École Polytechnique Fédérale de Lausanne

Making the Inter-Planetary File System (IPFS) more reliable

by Qiyuan Liang, Yuening Yang

Master Semester Project Report

Vero Estrada-Galiñanes
Project Supervisor

Prof. Bryan Ford
Project Responsible

EPFL IC IINFCOM DEDIS
BC 210 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 6, 2023

Abstract

One of the main benefits that distributed file systems have over traditional file systems that are stored on a single device or server is data redundancy. To achieve data redundancy, distributed file systems usually use replication. However, it is debatable whether replication is the best option in distributed settings. Researchers have proposed schemas that have some advantages over replication in certain aspects, for example, lowering the storage overhead, requiring less maintenance effort, and improving recovery capability. One of these schemas is Alpha Entanglement (AE) codes. In this project, we try to integrate IPFS, one of the most widely used distributed file systems nowadays, with AE codes. Users could only use replication in file reliability solutions provided by IPFS, which could be a large storage overhead and resource contention. The intention of this project is to offer one more solution on IPFS for file reliability. With AE codes integrated into IPFS, users could achieve a similar reliability level as replication with less storage and a slight compromise in the bandwidth overhead. Our solution builds on top of both IPFS and IPFS Cluster. Users are able to define and control their file reliability. The evaluation results suggest a $2.7\times$ storage saving, with a maximum 70% more bandwidth usage for file recovery, compared to replication.

Contents

Chapter 1

Introduction

The concept of web3 has gained a lot of attention nowadays [1, 2, 6, 17]. Web3 is an approach to develop the next generation World Wide Web with the use of decentralized and distributed technologies to eliminate the central authority. InterPlanetary File System (IPFS) is a peer-to-peer distributed file system that enables users to view and exchange files over a network without a central authority [3]. It is often used in conjunction with web3 technologies to build decentralized applications. These applications rely on IPFS to store and access data in a decentralized way, as IPFS provides a fast and efficient way to share and access large amounts of data over a distributed network. IPFS can be seen as an infrastructure that supports the web3, therefore, it is important that we achieve high reliability and availability at this layer so that higher-layer applications enjoy the benefits.

However, there are some well-known limitations of IPFS, for example, limited adoption in the market, unclear scalability, limited accessibility, and large file uploading procedures [7]. From the reliability aspect, IPFS has one major problem. When users upload a file, the file is only stored locally. The rationale behind IPFS design is three-fold: efficiency, privacy, and user willingness. Storing and sharing files locally reduces the need for data to be transferred over long distances, which can improve the efficiency of the network. Also, storing and sharing files locally can help to protect user privacy, as the data is not being stored on other nodes where it could potentially be accessed by third parties. A third reason is to provide the user with some protection against third parties storing unwanted files in the user's computer. However, we still see IPFS as a distributed file system, because the provider record of the uploaded file gets distributed. If another user comes and wants to look for the file, it will find the provider record inside the network, and finally, contact the node that stores the file to retrieve it. If the user wants, he could also become the second provider of the file. In this case, if the first provider goes offline, other users would still be able to find the file because there is a second provider. This design is helpful when the uploaded file is hot or you have enough recourse, multiple providers

Our code and models are publicly available at
`https://github.com/dedis/student_22_ipfs_alpha_entanglement_code`

reduce the contention and improve the file availability. However, if the file is not popular with the public nor there are enough resources to keep another copy on another machine, there is likely to be only one provider of the file inside the network. This means when the only provider goes offline, other users lose file accessibility entirely.

IPFS developers put significant effort to improve file reliability, for example, Filecoin, IPFS pinning service, and IPFS Cluster [15]. Filecoin uses blockchain technology to create a marketplace for buying and selling storage capacity. It is designed to allow users to store their data on a distributed network of computers. Filecoin aims to create a more secure, decentralized, and efficient way to store data and give users more control over their data and its accessibility. It is secured through a proof-of-replication and proof-of-spacetime mechanism, which ensures that data is stored and available as intended [4]. IPFS pinning is a service that allows users to store and retain a copy of specific files or data on the IPFS network [11]. When a user "pins" a file, they are effectively telling the IPFS network to keep a copy of that file and to ensure that it is always available for retrieval. This can be useful for keeping important data available even if the source is no longer available. There are several different services that offer IPFS pinning, including centralized and decentralized options. Some of these services charge a fee for their services, while others are offered for free. IPFS Cluster is a tool for managing a group of IPFS nodes as a single entity, allowing users to store and manage data on the IPFS in a more organized and efficient way. It is designed to work alongside the IPFS network and to provide additional features and functionality for users who need to manage large amounts of data or who want to share data with a specific group of people. All the solutions mentioned above build on replication. Replication does offer file reliability. With only one replication alive, the file would be available. However, the benefits come from the cost of high storage overhead. People with limited storage resources would like to have another solution, achieving the same redundancy level while demanding less storage.

There are some schemas of different features proposed on other distributed file systems. One prominent distributed file system is Swarm [16]. Swarm and IPFS are both decentralized, peer-to-peer file-sharing systems. However, there are some critical differences between Swarm and IPFS. One of them is the way that each system stores and retrieves data. Unlike IPFS, when users upload a file into Swarm, the file is divided into small blocks, and each block goes to a potentially different node. In other words, the file is no longer only locally available, it could go everywhere. To reconstruct the file, Swarm maintains a Merkle tree structure, and only the root hash is needed as input. Internal nodes contain pointers to other internal nodes or data nodes. Nygaard et al. proposed an entanglement algorithm on top of Swarm with user-controlled redundancy [13]. They observed the vulnerability introduced by the additional introduced hierarchical data structure, Merkle trees, and cascading failures due to uneven replication distribution. As a result, they designed Snarl, using Alpha Entanglement (AE) codes as the underlying mechanism [9]. Their evaluation shows that Snarl increases storage utilization by $5\times$ in Swarm with improved file availability, with acceptable bandwidth overhead.

Due to the low maintenance effort and high robustness against data loss, we believed that

AE codes are indeed a better candidate, compared to replication, in distributed file systems. Therefore, in this project, we tried to build a similar algorithm to Snarl in IPFS, to improve the file reliability in IPFS as well. However, due to the previously mentioned difference between Swarm and IPFS, we would not be able to deploy the same algorithm in IPFS. Therefore, we made some adaptions to the original algorithm and proposed some new features that are useful in real-life scenarios. One key challenge we are facing is the way we distribute files remotely. If all files are only stored locally, there is no file reliability improvement from the perspective of IPFS. Among all solutions proposed by IPFS, we find IPFS Cluster being the most flexible and amenable to AE codes. Our solution builds on top of both IPFS and IPFS Cluster. We offer a new application layer for user interaction and hide the details of the entanglement procedure. We call it IPFS Community. Users interact with IPFS Community in a similar fashion to that with IPFS.

