FileDAG: A Multi-Version Decentralized Storage Network Built on DAG-based Blockchain

Hechuan Guo, Minghui Xu, Jiahao Zhang, Chunchi Liu, Dongxiao Yu, Schahram Dustdar, Xiuzhen Cheng

Abstract—Decentralized Storage Networks (DSNs) can gather storage resources from mutually untrusted providers and form worldwide decentralized file systems. Compared to traditional storage networks, DSNs are built on top of blockchains, which can incentivize service providers and ensure strong security. However, existing DSNs face two major challenges. First, deduplication can only be achieved at the directory-level. Missing file-level deduplication leads to unavoidable extra storage and bandwidth cost. Second, current DSNs realize file indexing by storing extra metadata while blockchain ledgers are not fully exploited. To overcome these problems, we propose FileDAG, a DSN built on DAG-based blockchain to support file-level deduplication in storing multi-versioned files. When updating files, we adopt an increment generation method to calculate and store only the increments instead of the entire updated files. Besides, we introduce a two-layer DAG-based blockchain ledger, by which FileDAG can provide flexible and storage-saving file indexing by directly using the blockchain database without incurring extra storage overhead. We implement FileDAG and evaluate its performance with extensive experiments. The results demonstrate that FileDAG outperforms the state-of-the-art industrial DSNs considering storage cost and latency.

Index Terms—Decentralized storage networks, DAG-based blockchain, deduplication, file indexing

arXiv:2212.09096v1 [cs.DC] 18 Dec 2022

1 Introduction

Blockchain technology is allowing for decentralized computing by creating decentralized trust. This has led to the development of various trustless applications, such as decentralized learning [1], trusted IoT data collection [2], and blockchain-based cloud services [3]. To improve the performance and efficiency of decentralized computing, decentralized storage networks have been created to decrease the amount of redundant storage needed for blockchain. Decentralized Storage Networking is an emerging technology that can aggregate free storage spaces offered by independent storage providers and self-coordinate to provide data storage and retrieval services. Compared to traditional storage networks [4], [5], a decentralized storage network (DSN) is operated on a blockchain system, which works as an incentive layer. Blockchain rewards miners who provide reliable storage to clients, and thus enables an open manageable storage market. Besides, blockchain can act as a state machine replication protocol to ensure the consistency of file storage against Byzantine nodes. Leveraging blockchain technologies, DSNs (e.g., Filecoin [6], Storj [7], Sia [8], Swarm [9]) provide worldwide, robust and secure storage services among mutually untrusted users. Filecoin, as the most popular DSN, was built on top of InterPlanetary File System (IPFS) and adopts a novel proof-of-replication method proving that data is correctly stored. As storage infrastructures, DSNs have demonstrated their advantages in applications such as Web 3.0 [10], data sharing [11], and content delivery [12]. However, current DSN schemes overlook the following two problems, which significantly affect their performance.

[P1] Deduplication in multi-versioned files. Supporting multi-versioned file storage is necessary in DSNs since files are usually dynamically changed or edited and users need to query different versions of a file from time to time. However, files on current DSNs are not editable. Users have to upload all versions of a file, resulting in high redundancy. Even though some DSNs have made efforts in supporting directory-level deduplication, they cannot avoid fine-grained file-level redundancy. For example, Filecoin realizes directory-level deduplication using Merkle DAG [13], in which objects including files, file chunks, and directories are organized into a Merkle DAG based on their nested relationships, to remove duplicated objects among different directories; but redundencies among different file versions are still unavoidable. Missing file-level deduplication causes the waste of storage and bandwidth. Nevertheless, achieving file-level deduplication is challenging. Due to encryption and obfuscation applied on files, correlation among different versions is implicit in current DSNs, making it very hard to find duplicated contents and establish relationships among multiple versions.

[P2] File indexing with blockchain. Traditional version control systems commonly adopt a DAG-based version graph [14] to describe the relationships of multiple versions (also called derivative relationships) and help establish file indexing. However, such a graph should be maintained by a centralized server, e.g., Github [15]. In current DSNs,

- *H. Guo, M. Xu, J. Zhang, D. Yu and X. Cheng are with the School of Computer and Science and Technology, Shandong University. Email: {ghc, zjh}@mail.sdu.edu.cn, {mhxu, dxyu, xzcheng}@sdu.edu.cn*
- *C. Liu was with the Department of Computer Science, The George Washington University and now with Ernst & Young. E-mail: liuchunchi@gwu.edu*
- *S. Dustdar is with the Research Division of Distributed Systems, TU Wien. Email: dustdar@dsg.tuwien.ac.at*

Corresponding author: Minghui Xu.

a centralized server is not available, and the blockchain database simply stores file information in serialized transactions regardless of the derivative relationships. Therefore, it is unavoidable for each user to locally maintain an isolated database for additional metadata (e.g., a version graph) to ease its file manipulations such as create, version query, modification, merge, and fork. Furthermore, such a deficiency prevents a blockchain from serving many data-intensive applications since it takes a large amount of storage but cannot directly answer file queries in many cases, making itself like a "burden".

To address these problems, we propose FileDAG, a DSN system built on top of a DAG-based blockchain. FileDAG makes use of an increment-based storage mechanism to realize file-level deduplication in storing multi-versioned files. We apply increment generation algorithms to calculate the increment, an edit script that can transform a file from its previous version to its current version. Storing increments achieves fine-grained deduplication at file-level and saves storage space. Besides, we adopt a two-layer DAG-based blockchain ledger, which organizes transactions according to their derivative relationships. Facilitated with this ledger, one can manipulate files without establishing additional databases. Moreover, DAG-based ledgers can provide higher concurrency compared to chain-based ones [16], [17], making FileDAG being able to handle simultaneous queries. With these design considerations, FileDAG achieves low storage cost, high system throughput, and efficient file indexing.

To validate the performance of FileDAG, we build a full-fledged FileDAG over Filecoin, by implementing the increment mechanism and the two-layer ledger mentioned above. Such an implementation ensures that FileDAG not only inherits all the nice features of Filecoin but also extends Filecoin's functionality to include effective file-level deduplication and efficient file indexing. FileDAG is a practical system possessing industrial-grade performance, as is Filecoin. To broaden the application of and welcome examinations on FileDAG, we open-source our designs at GitHub (the two layer ledger: https://github.com/zhuaiballl/DAG-Rider; increment module: https://github.com/zhuaiballl/dyaic).

Contributions. Compared to the existing works, our unique contributions can be summarized as follows:

1) To our best knowledge, FileDAG is the first DSN that supports file-level deduplication for multi-versioned files. We introduce an increment generation method to calculate and store the increment between two neighboring versions rather than storing the entire new version. This significantly reduces the storage cost and bandwidth usage caused by dynamical file changes.
2) To support file indexing, FileDAG adopts a two-layer DAG-based blockchain ledger. The lower layer supports operations including create, update, merge and fork while the upper layer ensures ledger consistency. This design integrates version graphs with a DAG-based ledger, thereby saving extra storage space for file indexing.
3) Finally, we provide a practical full-fledged implementation of FileDAG and evaluate its performance with extensive experiments. The results demonstrate that FileDAG outperforms the state-of-the-art DSNs considering storage cost as well as the latency of put and get operations.

