, but its product has failed to capture the market. There are many potential reasons for Filecoin's lack of adoption - the API to read/write storage is cumbersome, it cannot be done from a smart contract, it requires asynchronously interacting with an orderbook (storage deals) rather than in Ethereum where users purchase blockspace synchronously from a proto-AMM (the progressive gas auction), the product does not provide guarantees of read/write speeds. By all measures though, Filecoin has failed to capture the market - many prominent crypto companies use IPFS to verifiably host data, such as Zora - but instead of paying to put the data on Filecoin, they run their own IPFS nodes.

1.3 Conceptual design.

Our design for this system comes from Google's distributed systems, namely GFS [5] and Bigtable [3].

Key contributions . We rebuild the Google File System within a byzantine trust model [6] as a cryptonetwork. Instead of Chubby [2] and Paxos, we use a blockchain on Tendermint [1]. The master server is replaced by the blockchain node, which performs deterministic and non-deterministic roles (such as checking liveness via heartbeats). Chunkservers, which provide the storage to clients, perform the same function as in GFS - though are regularly issued data availability challenges in the style of Celestia and Filecoin. The total storage capacity of the network is tracked by the master node, and sold via a custom-built automated market maker. Users of Goliath storage clusters interact via ECDSA wallets, and are able to read and write files and directories in an S3-compatible interface.

2 Design.

Our intuition for building Goliath is through reading the Google distributed systems literature, and noting the analogues.

Firstly, we will briefly explain Google File System. Then we will give an overview of the components and their analogues in the crypto space. Then we will sketch the design for the Goliath system.

2.1 Google File System.

GFS splits files into fixed-size workloads called chunks, and assigns the workload of serving reads/writes to chunkservers. GFS's key insight is that the allocation and management of chunks is a process that can fit on a single server, which is called the master.

In order to elect a server as the master process, GFS uses Chubby. Chubby is a strongly-consistent highly-available lock file service developed by Google. Chubby allows application developers to store small files locked to a single network process via the Paxos consensus algorithm. In GFS, a master server is identified by its lock on a Chubby file. If the master process crashes, the lock is released, and another server can take the lock and assume the role of the master. The orchestration of such servers is done by a separate system, Borg.

The analogue for these Google systems is clear. The Paxos consensus algorithm can be replaced by a byzantine fault tolerant algorithm such as Tendermint. The Chubby lock file service, a small stateful program, can be replaced by a smart contract on a blockchain. Finally, in order to verify the chunkservers are indeed storing data - we can introduce data availability sampling, whereby nodes provide a random proof that they are hosting a subset of data.

Google	**Crypto analogue**
Paxos consensus	Tendermint consensus
Chubby lock file service	Smart contract
GFS master	Blockchain node
GFS chunkserver	Chunkserver with data availability proofs

Table 2.1: Google/crypto distributed systems analogues.

Non-deterministic computation. One interesting difference to Ethereum L2's is that Goliath makes use of *non-deterministic* computation affordances in Cosmos blockchains. Namely, the master process is a Tendermint blockchain, though unlike Ethereum smart contracts, we perform out-of-protocol functions that are usually reserved for the role of *keepers* within a crypto context - an example of this is connecting to chunkservers, performing regular heartbeats to verify their liveness.

2.2 Overview.

1. **Goliath cluster**. A Goliath cluster consists of a Tendermint blockchain representing the master process, and a set of worker nodes referred to as chunkservers, which perform the data storage.
2. **Master process**. The master process is implemented in Go, as a Tendermint blockchain. It stores the file index (file paths, replication factor, chunks, permissions), chunk servers (address, chunk sets, heartbeat, capacities), read/write leases (locks), token balances (accounts). It runs the functions to manage chunkservers (indexing their chunks, heartbeats to check status, assigning a primary, data availability checks), allocating chunks to chunkservers (assignment, replication), pricing and selling storage (automated market maker), and allocating leases for file writes to users.
3. **Chunkserver process**. The chunk server process is implemented in Go, as a standalone process. It stores chunks. It follows commands given by the master blockchain, including replicating chunks, serving reads to users, answering data availability checks. It verifies leases issued by the master and then authorises writes. There is one chunkserver deemed the primary which determines write order, the rest follow.

4. **Client library**. The client library is used by users to read and write data to a Goliath cluster. Control flow begins at the master, where users acquire a write lease which ensures writes are consistent. Users then interact directly with the chunkserver primary to upload chunk data, which is then replicated on secondaries, and finally committed. Chunkservers are cached inside the client library to reduce load on the master.

References

[1] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on bft consensus, 2019. URL https://arxiv.org/abs/1807.04938.

[2] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006. URL http://labs.google.com/papers/bigtable.html.

[4] F. Collective. Suave specifications - confidential data store. URL https://github.com/flashbots/suave-specs/blob/main/specs/rigil/confidential-data-store.md.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450.

[6] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. URL http://dblp.uni-trier.de/db/journals/toplas/toplas4.html#LamportSP82.