When users upload a file using our service, we first use the AE codes to generate entanglement that could be used later on to recover the original file. IPFS uses Merkle Directed Acyclic Graph (DAG) as the internal representation. We rely on IPFS Cluster to distribute the entanglement remotely. The benefit of doing so is that we the local resources are no longer available, we could still recover the original file through remote entanglement. After the upload is successfully done, users will receive two results, one is the original file's content identifier (CID), the other is the metafile's CID. CID is the concept in IPFS. The connection between upload and download is CID. CID serves as the unique identifier for the file inside the IPFS network. The original file's CID is the same as the CID if users are to upload the file directly in IPFS. The metafile contains the information for all the entanglement. When users download a file, they are supposed to provide the original file's CID at minimum, If users want to enjoy the additional reliability from entanglement, they are supposed to provide the metafile's CID as well. If two CIDs are both provided, the download procedure directly downloads the original file when the original files are directly available. If the original files are missing, the download procedure will try to fetch the entanglement and restore the original file.

In short, our solution, IPFS Community, offers users a file reliability solution using AE codes. It builds on top of IPFS's and IPFS Cluster's open APIs. It allows users to control the redundancy level and manage the whole process of file redundancy. Through experiments, we found our solution uses $2.7\times$ less storage compared to replication. If the file is directly available, the bandwidth and memory usage are exactly the same as solutions using replication. When files are missing, the file recovery bandwidth usage is at a maximal 70% higher compared to replication. . Our core contributions could be summarized as follows:

- We proposed and implemented a new architecture using AE codes in IPFS. It allows users to choose a different solution from replication to improve file reliability.
- We evaluated the performance of the architecture and demonstrated its advantages and limitations against replication. Compared to replication, the storage saving is up to $2.7\times$, with 70% more bandwidth usage for file recovery, for $\alpha = 3$.

Chapter 2

Background

2.1 Alpha Entanglement Codes

Alpha Entanglement (AE) codes [9] is a redundancy scheme that is tailored for storage in unreliable settings. It achieves high fault tolerance and availability, and is able to recover from a large number of failures using low bandwidth and computation effort. In the meanwhile, it provides flexibility for the user to decide the erasure code parameter, so that different storage and bandwidth situations could be accommodated. Several redundancy schemes are commonly used, including replication or redundant array of independent disk (RAID) solutions. Yet these solutions facilitate the concurrent reading of the file, replication introduces large storage overhead. RAID solutions, such as RAID 5 and RAID 6, build on top of the erasure codes [14]. The commonly used type of erasure codes, Reed-Solomon (RS) codes, have a very low storage overhead. However, in practice, it is usually used in conjunction with replication because of its poor performance. Meanwhile, making RS codes useful requires constant monitoring and repairing, which becomes less practical in the case of decentralized settings because of the high communication cost and dynamic peer group. These disadvantages of either replication or RS code make AE codes a better candidate for decentralized systems.

AE codes create redundancy by entangling (mixing) files. Newly added blocks/files are entangled with old files to form a data chain. In case of data loss, these data chains offer a way to recover. The recovery program could use different paths to recovery. These paths could have different lengths, and the shorted path has a length of 2, which contains both a data block and an entangled block. Due to its high connectivity, it is very robust against failures. The AE codes are defined by three parameters: (i) α defines the number of parity blocks that are created per data block, (ii) s defines the number of horizontal strands, (iii) p defines the number of helical strands. Only α changes the storage overhead, while s and p change the topology of the generated data grid. Neither of these three parameters changes the cost of a single failure, which requires 2 blocks from the grid.

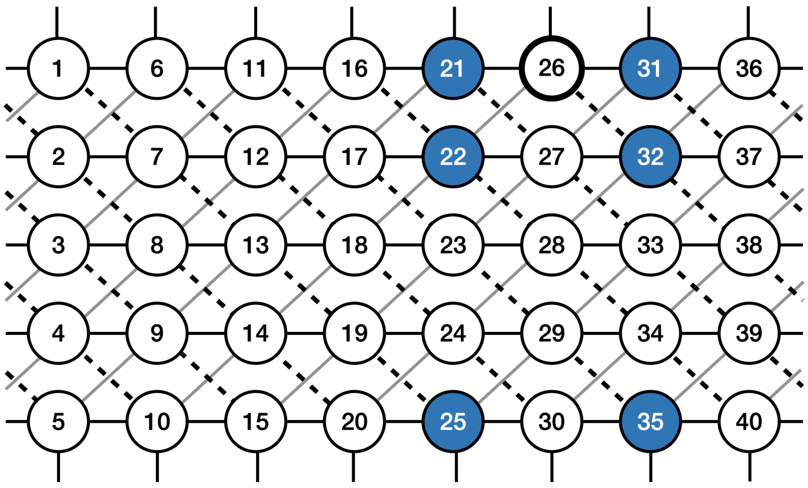


Figure 2.1: AE(3,5,5) lattice with $s = 5$ (rows) and $p = 5$ (columns/diagonals). Any node d_i participates in $\alpha = 3$ strands, e.g. d_{26} is a top node. Colored nodes are at one hop of node d_{26}.

When $\alpha = 1$, the generated data grid is only a strand/chain ($s = 1$ and $p = 0$). In case we have a sequence of data blocks $(d_1, d_2, d_3, ..., d_n)$, we insert 1 entangled block, $e_{i,j}$, between two consecutive data blocks, d_i, d_j. For simplicity, a dummy parity block, $e_{\_,1}$, is added at the beginning of the sequence. The sequence after AE codes becomes $e_{\_,1}, d_1, e_{1,2}, d_2, e_{2,3}, ..., e_{n-1,n}, d_n, e_{n,\_}$. The entangled block is computed as $e_{i,i+1} = e_{i-1,i} \oplus d_i$, for $i > 1$.

When $\alpha > 1$, the data grid is valid when $p \geq s$, and the data grid becomes intertwined ($s > 1$). For each data node, it connects to 2α other data nodes, i.e., it belongs to α strands. The strands could be classified into three groups, the horizontal (*H*), the right-handed (*RH*), and the left-handed (*LH*) strands. Each strand is considered independent, making it possible for us to repair multiple failures in the same round. For a data grid that has 2 or more horizontal strands, we could distinguish the data nodes from the row that they belong to, naming top, central, and bottom. For top and bottom rows, they are corner cases that require additional care. To build parity for the top and bottom rows, the entangled block is wrapped around. For example, we have an AE(3, 5, 5) scheme as shown in Figure 2.1, where $\alpha = 3, s = 5, p = 5$. We are inserting the d_{26}, it belongs to the top row. To generate entangled block for d_{26}, it is necessary to compute $p_{26,32}$, $p_{26,31}$, and $p_{26,35}$. The strategy applies to bottom rows as well.

2.2 Snarl: Entangled Merkle Trees

While AE codes are useful in the settings of decentralized file systems, modern systems, such as IPFS [3] and Swarm [16], rely on Merkle Tree (MT) to ensure the integrity of the stored file. When users upload files into the system, they remember their content addresses, usually the root hashes of the MT. The file blocks are then disseminated into the system. When users download files from the remote, they use the remembered content addresses as keys to search inside the system. After receiving all or partial file blocks, they could verify the file's integrity by computing the new root hash value of the MT and comparing it to the old one. Since the root value is used as the key to search inside the system, it introduces hierarchical dependencies, naming the user has to first reach the node that stores the root node, and then iteratively fetch internal nodes, until they reach the leaf nodes that contain the real content of the file. In case of any lost root node or internal nodes, we could not find the file content even if the leaves are alive. Current systems rely on replication to reduce the risk of data loss, but the replication factor is weakly guaranteed in decentralized systems. Entangled Merkle Tree (eMT) is a technique to reduce the impact of the hierarchical dependency [13]. Instead of relying on the underlying storage to ensure data availability, clients control the redundancy by themselves.