Organization of the paper. The rest of this paper is organized as follows. Section 2 summarizes related works and presents preliminary knowledge. Section 3 details our FileDAG design and demonstrates how it works. Key properties and performance evaluation results of FileDAGE are respectively reported in Section 4 and Section 5. Finally, we summarize this paper in Section 6 and discuss our future research.

2 Related Work and Preliminaries

2.1 Related Work

2.1.1 Decentralized Storage Network

Filecoin [6], developed by Protocol Labs, is a DSN built on top of IPFS [18]. It proposes Expected Consensus to adjust the winning probability of a miner based on the quantity and quality of its provided storage. Filecoin generates a hash-based content identifier (CID) for each file object (a file, a file chunk, or a directory), and allows users to reuse existing file objects for avoiding duplicatively storing them. Besides, CIDs form a Merkle DAG depicting the nested relationship of the file objects. To realize block concurrency, Filecoin introduces tipset, which allows multiple blocks to be confirmed at the same block height. Storj [7] and Swarm [9] were developed based on Ethereum [19]. They make use of Proof-of-Stake consensus[1] and a chain-based ledger that doesn't support concurrency of blocks. Storj employs Object keys as globally unique identifiers of its file objects while Swarm generates addresses as identifiers for file chunks. These two DSN systems both achieve directory-level deduplication. Sia [8] adopts PoW as its consensus protocol. It builds a Merkle tree for each file and takes the Merkle root hash as the identifier of the file, thus supporting directory-level deduplication. The ledger structure of Sia is a chain, thus it cannot process blocks concurrently. Besides, Sia employs the Threefish [20] algorithm to encrypt files, making it difficult to support version indexing.

2.1.2 File Indexing

File indexing is the process of mapping files with identifiers that can be efficiently searched. Centralized storage systems employ extra databases to record identifiers that are mapped to the locations of the corresponding files [21]. Traditional distributed storage networks typically use distributed hash tables to realize file indexing [4], [22], [23]. Decentralized storage networks, i.e., DSNs, use content addressing technology based on distributed hash tables for file indexing. Nevertheless, current methods in DSNs are not sufficiently effective as the complete derivative relationships are hardly retained. For example, IPLD [18] is a data model adopted in Filecoin to describe a file or a directory as an aggregate of components linked together. With IPLD, each file is mapped to a unique hash-based identifier, and the identifier of a directory is a hash of the directory contents combined with pointers to the files. By this way, IPLD links files and

1. Since September 15th, 2022, Ethereum has switched its consensus protocol from Proof-of-Work to Proof-of-Stake

directories together for file indexing. Adding, removing, or changing a file under a directory result in a different identifier of the directory, and the derivative relationship between two versions of the directory can be inferred because the two identifiers carry the pointers to the files that stay unchanged. But unfortunately IPLD fails to depict the derivative relationships among different versions of a file. Additionally, to achieve verifiability and immutability, files in DSNs are always encrypted and then made public; thus their metadata is unaccessible without a valid secret key, making file indexing a challenging problem. Based on the above analysis, one can see that the current file indexing approaches are immature and inefficient.

File provenance requires to track the derivation history of a file based on file indexing. Muniswamy-Reddy *et al.* [24] designed a storage system that can automatically collect and maintain provenance data. They claimed that provenance data should be maintained separately to serve different purposes. Provchain [25] embeds the provenance data into blockchain transactions to improve efficiency and avoid additional storage cost. This incentives us to make blockchain undertake more responsibility in file indexing.

2.1.3 Summary

TABLE 1: Comparison of FileDAG with Existing DSNs

	Consensus Algorithm	Ledger	On-Chain DR	Deduplication Level
Filecoin [6]	Expected consensus	DAG (tipset)	✗	Directory
Storj [7]	PoW	Chain	✗	Directory
Sia [8]	PoW	Chain	✗	Directory
Swarm [9]	PoW	Chain	✗	Directory
FileDAG	DAG-Rider†	DAG	✓	File

DR Derivative Relationship
† Modified

A summary on the major adopted technologies and properties of FileDAG and existing DSNs is reported in Table 1. One can see that current DSNs (e.g. [6]–[9]) only achieve directory-level deduplication, which means that only files can be reused but the common contents shared by different versions of a file are still stored redundantly. Missing file-level deduplication leads to the waste of storage and bandwidth. Additionally, Storj and Swarm built on Ethereum adopt a chain-based ledger, which stores transactions regardless of their derivative relationships. Sia doesn't consider storing derivative relationship between files either. Filecoin packs multiple blocks in a tipset and still ignores on-chain derivative relationships. Lacking a depiction on the complete derivative relationships among files render these systems fail to provide effective file indexing.

2.2 Preliminaries

In this subsection, we provide the preliminary knowledge that are needed by our FileDAG design.

Decentralized storage network (DSN). DSNs aggregate storage offered by multiple independent storage providers and self-coordinate to provide reliable and secure global data storage and retrieval services to clients without relying on any trusted third party. Generally speaking, the workflow of a DSN consists of two phases: put and get. Users put their files into the storage network and also get files with valid access keys from the network. A DSN must guarantee data integrity, retrievability and fault tolerance. We explain two techniques heavily used in FileDAG, namely content identifier (CID) and Proof-of-Storage (PoS). CID, as a fingerprint, is a hash-based unique identifier that maps to a data chunk. In FileDAG, a client can generate CIDs for each original file or increment. PoS helps miners prove that they have stored files physically. In Filecoin, a miner has to periodically generate proofs to demonstrate that files are indeed locally stored on hardware, which mitigates Sybil attacks.

DAG-based blockchain. A blockchain is a decentralized tamper-proof append-only ledger. Nodes in a blockchain network achieve consensus on the ledger using a consensus algorithm. According to the ledger structure, blockchains can be categorized as chain-based or DAG-based. For a chain-based ledger, transactions are packed into blocks. Each block is hash-chained to its previous block to ensure consistency and persistence [26]. As there can only be one block at a block height, chain-based blockchains have weak concurrency. Bitcoin-NG [27] intends to improve concurrency by adding micro blocks alongside a main chain. However, this method does not fundamentally improve concurrency. Therefore, DAG-based blockchains emerge [16], [28], [29]. In a DAG-based blockchain, each block (or transaction) can point to multiple previous blocks and form a directed acyclic graph (DAG). Filecoin makes use of tipset to increase network throughput, where a tipset is a set of blocks, and the blockchain in Filecoin is a chain of tipsets. Blocks in a tipset can point at multiple blocks in the previous tipset. As a result, blocks in Filecoin form a DAG. But tipset is not flexible enough to support file indexing; therefore we propose a two-layer DAG-based blockchain in FileDAG to address this issue.

3 FileDAG Design

In this section, we begin with the design objectives and overview of FileDAG and then describe its design details.

3.1 Design Objectives and Strawman

Design Objectives. We design FileDAG following three objectives: (1) Consistency. Honest nodes should agree on the same view of the blockchain ledger and the same set of proofs of storage. Deals of storage should be irreversible. (2) Deduplication. Files stored on a DSN can share common components, especially in a multi-version file system. FileDAG should use efficient deduplication methods to save storage space. (3) Fast put & get. The design of FileDAG should consider both bandwidth usage and latency; any mechanism that can help to save storage space should not bring too much extra latency. The overall latency of putting and getting a file in FileDAG should be low despite spending time on the increment generation.