Snarl resides between the client and the underlying storage. Users only interact with Snarl when uploading and downloading files. Besides the original MT, Snarl produces additional eMTs for repair purposes. Both the original MT and eMTs are uploaded to the storage system, and they are indistinguishable from each other for other users. When there is no data loss, the original MT is used for file retrieval, therefore, no additional cost is introduced. Yet, when there is data loss, Snarl tries to repair the loss from other eMTs. The blocks that contain the root node and internal nodes are also considered for entanglement. In other words, in case of the root node or internal node loss in the original MT, we could use parity blocks to recover the content for these nodes.

Snarl consists of four major components, eMT-coder, repair, replicator, and connector. The eMT-coder deals with the encoding and decoding of the eMT. The repair component is only used when there is a need for data repair. The replicator is used to further improve file redundancy. The connector abstracts lower-level details of interacting with the storage layer.

The eMT-coder consists of three parts, mapper, swapper, and entangler. The original MT is a tree data structure that does not fit directly into the AE codes. The mapper maps the tree into the lattice with a post-order traversal of the tree. Yet, using the post-order traversal means that the parent and children nodes will be close in the lattice. This pattern introduces additional irrecoverable situations (loss of the parent parity node that leads to the failure of finding children parity nodes) that could be avoided initially. To do this, the swapper moves the parent at least one lead window (LW) away from their children. The lead window describes the interval that a helical strand revolves around the cylinder's axis.

2.3 IPFS: The InterPlanetary File System

IPFS [3] is a peer-to-peer (P2P) networking protocol that is used to store and share data in a distributed fashion. IPFS allows users to store whatever data locally. The data is addressed by content identifiers (CIDs), which are computed through cryptographic hash functions on its content. When there is a network connection available, IPFS allows peers to share data among themselves, and the same files could get replicated inside the system to improve data availability. IPFS is based on CIDs, thus, nodess could verify the files' integrity without putting trust in others. The files are immutable, and every change to the file creates a new file in the system. IPFS could detect and perform deduplication if multiple copies of the same file exist in the system to improve storage efficiency. After initializing an IPFS node, it contains an empty directory, and the directory is again addressable by the CID. Similar to local file systems, IPFS supports standard file operations that allow users to create new files, create new directories, move files among directories, and so on. Files could be found either through a path (path could contain CIDs inside) or a CID. In practice, the CID does not apply directly to a file. Files are split into blocks, and each block is stored separately. This allows intra-file deduplication as well, and facilitates partial retrieval to optimize bandwidth

CIDs ensure the integrity and verifiability of a single file or directory and give a single file a unique global address. However, the data stored inside IPFS has relations, e.g., files are stored inside a directory, blocks belong to the same file, and so on. IPFS uses Merkle Directed Acyclic Graphs (DAG) to model these relations. The parent node (directories) contains the CID of its children (files or subdirectories) as its own data. The CID of the parent node is then computed on its data. As a result, a single modification to the children nodes will be reflected in all its ancestors. Unlike MT, Merkle DAG allows one node to have multiple parents. Therefore, it allows more fine-grained deduplication. Deduplication could be done at a different granularity though. While the CID is used to deduplicate a file, we could use Merkel DAG to deduplicate a common subtree. The use of the Merkle DAG gives us a way to traverse the nodes and understand the links among them.

Traditional centralized networks have a trusted party to manage the network and content with location-based routing. The network is based on the locations (IPs) of the destination. In a decentralized network, where we use content routing, we have to find out the location of each content. IPFS uses libp2p as its underlying network stack. To make IPFS work, the network stack has to support three major functions: *Provide*, *Resolve*, and *Fetch*. *Provide* makes the local content available to other nodes. Yet, the content is not uploaded or replicated to other nodes. Nodes only get a pointer to the provider node. The data only gets replicated when other nodes are requesting the file, i.e., the same content is stored in multiple locations. The process is only temporary unless the requesting node decides to pin the content. *Resolve* helps to locate user-specified content in the network. A node request to k closest neighbors to the CID that it would like to find. Since the information is distributed, these nodes may not have the provider record of the content. In such a case, the requesting node has to perform a distributed hash table (DHT) query until it finally finds the target. *Fetch* retrieves the content from a provider. After finding out the multiaddress of the target node, we could perform a normal network request to retrieve the file.

There are still many use cases that the data is better to be mutable, for example, the web page. The web host would like to keep the same address while changing the content inside it. However, the CID is computed directly on the data, i.e., changing the content changes the address. IPFS solves this problem with InterPlanetary Name System (IPNS).

By default, the file in the IPFS is only stored locally. When you add a new file to the IPFS client, the file does not go anywhere until someone requests it from outside. The other user could choose to pin the content, and become an additional content provider. Any intermediate nodes also become a temporary content provider until the content is garbage collected. To allow other users to find your file, the file reference, CID, is made public and shared among other nodes. The DHT of IPFS is essentially used for the distribution and query of the CIDs. For files belonging to an individual, other nodes have no incentive to store them. This mechanism limits the reliability of the storage if the file you are offering is not popular, i.e., there will be no access to your file if your node is down. As a result, many third-party services are offering a paid remote pinning solution, when you want to (i) improve the file availability, (ii) have a persistent backup, (iii) store files larger than the local space limit. Besides the paid services, IPFS also announces a new application named IPFS Cluster. IPFS Cluster takes a pre-defined set of nodes, and performs global pin and file replication among them. IPFS Cluster offers a solution to increase horizontal scalability without any incentives. Despite the problem of file availability, the IPFS design also introduces an issue concerning file deletion. It is impossible to enforce file deletion once your file is replicated into other nodes, since all deletion only happens locally.

Chapter 3

IPFS's Solution for File Reliability

All existing solutions in IPFS for file reliability are based on replication. The purpose of this project is to implement entanglement codes in IPFS, which offers a solution that is not based on replications for users to choose from. Based on their own needs, users may prefer AE codes to replication, because of the low storage overhead and relatively efficient file recovery procedure.

Due to the substantial differences in the design of IPFS and Swarm and time constraints for this project, we were expected to analyze the differences and adapt the Snarl implementation or propose a different implementation of AE codes that is suitable for IPFS. As a result, our design differs from the Snarl implementation in the following way: i) the original file remains local; ii) the entangled file is uploaded to an additional service named IPFS Cluster.

IPFS and Swarm have very different design decisions on the way the files are stored [8]. In Swarm, nodes store blocks from other users. Particularly, when a user "uploads a file to the network", the Swarm client will chunk the file and sync it. During syncing, the network distributes the blocks, and eventually, remote nodes will store them. The data is stored together with the DHT. Swarm calls this solution DISC, which stands for Distributed Immutable Store for Chunks. When the user stores files in Swarm, the file is split into blocks of size 4 kilobytes (KBs). Each block is stored on nodes that are nearest to the block address. The nearest nodes are chosen based on the underlying Kademlia DHT algorithm. The service of storage comes at a price that the user has to "pay". The payment is via micropayment channels embedded in the Swarm design. The user obtains the file availability as a return, i.e., even if his nodes go offline, other users would be able to download the files, by requesting blocks from different nodes.

IPFS, as mentioned in the previous Section 2.3, chooses a lightweight design. The lightweight comes from the fact that DHT only stores the provider IDs of each CID, and no actual data is stored. IPFS is not really distributing the files, instead, it only distributes references (provider records) to files. There is no mechanism in IPFS that allows a user to request his file to be stored in other nodes. In other words, if the user's file is not popular, the user is likely to be the only provider in the network. As a result, if the user goes offline, there are no means that others could use to download the file. Though, if the file is popular, there could be many copies inside the network. It means that the file redundancy is well established. Even if the original node leaves, other users are still able to download.

The design of IPFS originates from the idea that the files should be stored only if the host would like to. If IPFS allows the files/blocks to be stored directly in other nodes' storage without the host's consent, like what is done in Swarm. There could be potential legal issues, i.e., the content of files is not allowed in certain regions. Examples could be copyrights, censorship, and so on [12]. Yet, if the user requests to download the file himself, it is his responsibility to follow the law.

The argument of IPFS storing files locally could be valid, but it inevitably introduces the problem of a single point of failure. The developers of IPFS are also aware of this issue. Therefore, they also introduce two different mechanisms to increase file persistence. The key ideas behind this are that nodes with large spare disk can rent their storage to users, and get paid for it.

3.1 IPFS Pinning Service and Filecoin

IPFS pinning service offers users an option to store their files remotely, and the files are accessible via the IPFS network. The pinning service becomes handy when: i) your local nodes are not always online, ii) you would like to make a backup or permanent copy of your file, iii) you do not have enough storage at your local node. There are many third-party commercial pinning services online. The usage is tightly integrated into the IPFS daemon. You could specify your target pinning service in the IPFS configuration, and IPFS would wrap and handles the interactions for you.

In the above use case, it is hard for the users to verify that the service provider stores the files. Users could request the stored file from the IPFS network. In case he receives a result, it only indicates that there is at least one node in the network storing the file, but not necessarily the service provider. Besides, the duration guarantee is weak, in the sense that the remote service provider could delete the files without your consent. To offer a strong guarantee, IPFS encourages its users to store data with Filecoin for long-term storage [15]. To work with Filecoin, users start a storage deal with the service provider. The network verifies that the provider does store the data. The payment is made on a regular base for the duration specified in the deal. Storage providers that do not follow the deal are penalized. But one side note is that storing files with Filecoin is computationally expensive and can be slow.

In either case, storing files with a remote pinning service or Filecoin means that there is trust between you and the third party. Data privacy and security become a rising concern. These are essentially the issues that we want to avoid when using decentralized systems [5].

3.2 IPFS Cluster

IPFS Cluster is a distributed p2p database with the purpose of coordinating pinnings. It operates on top of the IPFS, and offers file redundancy and availability. Clients interact with the IPFS Cluster, and the IPFS Cluster interacts with the IPFS.

The cluster shares a global pin set, which is tracked and modified by every node. The global pin set is stateful, which records the addition or removal of files. The global pinning is load-balanced, based on free disk space, pin queue size, and geographic location. The priority could be specified by users. Cluster nodes run a libp2p host and form a private network. Each cluster node offers an RPC interface open to other nodes. A node can retrieve information from other nodes, and add blocks to other nodes. When a node submits a file, it asks some other nodes inside the cluster to store it. In case of failure, try again. Exactly how many nodes depend on the configuration.

The pinning process of the IPFS Cluster has two main stages, namely, the cluster-pinning stage, and the IPFS-pinning stage. The cluster-pinning stage ensures that the pin is persisted by the IPFS Cluster, and broadcasted to other nodes, while the IPFS-pinning stage makes sure that every IPFS daemon that the IPFS Cluster node is running on pins the content. The pinning process can be summarized as follows: (i) a pin request arrives, (ii) the allocation process decides which nodes should pin the content, (iii) the pin request is also sent to the chosen nodes.

There are also collaborative clusters, which are untrusted third parties. These parties only get read permission on the IPFS Cluster. The followers subscribe to the cluster and pin things accordingly. It makes an easy way for the data host to distribute content, and for others to follow updates.

Chapter 4

Our Solution: IPFS Community

4.1 Preliminaries

There are multiple solutions to achieve our final goal, offering users AE codes, a different solution from replication, to improve file reliability. Yet, they have their own pros and cons. The following lists a few viable solutions: (i) use what is provided by IPFS's pinning service and Filecoins, (ii) builds on top of the IPFS Cluster, and (iii) change the logic that IPFS works underhood.

Solution (i) introduces an extra burden on users, i.e., users have to set up and possibly pay for the service. Meanwhile, although entanglement does reduce the storage overhead compared to replication, how service providers store the user-uploaded data is out of our control. Service providers are free to use any redundancy schema that they have at hand, and deliver it to the users. As a result, we consider this solution less optimal.

For solution (ii), we would build an application on top of the IPFS Cluster. IPFS Cluster will become our main entry point of interaction with the outside world, naming to send our entanglement blocks and ask them to be stored by other nodes. However, building software like so does introduce inefficiency, and the security guarantee for IPFS Cluster is weak. By default, all cluster nodes have to trust each other. Our initial goal is to offer a platform that could be used by the public decentralized. This limitation is somehow against our objective.

Solution (iii) would be the most optimal solution. All users using IPFS would not notice a difference. All the mechanisms of the AE codes work at the storage layer, without user interaction. One possibility is that we could modify the IPFS to mimic what is done by Swarm, by sending entanglement blocks to other nodes and asking them to store the content. Another possibility is that all files get entangled by default, therefore, creating a large entanglement mesh that boosts the IPFS file reliability. The prospect looks pretty, despite the fact that it is too exacting to fulfill. The intricate code design of IPFS forbids the work to be done within a reasonable amount of time.