Strawman. Here we provide a strawman as shown in Fig. 1 to illustrate the whole picture of FileDAG. There are two entities in FileDAG, client and miner. Clients pay tokens

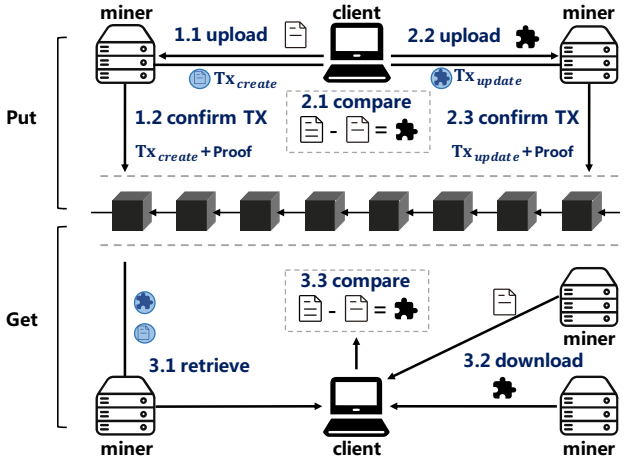


Fig. 1: A strawman design of FileDAG

to use storage, while miners earn tokens by providing services. Miners pledge storage to the FileDAG network to provide storage and retrieve services by helping clients search information on a blockchain. All miners maintain the blockchain ledger of FileDAG.

The workflow can be divided into two phases, put and get. In the put phase, a client can either create an original file or update an existing one on the FileDAG network. An original file should be uploaded to a miner [Step 1.1]. The miner then generates a transaction $\mathsf{TX}_{\mathsf{create}}$ for the file and broadcasts the transaction to the blockchain network [Step 1.2]. To update a file, the client first compares the new version to the previous version to get the increment [Step 2.1]; then the increment is sent to a miner who can issue the corresponding $\mathsf{TX}_{\mathsf{update}}$ (or $\mathsf{TX}_{\mathsf{merge}}$, $\mathsf{TX}_{\mathsf{fork}}$). Note that only when transactions are confirmed on the blockchain can the put phase succeed. In the get phase, a client first sends a retrieval request [Step 3.1], then download the original file and the increments from the file holders to obtain a specific version [Step 3.2]; finally, the client assemble all fragments to recover the file [Step 3.3].

In the following subsections, we detail the cores of FileDAG, including the increment generation, the two-layer DAG-based ledger, and the file recovery components. Note that our elaboration on increment generation focuses on the storage of multi-versioned files; but the idea is applicable to the more general case where a client specifies the relationship between two files, which is common in applications such as recreation of digital arts and quotes of contents. Table 2 lists frequently used symbols to facilitate our presentation.

3.2 Increment Generation

Increment mechanism has been widely adopted in cloud computing to shorten backup windows and save storage [30]. As we have discussed in our strawman design, FileDAG updates files by uploading increments instead of an entire new file. We propose an increment generation method based on our insight that files on DSNs are not simply static but changes over time. Such dynamicity can be found everywhere especially when storing codebases, medical records, mobile applications etc. Neighboring versions

TABLE 2: Summary of Symbols

Symbol	Description
G	DAG-based ledger of FileDAG
V	the set of vertices in G
E_l	the set of edges in the lower layer of G
E_u	the set of edges in the upper layer of G
v	version of a multi-versioned file
Δ	increment
TX	transaction
N	network size
f	the maximum number of Byzantine fault nodes to tolerate
CID_v	Content ID of v
REV	revision operation
ADD	addition operation
v_t	the t-th version of a multi-versioned file
τ_t	the type of operation that outputs v_t
S_t	size of v_t
I_t	size of the increment between v_t and v_{t-1}
$E(\cdot)$	expectation operator
C	expected storage cost without increment-based storage
C'	expected storage cost of increment-based storage
n	the number of versions

of a file usually share a large amount of duplicate contents. Our increment generation method intends to identify such contents, which later will be used for file recovery.

In concrete, FileDAG adopts patch algorithms to generate increments for multi-versioned files. To achieve a better performance, we adaptively use two patch algorithms, i.e., Myers [31] and BSDiff [32], to process text files and non-text files (binary files), respectively, rather than rely on one algorithm. Assume we have two files, an old one A (of size $|A|$) and a new one B (of size $|B|$). Both Git and diff commands in Linux use Myers, which takes $O(|A|+|B|+D^2)$ expected-time under a basic stochastic model [31], where D is the size of the minimum edit script between them. The Myers algorithm can quickly generate patches for text files, but cannot efficiently handle binary files. When forcing Myers to treat binary files as text files, the algorithm runs slowly and the complexity of generating an increment becomes $O(|A||B|)$. To process non-text files, we choose BSDiff which runs in $O((|A|+|B|)\log|A|)$ time. Besides, BSDiff has been widely used to generate patch files for mobile applications, which proves its effectiveness. In our implementation, FileDAG adaptively switches between Myers and BSDiff. It feeds the files into an increment module (see Fig. 5), which selects Myers for text files and BSDiff for non-text files to generate increments. In addition, we employ a small trick in which if $|\Delta_{AB}| > |B|$, FileDAG takes B as a new original file instead of storing the increment Δ_{AB}.

To update a file, the client sends the increment to a miner who responds with a CID. Then the miner generates a proof for this increment following the Proof-of-Storage protocol. In our implementation, we adopt the same PoS protocol as Filecoin since FileDAG does not focus on improving this process.

3.3 Two-Layer DAG-based Ledger

FileDAG uses a two-layer DAG-based ledger denoted as $G = (V, E_l, E_u)$. Both layers share the same set of vertices V. E_l and E_u are respectively the sets of edges in the lower layer and the upper layer. Each vertex in the ledger represents a transaction (a file version). Edges in the lower

layer are used to describe derivative relationships between neighboring versions while the upper layer adds more edges to ensure consistency.

Lower Layer E_l. Fig. 2 demonstrates an example lower layer E_l, in which each vertex (i.e., a transaction) also corresponds to a specific file version since it contains the CID of an original file or an increment. Each edge represents the derivative relationship between two vertices.

We allow four different types of transactions to describe operations launched by a client, including create, update, merge and fork, where the latter three depict the derivative relationships among different file versions. Correspondingly, an edge in E_l represents an update, or a merge, or a fork operation. A create transaction is used to record a new original file. When a client creates a new file, it transfers the file to a miner, then constructs a create transaction $\mathsf{TX}_{\mathsf{create}} \leftarrow \langle \mathsf{CREATE}, \mathsf{CID}_{v_0} \rangle$ and broadcasts it to blockchain, where CID_{v_0} is the identifier generated for the file. To verify that a transaction is sent by a client, each transaction should be correctly signed by the client's secret key. For convenience and clearance, we omit signatures when describing transactions in the rest of this paper. When a client intends to put an increment to FileDAG, it sends an update transaction $\mathsf{TX}_{\mathsf{update}} \leftarrow \langle \mathsf{UPDATE}, v, \mathsf{CID}_{\Delta} \rangle$, where v is the version that the update follows, and CID_{Δ} is the identifier of the increment.