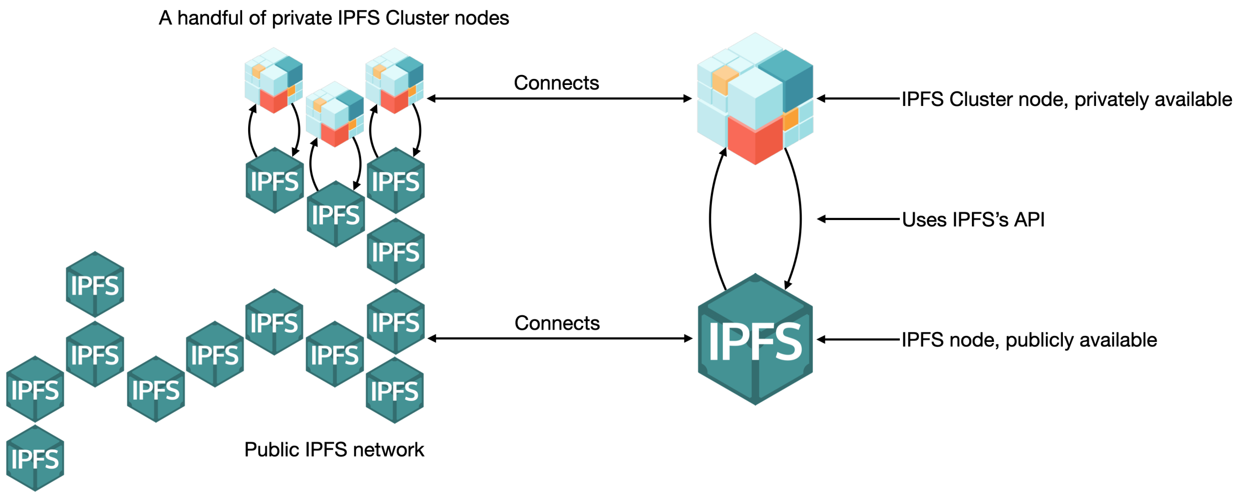

A handful of private IPFS Cluster nodes
Connects
IPFS Cluster node, privately available
Uses IPFS's API
Connects
IPFS node, publicly available
IPFS
Public IPFS network

Figure 4.1: IPFS Cluster architecture. An IPFS Cluster node runs on top of an IPFS node. IPFS Cluster nodes themselves do not store the file content. It instructs the IPFS node to do so. The connection between different IPFS Cluster nodes is only privately known. On the other hand, IPFS nodes are all publicly available, meaning that contents stored in them could be accessed by others. Unless you limit the accessibility of the IPFS nodes.

As a compromise, we seek for solution (ii), which uses IPFS Cluster as the underlying mechanism to distribute the entanglement. It is more flexible compared to Filecoin, such that we could tailor our own logic on top, and it is less tight compared to changing IPFS entirely.

4.2 Methodologies

4.2.1 Overview

As mentioned above, we decided to build our solution on top of IPFS Cluster. Before moving on to our detailed design, we first showcase how IPFS Cluster works under the hood. Figure 4.1 shows the architecture of the IPFS Cluster and how it interacts with IPFS. One IPFS Cluster node and one IPFS node are ideally running on the same machine, as shown on the right-hand side of the figure. The IPFS Cluster nodes interact with IPFS using IPFS's exposing APIs. The IPFS Cluster nodes know each other, and the connection is private. To bootstrap an IPFS Cluster node, it has to know who it can talk to initially, and what is the secret of the cluster. The IPFS node can be bootstrapped as normal, which connects to the public network.

IPFS Cluster is used to orchestrate a global pinning set. If you would like to have some content available on other remote machines, you could pin it via IPFS Cluster's API. Your IPFS Cluster node will submit a request to other nodes via the Remote Procedure Call (RPC) API.

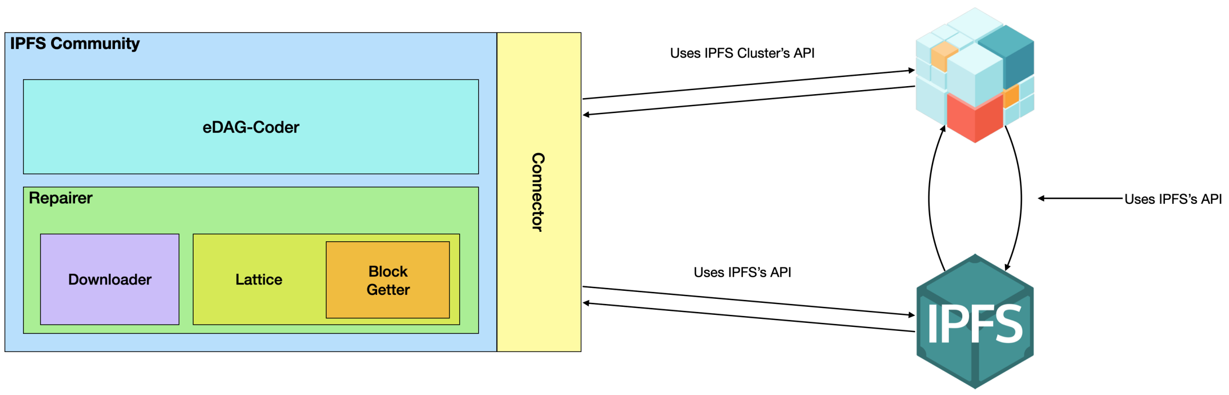
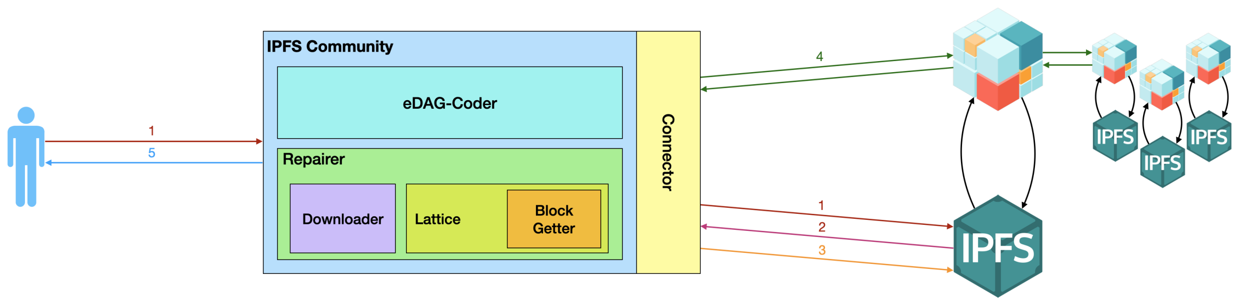


Figure 4.2: Our solution: IPFS Community architecture. The architecture can be divided into three main sub-components, eDAG-Coder, Repairer, and Connector. The eDAG-Coder performs the entanglement on the given input with DAG representation. The Repairer uses the entanglement to repair file/block loss if any. The Connector handles internal requests to IPFS Cluster or IPFS. The Repairer is further subdivided into modules, Downloader, Lattice, and Block Getter. How these three submodules interact with each other is illustrated in Figure 4.3.

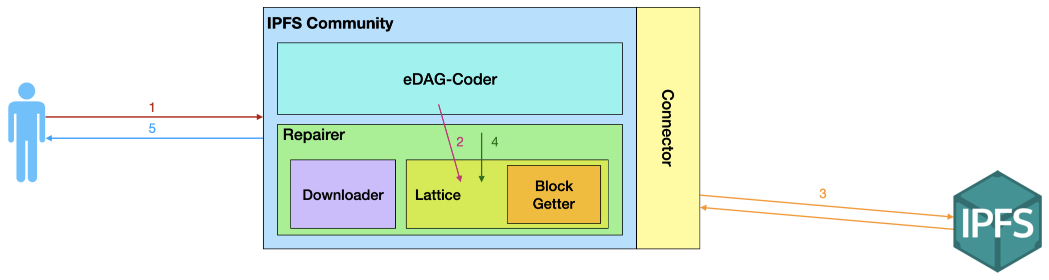
Upon receiving your request, the IPFS Cluster node will instruct the underlying IPFS node to pin the specified content. Through this mechanism, you are able to make your local files remotely available. The base for this mechanism to work is trust among nodes. The implication of the trust is discussed in Section 6.4.

The main use case that brings up the development of the IPFS Cluster is that users would like to have some of their files available on some other machines, such that he/she could, later on, retrieve them even his/her machine goes down. It is typically useful for a web service provider who would like to add some redundancy to his serving content. Therefore, the design is tailored to the use case. For example, the user could configure the system to perform the specified replication factor. In case some nodes go offline, it will re-trigger the replication, guaranteeing that the specified replication factor is followed. A consensus algorithm is also used to guarantee that every node in the cluster has the same view of the pinning set, avoiding the leader's role to control the whole system. Ideally, the cluster would work no matter which node leaves the system. The offered options for the consensus are Conflict-free Replicated Data Type (CRDT) or Reliable, Replicated, Redundant, And Fault-Tolerant (RAFT) consensus. Two algorithms are for different use cases, and the difference is mainly performance-wise.

Given the design of the IPFS Cluster, we propose our solution, IPFS Community. The idea is to give every participating individual, not necessarily from the same party, the chance to enjoy the service of file redundancy. Figure 4.2 shows the composing component of our design. The design is divided into three major components, eDAG-Coder, Repairer, and Connector, each responsible for different functionalities. To run our service, we assume both one IPFS Cluster instance and one IPFS instance is running locally. The IPFS Cluster instance connects to some remote nodes, while the IPFS instance would be publicly available. When we upload a file into



(a) *UPLOAD*. (1) Users upload the file, and the file is further uploaded to local IPFS to create all Merkle DAG blocks. (2) The eDAG-Coder retrieves the blocks back to perform the entanglement. (3) The eDAG-Coder uploads the entanglement to the local IPFS, making it locally available. (4) The eDAG-Coder pins the entanglement via IPFS Cluster's API, making it remotely available. (5) Users obtain the file CID and the metafile CID back.



(b) *DOWNLOAD*. (1) Users input the file's CID and the metafile CID. The metafile CID is optional. (2) The eDAG-Coder triggers the building of the Lattice, based on the total number of data blocks. (3) The Lattice requests the desired CIDs from the IPFS network using the original Merkle DAG structure. (4) The Repairer uses BFS to find the desired blocks to repair if a block is missing. (5) Users retrieve the original file back.

Figure 4.3: Control flow among components when users upload to or download from IPFS Community.

IPFS, the files are converted into a DAG representation, as discussed in Section 2.3. Given the input as a DAG, the eDAG-Coder entangles the DAG with itself block-by-block, creating α eDAGs. The α is the parameter in AE codes, which is specified by users. The Connector connects IPFS Community to the running IPFS Cluster and IPFS instances. IPFS Community changes the internal logic of neither IPFS Cluster nor IPFS, instead, it uses them as tools to deliver the service. The Repairer component is useful when users would like to download a file. If some or all blocks of a file are missing, the repair process is triggered using AE codes. Figure 4.3 shows the detailed control flow when users upload to or download from IPFS Community, and how each module of IPFS Community is used.

4.2.2 User Interfaces

Our system offers two major interfaces to users, namely *UPLOAD* and *DOWNLOAD*. For the moment, it only supports command-line interfaces. *UPLOAD* allows users to upload a file. The file gets entangled and automatically pinned remotely. Users obtain the original file CID and a metafile CID as output. *DOWNLOAD* allows users to download a file after inputting the file CID and metafile CID. The output would be the original file that users request, if the file is available or can be recovered. The interaction between users and the system is illustrated in Figure 4.3.

In the case of *UPLOAD*, users could upload a file in a way that is very similar to how they upload a file to IPFS, namely they provide the path of the file to upload, and they finally retrieve back CIDs pointing to the file. The underlying mechanisms are different though. The key difference is that the DAG representation of the file gets entangled using AE codes, and the entanglement is uploaded to some remote nodes. To obtain the DAG representation first, instead of replicating what has been done by IPFS, we use IPFS to help us produce the results. We upload the file to the IPFS node and retrieve each block in sequence. As we mentioned earlier that uploading to IPFS only happens locally, therefore, the speed of the upload and retrieval is fast and the performance overhead is limited. After retrieving the blocks, we perform the entanglement at a block level using AE codes. The procedure is optional and can be controlled by user input. After generating the entanglement, we submit a pinning request via IPFS Cluster's API, which makes the entanglement remotely available. We upload the entanglement block by block. In the ideal case, we have a large enough community size that is larger than the total number of blocks, and each block would go to a different node. This offers better robustness against node churns. When one node goes offline, it only means that one entanglement block is gone. However, if we have a limited community size, one node going offline can mean that multiple entanglement blocks are lost. However, it is still in a much better position compared to the case where we upload the entire entanglement as a whole. Besides, to offer better load balance, and a higher probability of recovery, we upload blocks to IPFS Cluster nodes in a round-robin fashion. It means that one block is uploaded to i^{th} node, and the next block will be uploaded to $(i+1)^{th}$ node.

In the case of *DOWNLOAD*, users have to input at least the CID that points to the original file. This is just the same as how they get the file from IPFS network. If only the original CID is given as input, the IPFS Community will act the same as another type of IPFS CLI tool, i.e. without any repairing functionalities. To use the entanglement and activate the repairing process, users have to additionally input an entanglement CID, which points to a metafile. The metafile records all meta information that IPFS Community needs to repair the whole file, including how many blocks a file contains, how to find the entanglements stored inside the community, etc. The CID of the metafile is returned when users upload the original file at the initial place. The metafile is replicated among all communities.

After receiving the file CID and metafile CID, the application traverses the Merkle DAG of the file from root to leaves to retrieve the data back. If any of the data is missing, the recovery