A merge transaction $\mathsf{TX}_{\mathsf{merge}} \leftarrow \langle \mathsf{MERGE}, v, v' \rangle$ combines two version branches v and v'. A merge operation does not require adding new information therefore it does not generate increments. We only allow FileDAG to merge two versions at a time because merging multiple versions at once incurs a large complexity of addressing content conflict; but FileDAG can merge multiple versions by calling the merge operation multiple times. A fork transaction $\mathsf{TX}_{\mathsf{fork}} \leftarrow \langle \mathsf{FORK}, v, V' \rangle$ can fork a version v to get a set of new versions denoted by V'. Each new version contains an empty increment but is assigned a new CID. With fork operations, users can create and work on their own branches without the need of making new copies.

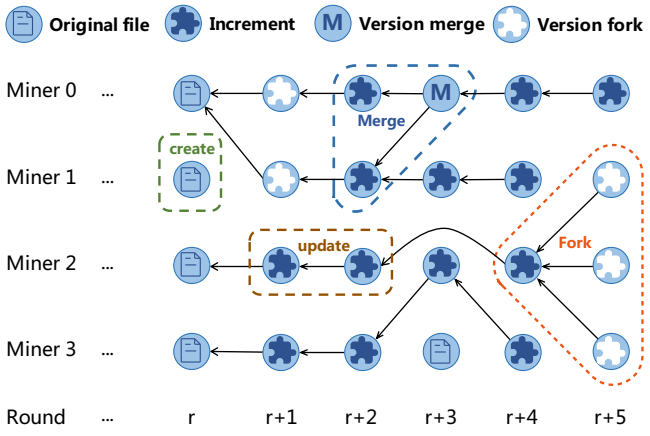


Fig. 2: Lower layer E_l and the four types of transactions

Upper Layer E_u. The DAG ledger formed by E_l and V has no consistency guarantee as it might be disconnected, making a miner unaware of a newly added transaction if the transaction is not linked to the ledger component stored by the minor. For example in Fig. 2, Miner 0 does not know the newly added transaction created by Miner 3 at round $r+3$. This implies that honest miners may have different views of the ledger and fail to output the same result for a query, thus breaking the ledger's consistency property. To overcome such a problem, we add extra edges as shown in Fig. 3 (the dotted arrows) to form the upper layer edge set E_u. More specifically, we modify the ledger construction algorithm in DAG-Rider [33] to construct E_u. In DAG-Rider, each vertex is associated with a round number (see Fig. 3). Each miner broadcasts one transaction (creating one vertex) per round and each vertex references at least $2f+1$ vertices in the previous round, where f is the maximum number of Byzantine nodes to tolerate. That is, to advance to round $r+1$, a miner first needs to identify $2f+1$ vertices constructed by different miners at round r. Such a DAG construction is proved to achieve Byzantine atomic broadcast [33], which possesses a strong consistency guarantee. Note that one can adopt other approaches to construct E_u, as long as the ledger formed by all edges in $\mathrm{E}_l \cup \mathrm{E}_u$ realizes Byzantine atomic broadcast. More details about the Byzantine atomic broadcast will be discussed in Section 4.

The whole procedure of constructing our two-layer DAG-based ledger can be summarized as follows. At any round, an incoming transaction first points to those confirmed in the previous rounds, following the derivative relations (update, merge, or fork) to contribute edges to E_l; then we follow appropriate rules to select a number of other confirmed transactions and link the incoming transaction to them to construct edges for E_u. The DAG ledger formed by $\mathrm{E}_l \cup \mathrm{E}_u$ has strong consistency guarantee (see Section 4).

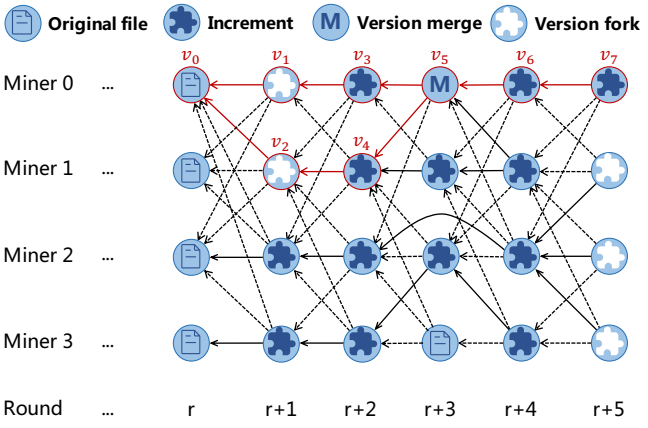


Fig. 3: Upper layer E_u

3.4 File Recovery

Based on the lower layer of the DAG-based ledger, a miner can easily gather file fragments needed to recover a file. Considering that files are stored as increments for different versions, we propose Algorithm 1 to retrieve versions and recover the queried file. This algorithm consists of two functions, namely Retrieve() and Recover().

First, the miner runs Retrieve(v) (line 2-11) to obtain the versions of all fragments needed to recover the file of version v. Retrieve(v) starts breadth first search (BFS) at version v traversing the edges in E_l, and stops iteration when

meets the version corresponding to a complete file. Using BFS rather than DFS (depth first search) can ensure that all the increments are marked in a reverse topological order without a sort procedure whose time complexity might be superlinear. All the traversed nodes are placed in the array versions. When BFS stops, Retrieve(v) reverses the order of the array versions and then returns it to the client (line 12). After receiving versions, the client runs Recover(versions) to download the original file and the increments in versions, patch the increments to the original file following the order in versions, and finally output the requested file (line 15-20). The Patch algorithm takes either Myers or BSDiff, depending on whether the file is a text or not, as explained in section 3.2. These two algorithms both have time complexity of $O(|A|+D)$, where $|A|$ is the size of the file being patched and D is the size of the increment. Thus, the overall time complexity of file recovery is linear to the total size of the original file and the increments.

Algorithm 1: File Recovery

```
1  //Find vertices of a requested version on a ledger
2  Function Retrieve(v)
3  versions[] ← an empty array
4  Q ← an empty queue
5  Q.Push(v) //an empty queue to store versions
6  while Q is not empty do
7      temp ← Q.Dequeue()
8      versions.Append(temp)
9      if temp.type ≠ origin then
10         for each pre ∈ temp.previousVersions do
11             Q.Push(pre)
12 Reverse and then return versions
13 //Recover a file using file fragments
14 Function Recover(versions)
15 Download all file fragments as Data
16 v_0 = versions[0]
17 file ← Data[v_0] //initialize file to an original one
18 for each v ∈ versions with v.type=increment do
19     Patch(file, Data[v])
20 return file
```

3.5 FileDAG Workflow

To end this section, we provide the workflow of FileDAG, which consists of five major steps including Create, Update, Retrieve, Download and Recover, as illustrated in Fig. 4.

Create. When creating an original file v_0 in the FileDAG network, a client first calculates CID_{v_0} as the fingerprint of v_0. The client then sends a message containing CID_{v_0} to a miner that might later provide storage services. If the miner is willing to store the file, it starts synchronizing v_0 with the client. After synchronization, the client signs and sends a create transaction $\langle \mathsf{CREATE}, \mathsf{CID}_{v_0} \rangle$ to the blockchain ledger. Then the miner generates a proof-of-storage for v_0 and settle down the received transaction.

Update. The main difference between creating and updating a file is that updating a file needs to store an increment rather than the entire complete file. Suppose we have a version v and a new version v'. The increment generation method provides the client with an increment denoted by Δ. Then the client calculates the CID of Δ and sends Δ to a storage miner who is responsible for generating a proof and settling down the transaction. Recall that when updating a file, a client can issue three types of transactions, namely update, merge and fork.

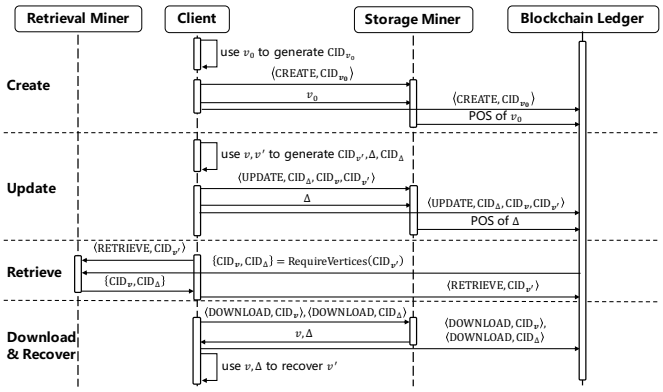


Fig. 4: Protocol sequence diagram of FileDAG.

Retrieve. Fetching a file with a specific version in FileDAG consists of two steps: Retrieve and Recover. A client sends a retrieve message containing CID_v to a miner that provides retrieval services to get the list of CIDs to recover v. After receiving a retrieve message, the miner first looks up in its DAG ledger to locate CID_v and then calls function Retrieve(CID_v) and forwards its output versions to the client. In the example illustrated in Fig. 4, the CID list contains CID_v and CID_Δ.

Download & Recover. After obtaining versions, the client calls function Recover(versions) to downloads all related file fragments based on versions. For each file fragment, the client needs to send a download message along with the CID to the miner who stores the data. After gathering all the required components, the client patches the increments to the original file to recover file v.

4 Analysis

In this section, we analyze two properties of FileDAG: consistency and efficiency (in terms of storage cost).

4.1 Consistency

Byzantine atomic broadcast [33] guarantees the following properties:

- **Agreement.** If an honest node p commits vertex i in a DAG, then every honest node p' eventually commits i.
- **Integrity.** For each round r and node p, an honest node p' accepts at most one vertex proposed by p in round r.
- **Validity.** If an honest node p proposes a vertex i in round r, then every honest node p' eventually commits i.
- **Total order.** If an honest node p commits vertex i before committing vertex j, then no honest node commits vertex j without first committing vertex i.

The construction of the two-layer DAG ledger ($\mathrm{E}_l \cup \mathrm{E}_u$) in FileDAG follows the algorithm of DAG-Rider; thus $\mathrm{E}_l \cup$

E_u can be reduced to DAG-Rider's ledger. As DAG-Rider is a byzantine atomic broadcast implementation which guarantees the above properties, the two-layer DAG ledger of FileDAG possesses these properties as well.

Definition 4.1. *(Consistency of multi-version DSN). For any version v of a file, an honest node can be convinced by the PoS proof that v is available in the FileDAG network only when v is indeed available; and if an honest node claims that v is available, then all other honest nodes claim the same.*

Theorem 1. *FileDAG meets consistency of multi-version DSN.*

Proof. First, the set of PoS committed in the FileDAG ledger is consistent. Based on the agreement and validity of byzantine atomic broadcast [33], each PoS proposed by an honest node is committed by all honest nodes, and every honest nodes commit the same set of PoSes.

Second, the consistency of PoS verification, i.e., for any version v of a file, if any honest node p accepts that v is available by verifying the PoS proofs, then every other honest node p' accepts that v is available if p' verifies the corresponding proofs. FileDAG employs the PoS algorithm of Filecoin. Assuming the soundness of the PoS algorithm, i.e., a miner can output a valid PoS of a file if and only if it is able to output a copy of the file, every honest node outputs the same verification result for the same PoS. Provided the consistency of PoS committed in the FileDAG ledger, the array of CID versions output by function $\mathsf{Retrieve}(v)$ is determined for determined v, and thus the consistency of PoS verification is satisfied.

Last, FileDAG meets consistency of multi-version DSN. Assume that the diff algorithms and the corresponding patch algorithms are correct so that when comparing two files (A and B), a diff algorithm always generates the same increment, and patching the increment to A always yields B. Combining the ledger consistency and the consistency of PoS verification, one can see that the consistency of multi-versioned files is proved. □

4.2 Storage Cost

Next we analyze the storage cost of FileDAG. Let S_t denote the size of the tth version of a multi-versioned file and I_t denote the size of the increment that the tth version differs from its previous version. After analyzing the growth of several GitHub repositories, we found that for each repository, typically there are two types of modifications for its growth, namely revision and addition. A revision (REV) operation on a repository does not significantly change the size of the repository, and the size of increment between the updated version and its previous version is much smaller than that of the whole repository. An addition (ADD) operation usually adds a large quantity of contents to a repository. For most repositories under our analysis, the number of revision operations is about ten or hundred times of that of the addition operations. But the size of the increment brought by an addition operation is ten or hundred times of the increment brought by a revision operation. To analyze the storage cost of FileDAG, we make a few assumptions on the growth of a multi-versioned file.

Consider the initial version of a file as an empty file with size zero, then the creation of a file can be regarded as an addition operation. In other words, we have $S_0 = 0$ and S_1 is the length of the original file.

For each version $v_t, t > 1$, the type τ_t of operation that outputs v_t can be regarded as a random variable (and the sequence of operations observed by each node is consistent, as proved in Theorem 1). Let

$$\mathsf{Prob}(\tau_t = \mathsf{ADD}) = p, \mathsf{Prob}(\tau_t = \mathsf{REV}) = 1 - p$$

and

$$I_t = \begin{cases} r_t & \tau_t = \mathsf{REV} \\ a_t & \tau_t = \mathsf{ADD} \end{cases} \quad (1)$$

For a specific multi-versioned file, one can assume that the ratios $\frac{1-p}{p} < 1$ and $\frac{\mathrm{E}(r)}{\mathrm{E}(a)} > 1$ are constants. Then we have

$$\frac{1-p}{p} \cdot \frac{\mathrm{E}(r)}{\mathrm{E}(a)} = O(1)$$

Theorem 2. *Let C and C' denote the expected storage cost of storing a file having n versions without and with the increment mechanism, then we have $C' = O(n^{-1})C$.*

Proof.

$$\mathrm{E}(I_n) = p\mathrm{E}(a) + (1-p)\mathrm{E}(r),$$

$$\mathrm{E}(S_n) = \sum_{t=1}^{n} p\mathrm{E}(a) = np\mathrm{E}(a)$$

$$C = \sum_{t=1}^{n} \mathrm{E}(S_t) = \frac{n(n+1)}{2} p\mathrm{E}(a) = O(n)\mathrm{E}(S_n).$$

$$C' = \sum_{t=1}^{n} \mathrm{E}(I_t) = np\mathrm{E}(a) + n(1-p)\mathrm{E}(r).$$

Based on our assumption,

$$\frac{1-p}{p} \cdot \frac{\mathrm{E}(r)}{\mathrm{E}(a)} = O(1)$$

then

$$C' = np\mathrm{E}(a)(1 + O(1)) = O(1)\mathrm{E}(S_n).$$

Therefor, the expected storage cost of FileDAG to keep all versions of a multi-versioned file is nearly equal to the size of the newest version of the file, while traditional solutions cost $O(n)$ times in expectation.

$$C' = O(n^{-1})C.$$

□

In section 5, we report our experimental results to further support the above conclusion.