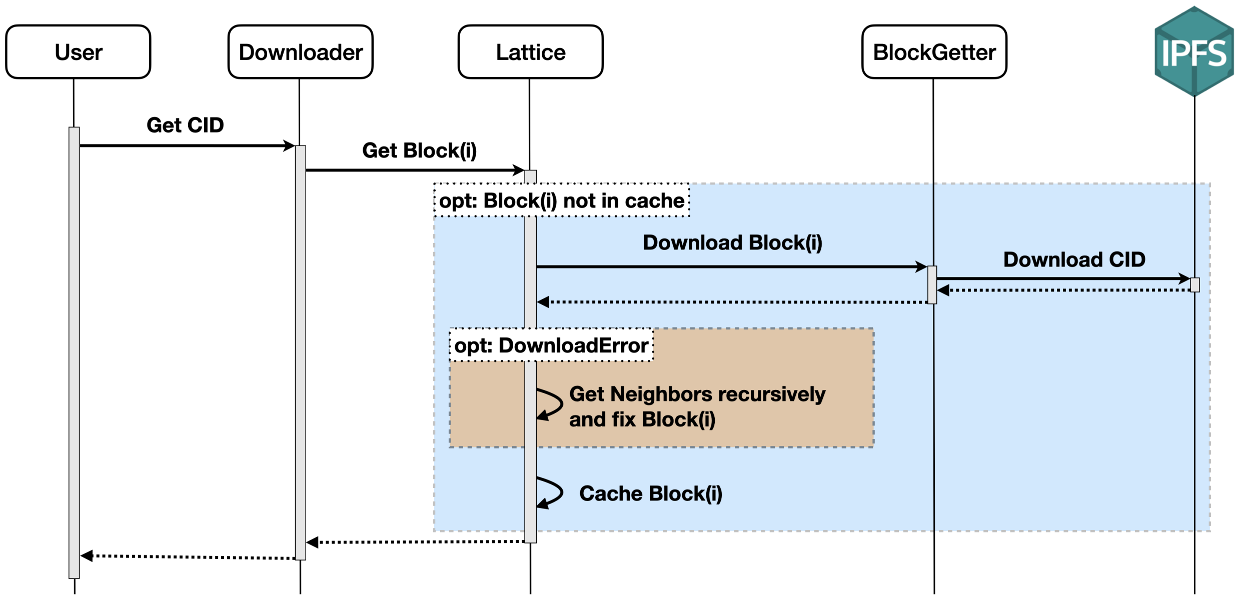


Figure 4.4: Sequence diagram when downloading with Lattice. (1) User submits a request with CID. (2) Downloader transfers CID to block index in lattice using a pre-generated map. (3) Lattice checks the cache. If no data is found, it tries to download the data from IPFS network. Further recovery will be triggered if the download fails. The data will be cached to be used to recover other missing blocks. (4) Lattice returns the block to Downloader and the data is given back to User.

process will be triggered automatically. As described in Figure 4.4, the application download process, acting as User, will start the request with the CID of the targeted block, and the CID will be parsed to index and given to Lattice. Lattice is a graph of data blocks and parity blocks connected in a way specified by AE codes, and it handles all the logic related to data downloading and recovery. The BlockGetter implements predefined interfaces for Lattice to call and manages all the communications between Lattice and IPFS node. It transfers the index back to CID by doing a map lookup and reports errors to Lattice if it fails to get the data from IPFS. All the communications between User, Download, Lattice, and BlockGetter happen locally inside our application, and BlockGetter communicates with IPFS node by HTTP requests. The explicit definitions of Downloader and BlockGetter enable us to separate the IPFS logic and the logic of Lattice, i.e. Lattice focuses on realizing AE codes, while Downloader and BlockGetter connect it to IPFS network. In this case, we can easily migrate Lattice to other decentralized networks, to the local filesystems, or even integrated it inside an IPFS node by just changing the Downloader and BlockGetter.

4.2.3 Dynamic Repair Strategy

One of the main advantages of AE codes is that it reduces the storage overhead compared to replication, and reduces the performance overhead compared to erasure codes. The advantage of lower storage overhead has been attested in Snarl [13]. We also have a separate Section 6.3.1 detailing our recovery rate given the data loss rate. However, to achieve the advantage of performance overhead reduction, we are limited by the fact that we have to do the repairing in order. If we do multiple repair searches in parallel, some searches are meant to be wasted, despite the fact that parallel searches would reduce the overall time delay if resources suffice. We would like to enjoy both the low time delay while not downloading too much unnecessary data. We propose to use a dynamic repair strategy, which switches between in-order and parallel recovery.

The IPFS Community's recovery process will be triggered when a requested block is missing. In this process, the two neighbors of the requested block in the same strand are needed. If those neighbors are missing, further recovery needs to be done for the neighbors recursively. Taking efficiency and the download overhead of the parity blocks into consideration, two mechanisms of repair are provided. The sequential repair checks the parity blocks strand by strand and stops immediately once both two neighbors in the same strand are available. In this case, it succeeds to minimize the overhead of downloading and storing parity blocks if only a little loss happens. The parallel repair checks all the strands at the same time with the help of multi-threading. It is able to make the recovery process succeeds in a short time especially when there is a heavy loss of data. Those two mechanisms are dynamically switched if a threshold of data loss is observed. The intuition is as follows: when the data loss is low, there is no need to do the parallel repair. In other words, every additional block you download will get wasted finally. However, if the data loss is high, it implies that you might sooner or later need to download all these blocks. Therefore, we will switch to the parallel repairing strategy, which minimizes the time delay, while still not increasing the overhead too much.

Chapter 5

Implementation

The code is available at `https://github.com/dedis/student_22_ipfs_alpha_entanglement_code`. The implementation is written in Go. The code base is divided into 6 sub-directories, `cmd`, `entangler`, `ipfs-connector`, `ipfs-cluster`, `util`, `test`, and `performance`. To run the program, we provide a command line interface in `cmd` to users, so that they could use IPFS Community in a similar fashion as the IPFS command line tool.

5.1 Implementation Details

5.1.1 Connector

We have two connectors, connecting to IPFS and IPFS Cluster separately. The connection to IPFS is via shell, while the connection to IPFS Cluster is via HTTP requests.

IPFS Connector handles all interactions with the local IPFS node. It relies on the 'go-ipfs-api' package developed by the IPFS team and uses HTTP GET or POST requests for communication. It supports uploading and downloading files for both existing files in the filesystem or pieces of data in memory. It also supports raw data operations so that the resulting blocks inside IPFS are without any metadata added automatically by IPFS. In addition, the connector is able to return an in-memory Merkle tree that mimics how the requested file is structured inside IPFS network, and the real file data is lazily loaded on usage.

IPFS Cluster Connector wraps all necessary operations against IPFS Cluster within Go functions. Inside each Go function, it submits an HTTP request to the local running IPFS Cluster instance using GET or POST. Supported APIs include PeerInfo, PeerLs, PinStatus, AddPin, and PeerLoad. All responses from IPFS Cluster contain a Newline Delimited JSON record, which could be parsed for detailed results. AddPin adds a specified CID to the global pinning set of the cluster. The IPFS Cluster connector will instruct where to pin each uploaded block. It adds pins in a round-robin fashion. To achieve this, the connector has to know beforehand about all available peers inside the cluster. Moreover, it has to maintain a state variable storing which peer to pin in the next round. The knowledge of all available peers is easy to acquire by simply submitting a status query into the IPFS Cluster. IPFS Cluster is configured so that one peer knows some peers inside the system if not all, and the query will propagate to all online peers.

5.1.2 Uploader

The Upload process starts with uploading the original file to IPFS and terminates if no entanglement parameters are specified. This part does not introduce any extra cost compared to IPFS CLI, which also uses HTTP requests to interact with the local IPFS node and perform the same upload action. If entanglements are wanted, the application will then retrieve the file Merkle Tree through IPFS Connector after uploading the original file and will flatten the retrieved Merkle Tree to an array in a pre-order sequence. The array will then be swapped to separate parent and child nodes so that if a parent node is missing, the repairer can still access its neighbors and repair it.

A **eDAG-Coder** will then be created, which handles the core logic of creating entanglements. The Merkle Tree node's data are loaded from IPFS and sent to the eDAG-Coder in a pipeline. The eDAG-Coder, with the help of previously cached parties on each strand, will perform the entanglement on the newly received data blocks, return the newly generated parity blocks, and replace the old parities in the cache with them. The eDAG-Coder only caches a constant number of parities based on the value of α, s, and p. Therefore, it only causes a limited memory overhead. Since chunks inside IPFS, especially for internal nodes and the leaves, vary a lot in their sizes, all blocks are zero-padded during entanglement to the larger size of the two input blocks.