5 Performance Evaluation

To evaluate the performance of FileDAG, we carry out real experiments. Specifically, we implement FileDAG based on the description in Section 3, and perform the full processes of put and get operations.

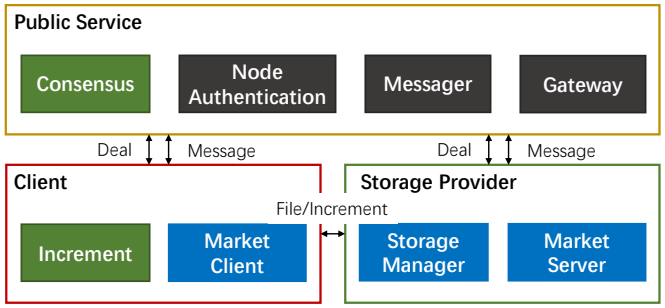


Fig. 5: Block diagram of FileDAG

5.1 Implementation

As FileDAG shares many common properties with Filecoin, we implement it by making modifications on Filecoin. This also provides a chance for us to objectively compare FileDAG with Filecoin. Venus, which has a modular design and was written in Golang, is one of the four main implementations of Filecoin. We fork a branch from Venus (version v1.1.3-rc1) to build FileDAG. A typical Venus deployment consists of three components, namely Public Service, Client, and Storage Provider (Storage Miner) [34]. Correspondingly, FileDAG has the same three components and the modules included in each component are illustrated in Fig. 5. Specifically, we build the Consensus and Increment modules from scratch, and they are marked green in Fig. 5. Additionally, we make adaptations on the Market modules (Market Client and Market Server) and the Storage Manager of Venus and reuse them in FileDAG – they are colored blue in Fig. 5. The three modules marked gray, i.e., Node Authentication, Messager, and Gateway, are directly inherited from Venus. The Increment module is implemented with 1096 lines of code in Golang and is included in Client. It consists of an Increment Generator and an Increment Accumulator, which can be deployed separately, as a data consumer who does not create or update files may not need the Increment Generator while a data provider who does not download files may not need the Increment Accumulator. We build a brand-new Consensus module to replace the one in Venus and develop our two-layer DAG ledger for FileDAG. The upper layer of the ledger is realized by an independent prototype of DAG-Rider with 424 lines of code in Golang. Note that the lower layer is implemented when revising the Market Client. We also modify the Market Server in Storage Provider to attach version control information when transferring files/increments, and add the Retrieve() function to the Storage Manager module to obtain the versions of all fragments needed to recover the file of a particular version. Modifications on Venus take 603 lines of code in total. Other modules of Venus that are not mentioned in this paper stay as they are so that we can objectively evaluate the performance enhancement brought by our innovation. All components of FileDAG implementation are written in Golang, and we build and test FileDAG with go1.17.11.

5.2 Experiment Setup

We deploy FileDAG on 5 computers, with each having 2-Core CPU, 4GB memory and 40GB NVMe SSD, and running Ubuntu 22.04 LTS. The bandwidth of each computer is 1MB/s. We use the 5 computers to run: 1 Service node, 3 Storage Providers, and 1 Client.

Based on the design difference between FileDAG and Filecoin, one can see that the performance changes brought by FileDAG come from three aspects: 1) storage space saved by the increment mechanism and the novel DAG ledger, 2) extra processing latency (in both uploading and downloading) brought by the increment mechanism, and 3) the decreased transmission latency due to smaller sizes of the payloads. We evaluate the storage cost and runtime of the operations in the put and get phases when providing multi-versioned file storage.

Files used in our evaluation consist of three types: text, multimedia, and binary. There exist plenty of online text files, e.g., code repositories, at GitHub. As shown in Table 3, we clone 4 repositories from GitHub, namely IPLD, go-ipfs, ccf-deadlines, and Git. We extract all the versions in each repository, and store them in our implemented FileDAG network. Each of these repositories has hundreds of versions and their average sizes range from 1.8 MB to 50.9 MB. Additionally, we use FileDAG to store a multi-versioned presentation PPT (Microsoft PowerPoint) as an example of multimedia file. Such files are common in practice as when preparing academic reports or degree defenses, people usually make revisions on a PPT multiple times and keep historical versions for possible rolling backs. In our evaluation, the PPT file used is a research report maintained by one of the authors. This file has 21 versions and the average size is 8.7 MB. Finally, we take the APK (Android application package) files of several popular apps as examples of binary files. These apps include a game (Minecraft), an instant messaging software (WeChat), and an entertainment app (Netflix). We have downloaded these APK files from Uptodown[2], which provides downloads of APK files with different versions. As of this writing, WeChat and Netflix have 18 and 20 versions on Uptodown, while Minecraft has 277 versions. The average sizes of them range from 14.4 MB to 228.4 MB. We test the APK files because the BSDiff algorithm used in our Increment module has been widely adopted to generate patch files for APK updates in Android applications. For comparison purpose, we select two market-tested DSNs, i.e., Filecoin and Sia mentioned in Section 2, as the baselines for our evaluation. Particularly, we include two implementations of Filecoin, namely Venus and Lotus, for a more comprehensive comparison study.

5.3 Evaluation Results

A summary of the evaluation results averaged over 100 trials on the eight datasets mentioned above are reported in Table 3, in which the three sections show the evaluation results in terms of storage, put (upload) runtime, and get (download) runtime. Storage measures the storage cost of the DSNs storing a multi-versioned file. The put/get runtime measures the average time cost of the DSNs to complete a put/get operation over one version of a multi-versioned file. One can see that FileDAG saves up to $25\% \sim 99\%$ storage space compared to Filecoin. This is because FileDAG stores increments instead of the complete versions of the multi-versioned files. For the text and multimedia files,

2. https://en.uptodown.com

TABLE 3: Comparison between FileDAG and other state-of-the-art DSNs

	Text				Multimedia	APK (Binary)		
	IPLD	go-ipfs	ccf-deadlines	Git	PPT	Minecraft	WeChat	Netflix
# of versions	212	129	244	845	21	277	18	20
Average size (MB)	3.3	7.3	1.8	50.9	8.7	14.4	228.4	99.1
Storage (FileDAG) (MB)	7.1	16.3	2.9	104.8	11.8	1503.0	3052.3	356.8
Storage (Filecoin (Venus)) (MB)	712.8	985.3	483.4	43035.7	183.7	3994.4	4110.4	1981.7
Storage (Filecoin (Lotus)) (MB)	712.8	985.3	483.4	43035.7	183.7	3994.4	4110.4	1981.7
Storage (Sia) (MB)	642005.1	47889.1	11850.5	2592801.5	391.6	12839.2	78046.4	47644.7
Put runtime (FileDAG) (s)	5.1	5.3	5.0	5.4	7.7	22.2	760.5	92.1
Put runtime (Filecoin (Venus)) (s)	9.4	13.7	7.9	59.2	14.5	20.9	232.3	103.3
Put runtime (Filecoin (Lotus)) (s)	9.3	13.8	8.2	59.0	14.7	20.3	234.2	106.3
Put runtime (Sia) (s)	112.6	407.1	51.1	500.9	74.1	33.7	534.6	227.7
Get runtime (FileDAG) (s)	0.8	1.0	0.7	1.0	1.3	6.6	182.8	19.2
Get runtime (Filecoin (Venus)) (s)	4.2	8.3	2.6	53.5	9.9	15.7	232.7	103.2
Get runtime (Filecoin (Lotus)) (s)	4.0	8.4	2.7	53.9	9.6	15.5	230.9	104.1
Get runtime (Sia) (s)	8.3	24.1	5.0	106.6	20.1	43.8	671.2	313.2

the put/get runtimes of FileDAG are significantly shorter than those incurred by other DSNs. However, limited by the performance of the increment generating algorithms processing binary files, the put runtime of FileDAG might be longer than those incurred by other DSNs, especially for WeChat, whose size is much bigger than those of the other two binary files. But this doesn't mean that FileDAG is not suitable for binary files. One can see that the get runtime of FileDAG is still significantly shorter than those of the other two Filecoins. As in practice, a binary file, e.g., a software installer, is usually downloaded multiple times (by different team members, or from different computers) once being put on the network, the longer time of an upload in FileDAG can be easily amortized by the shorter time of many downloads. Besides, as shown in Table 3, compared to FileDAG and Filecoin, Sia has extremely high storage cost and latency; thus it is ignored in the following studies.

To better demonstrate the superiority of FileDAG over the baseline DSNs, we use Fig. 6 and Fig. 7 to respectively depict the accumulated storage cost and the put/get runtimes of the three types of files under our study. Due to limited space, we report the results of only one multi-versioned file in each category and select Git for text, PPT for multimedia, and Minecraft for binary. Particularly, in Fig. 7, as it is impossible to draw the bar graphs of all versions, we choose 5 versions of each file, including the initial and final ones, to illustrate the results. The experimental results of other files exhibit very similar trends.

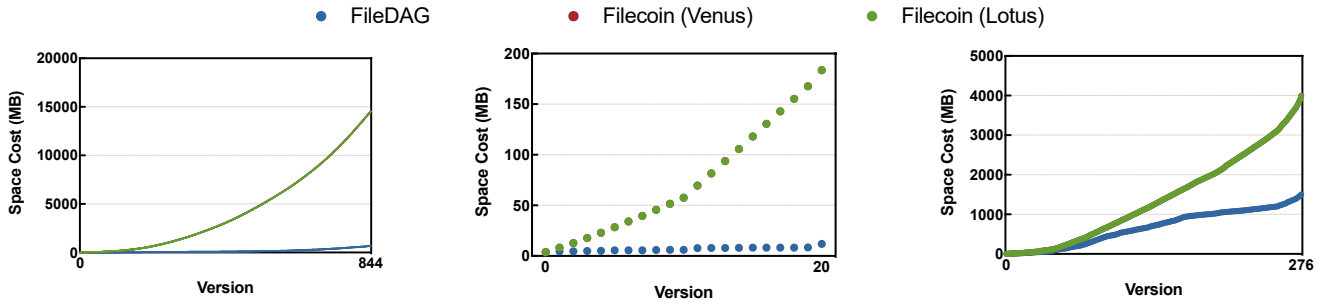
Storage Cost. We measure and compare the disk usages of the DSN miners to demonstrate that FileDAG achieves low storage costs. As one can see from Fig. 6[3], while a multi-versioned file updates its versions, the storage size increases for all DSNs, but the growth of FileDAG is much slower compared to both implementations of Filecoin. More specifically, one can see that the storage costs of Filecoin exhibit roughly quadratic growth, while those of FileDAG show approximately linear growth. This confirms our analysis in Section 4. One can also see that the more version a file has, the more storage space FileDAG can save via the increment mechanism, and the size of the increment generated is related to how much a version differs from its previous version. This explains why the save of the storage space of Minecraft by FileDAG is not as big as those of the other two files, as versions of Minecraft are quite different.

Put Runtime. The runtimes of the put phase of FileDAG and the baselines storing the above-mentioned three files are reported in the top three subfigures of Fig. 7. In a typical DSN, the put phase consists of three steps: processing, transmission, and on-chain confirmation. During the processing step, a DSN client packs a new file version (in Filecoin) or the increment of the new version (in FileDAG) into specific format of payloads. Transmission gets the payloads transmitted from the client to the designated storage miner. After transmitting the file (or increment) to the miner, the client creates a transaction that contains the file CID and the miner ID, then gets it confirmed on the blockchain to complete the storage put operation. To better demonstrate these three steps, we split each bar in Fig. 7 into three sections with different textures.

As shown in Fig. 7a, Fig. 7b and Fig. 7c, one can see that the on-chain confirmation latency of all the tested files are similar and remain stable. This is because the transaction sizes remain stable no matter how large the files are as transactions only contain CIDs or hashes whose sizes are constants. Additionally, based on our observation, the confirmation latency of FileDAG is about 1 second shorter than those of the two Filecoin implementations, thanks to the performance improvement brought by the consensus algorithm in our two-layer DAG ledger. Processing and transmission latencies intuitively depend on the sizes of the payloads. In Fig. 7b and Fig. 7c, one can see that due to increment generation, FileDAG has higher processing latencies and lower transmission latencies[4]. In most cases, the put latencies of FileDAG are lower than those of the baseline. In some other cases such as for Minecraft, due to the complexity of the increment generation algorithm, the put latencies of FileDAG might be longer, but they are still acceptable, as usually put (upload) is an infrequent

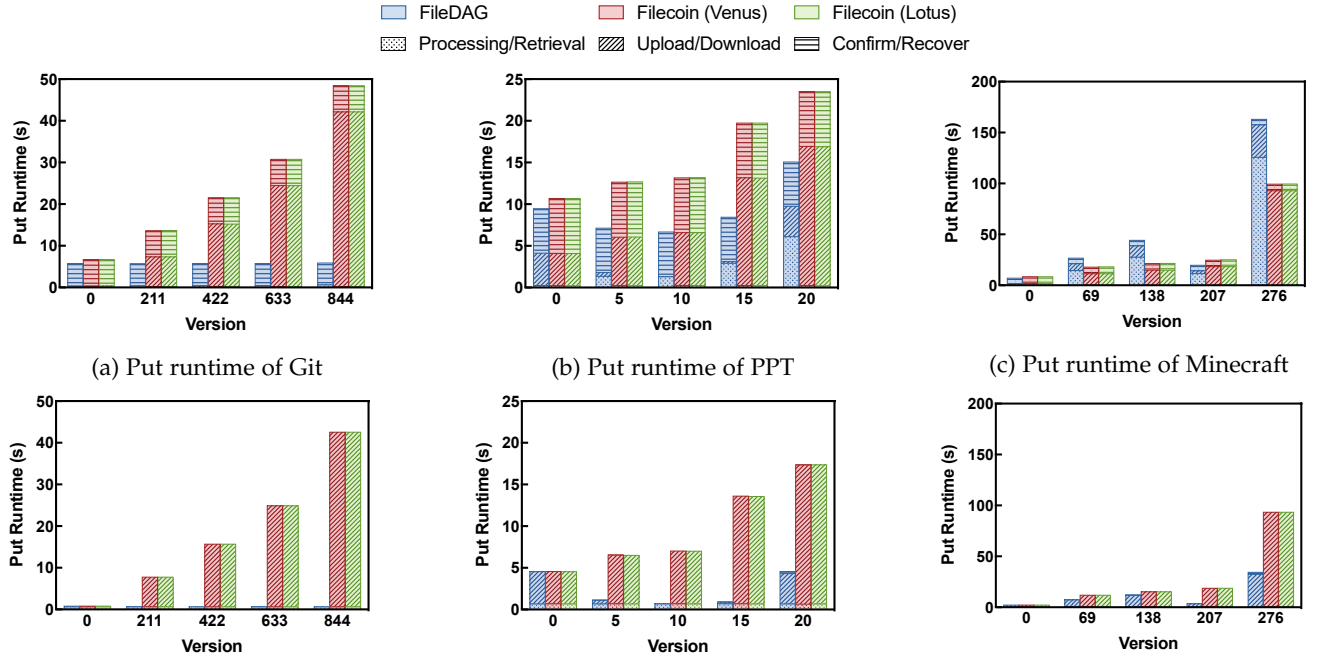
3. Because Venus and Lotus adopt the same deterministic packing algorithm, these two DSNs take equal storage spaces; thus the curves for their storage cost results overlap in Fig. 6.

4. The processing and transmission latences of Git in Fig. 7a is not obvious as they are too short.



(a) Git storage cost (b) PPT storage cost (c) Minecraft storage cost

Fig. 6: Full Version Storage Costs



(a) Put runtime of Git (b) Put runtime of PPT (c) Put runtime of Minecraft

(d) Get runtime of Git (e) Get runtime of PPT (f) Get runtime of Minecraft

Fig. 7: Put and get runtime

operation compared to the get (download) operation of the same file. What's more, one can see that for the initial version of each file, FileDAG and Filecoin spend equal time on processing and transmission, because all DSNs process and transmit the same full-versioned file.

Get Runtime. Similar to the put phase, the get phase of most DSNs also consists of three steps: retrieval, transmission, and file recovery. During retrieval, a miner gathers the required information (CIDs and the miner addresses) from the blockchain. Then the corresponding files (or increments) are sent to the client during the transmission step. As files can be encrypted for transmission or transmitted in small pieces, the client needs to recover the file in the last step. Therefore we also split each bar in Fig. 7a, Fig. 7b, and Fig. 7c into three sections with different textures. One can see that the latency of a get operation is mainly attributed to retrieval and transmission, while file recovery latency is negligible. The retrieval latency of FileDAG and Filecoin are close and both stay stable at around 0.7 seconds. The transmission latency of FileDAG is lower because of the increment mechanism. Particularly, version 207 of Minecraft has a small update compared to version 206, so the corresponding increment is small, thus the transmission process finishes shortly. Note that the latency saved by FileDAG during the get phase is much more than that saved during the put phase, because the increment generation takes more time than recovery. As we have mentioned earlier, practically a file is usually downloaded multiple times once being put on the network, the total latency saved by FileDAG can be significant.

6 Conclusions

In this paper, we describe the design and implementation of FileDAG, a DSN built on DAG-based blockchain. FileDAG supports file-level deduplication in storing multi-versioned files by adopting an increment mechanism. Besides, FileDAG supports flexible and storage-saving file indexing by introducing a two-layer DAG-based blockchain

ledger. We implement an actual instance of FileDAG and evaluate its performance. The results demonstrate that FileDAG outperforms the state-of-the-art industrial DSNs in storage cost and latency. In our future research, we plan to improve the performance of DSN by designing a faster and more effective proof of storage mechanism to further save storage cost of DSNs.

REFERENCES

[1] M. Xu, Z. Zou, Y. Cheng, Q. Hu, D. Yu, and X. Cheng, "Spdl: A blockchain-enabled secure and privacy-preserving decentralized learning system," *IEEE Transactions on Computers*, 2022.

[2] C. Liu, H. Guo, M. Xu, S. Wang, D. Yu, J. Yu, and X. Cheng, "Extending on-chain trust to off-chain – trustworthy blockchain data collection using trusted execution environment (tee)," *IEEE Transactions on Computers*, vol. 71, no. 12, pp. 3268–3280, 2022.

[3] M. Xu, S. Liu, D. Yu, X. Cheng, S. Guo, and J. Yu, "Cloudchain: a cloud blockchain using shared memory consensus and rdma," *IEEE Transactions on Computers*, 2022.

[4] B. Cohen, "The bittorrent protocol specification," 2008.

[5] J. Tate, P. Beck, H. H. Ibarra, S. Kumaravel, L. Miklas *et al.*, *Introduction to storage area networks*. IBM Redbooks, 2018.

[6] P. Labs. (2017) Filecoin: A decentralized storage network.

[7] S. Wilkinson, T. Boshevski, J. Brandoff, and V. Buterin, "Storj a peer-to-peer cloud storage network," 2014.

[8] D. Vorick and L. Champine, "Sia: Simple decentralized storage," *Retrieved May*, vol. 8, p. 2018, 2014.

[9] the Swarm team. (2021) Swarm: Storage and communication infrastructure for a self-sovereign digital society. [Online]. Available: https://www.ethswarm.org/swarm-whitepaper.pdf

[10] P. Labs. (2022) Web3 storage - the simple file storage service for ipfs & filecoin. [Online]. Available: https://web3.storage/docs/

[11] BitTorrent. (2022) Btfs. [Online]. Available: https://docs.btfs.io/

[12] Filebase. (2022) Use filebase as the origin for your cdn. [Online]. Available: https://filebase.com/solutions/content-delivery/

[13] P. Labs. (2022) Merkle dags. [Online]. Available: https://docs.ipfs.tech/concepts/merkle-dag/

[14] R. Achar, *The Global Object Tracker: Decentralized Version Control for Replicated Objects*. University of California, Irvine, 2020.

[15] GitHub. (2022) Configuring branches and merges in your repository. [Online]. Available: https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository

[16] S. Popov, "The tangle," *White paper*, vol. 1, no. 3, 2018.

[17] L. Baird, "The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance," *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, vol. 34, 2016.

[18] J. Benet, "Ipfs-content addressed, versioned, p2p file system (draft 3)," *arXiv preprint arXiv:1407.3561*, 2014.

[19] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[20] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The skein hash function family," *Submission to NIST (round 3)*, vol. 7, no. 7.5, p. 3, 2010.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43.

[22] (2022) Regarding gnutella. [Online]. Available: https://www.gnu.org/philosophy/gnutella.html

[23] (2022) Coral cdn. [Online]. Available: http://www.coralcdn.org/

[24] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *Usenix annual technical conference, general track*, 2006, pp. 43–56.

[25] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla, "Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 468–477.

[26] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 281–310.

[27] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "{Bitcoin-NG}: A scalable blockchain protocol," in *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, 2016, pp. 45–59.

[28] N. community. (2022) Nxt whitepaper. [Online]. Available: https://nxtdocs.jelurida.com/Nxt_Whitepaper

[29] Y. Ribero and D. Raissar, "Dagcoin whitepaper," *Whitepaper, no. May*, pp. 1–71, 2018.

[30] NAKIVO. (2021) Incremental backup. [Online]. Available: https://www.nakivo.com/incremental-backup/

[31] E. W. Myers, "An o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[32] C. Percival, "Naıve differences of executable code," 2003.

[33] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, ser. PODC'21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 165–175. [Online]. Available: https://doi.org/10.1145/3465084.3467905

[34] I. Force. (2022) Venus docs. [Online]. Available: https://venus.filecoin.io