After getting the generated parities from the eDAG-Coder in a pipeline, the uploader handles the logic of uploading parities to IPFS and constructing a metafile to store the information of the parities. All this information will be pinned in IPFS Cluster so that they can be accessed even when the original owner is offline. Considering the overhead in IPFS Cluster, all entanglements are pinned only once, while the metafile that is only one IPFS-chunk large is pinned on every node. The upload process directly returns when this information is uploaded to the IPFS network so that users can get the CID of their files and metafile immediately. The pinning operations are done in go routine, and users can optionally wait until its finishes to see whether the pinning succeeds.

5.1.3 Repairer

The Repairer component is triggered during downloading when the first missing of data blocks happens. A metafile generated during the upload process is downloaded at first so that the repairer knows the size of the total blocks of the original Merkel DAG file and the swapped node sequence. After that, a lattice is generated based on the information stored in the Metafile. The lattice is a sparse graph constituted by data and parity blocks connected in a way they were built in eDAG builder specified by AE codes. It is empty at the beginning, i.e. all the nodes are empty structures without any actual data.

The Recover process starts with the limited depth-first search. Starting from the missing data block as the root, the algorithm first focuses on a single strand to find the left and right parities to perform recovery. If any of these two parities is missing, the algorithm further expands the search ring to look up the node pairs that are needed to recover that parity. This process is done recursively until a valid recovery path from the first available pairs to the root is found. Otherwise, it terminates when a specified limited depth is reached and starts the same process on another strand. Once the limited DFS failed on all strands, a parallel BFS will be launched, as the failure indicates a heavy loss inside the lattice. It starts from the root and searches all the strands in parallel and stops all threads once the root is recovered. The algorithm is designed to explore each node only once in a single search, so if no valid path is found, it will still terminate by exploring all nodes inside the lattice.

The recovered blocks usually contain padding of zero bytes, which are removed after the recovery and re-uploaded to IPFS to prove the correctness. We concatenate the data chunks back to files by hand, which mimics how IPFS does with the leaf nodes' data. Since blocks of IPFS are special structures containing the data, we have to unmarshal the raw bytes to the structure before we could retrieve the real file data chunks stored inside the blocks.

5.2 Comparison with Snarl

Since the design of IPFS Community is inspired by Snarl, we would like to give a short analysis of what improvement it has on Snarl to apply AE code on decentralized file systems.

Regarding to system design, Snarl focuses on applying AE code on Swarm, and their designs are highly related to this specific network. For example, they build the Merkle tree locally based on the principle of Swarm, and the interaction of the Swarm connectors is deeply embedded in their Entangler and Repairer. On a different design focus, we want our eDAG-Coder and Repairer to be portable to other decentralized file systems or to be integrated inside the IPFS node. To achieve that goal, we modularized the system to separate the logic of eDAG-Coder and Repairer from the network connector, and the system can therefore easily works on other decentralized file systems by just changing the connector module. In addition, IPFS Community does not create a Merkle tree itself. Instead, it hands the work to the underlying file system, so that IPFS Community will still work on other decentralized file systems or if IPFS changes its way of creating the Merkle tree.

To improve efficiency, we pipeline the whole upload process, including how we create entanglement, upload parities, and do the pinning. Snarl only applies pipeline in its Entangler, but the original file which could be very large must be fully loaded before starting the entanglement, and the parities will only be uploaded when the full entanglement is generated. We also add the dynamic switch between sequential recovery and parallel recovery to balance download overhead and efficiency. In contrast, snarl focuses more on minimizing download overhead and only provides sequential recovery.

Chapter 6

Evaluation

In this chapter, we discuss in detail how we set up the experimental environment, and the methods that we apply to test the correctness of the implementation, together with different aspects of the system, including performance, security, and reliability. For AE codes, we are using parameters $\alpha = 3, s = 5, p = 5$. All performance tests are repeated 100 times, and the average values are reported.

6.1 Environment Setup

The experiment is conducted on MAC OS, MacBook Pro with 2.6 GHz 6-Core Intel Core i7. To allow us to have multiple running instances of IPFS Cluster, we utilize docker to simulate the node topology. We use the latest version of both `ipfs/go-ipfs` and `ipfs/ipfs-cluster` containers. More precisely, `ipfs/go-ipfs@sha256:803fac58ba15bd763b97a1ce17bca57f75348 f2088d384a908b77e8540f35560` and `ipfs/ipfs-cluster@sha256:dfc07e42de39512e02c4b80ad9 fb8515841f3c5e465db99276359f84c8989a44`. For the testing purpose, we run concurrently 10 IPFS Cluster nodes with 10 accompany 10 IPFS nodes. Each of the 10 IPFS Cluster nodes has a port accessible from the host machine, allowing us to interact with every individual. All 10 IPFS Cluster nodes are active users, namely, everyone can perform the entanglement, and add or remove pins. The connection among the IPFS Cluster nodes is private, where they share a secret. The underlying IPFS is public, meaning that it can either download content from the Internet or be accessible from the outside world as long as our machine has a public interface.

6.2 Correctness

To test the correctness of the individual function, module, and system, we write both unit tests, module tests, and system integration tests.

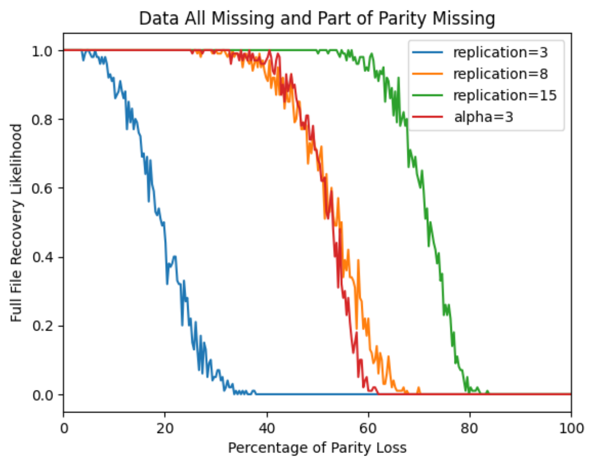
Unit tests were done on each part of the application, including IPFS Cluster connector and block getter that is used by the repairer to download data from IPFS. Regarding IPFS Cluster connector, each API was tested to see whether there were unexpected errors and whether the expected action was performed. In contrast, block getter implements an interface with simple APIs but has complex logic behind it. In this case, we used different test cases from simple to complex to prove their correctness.

Module tests focus on eDAG-coder and the repairer. For eDAG-coder, we slightly modified snarl's code and separated its entanglement mechanism from other functionalities. We use the modified code to generate the entanglement on the same file and compared the output of snarl with ours. The way how snarl produces the entanglement is a little bit different from ours, so the result was sorted before comparison. We did the tests on files of different sizes, from small files with only a limited number of blocks to large files with many blocks, to ensure the eDAG-coder works fine for edge cases. For the repairer, different data loss situations were simulated, including single data loss, multiple data loss, separated data and parity loss, continuous data and parity loss, whole data loss, etc. All those situations were simulated multiple times on different positions inside the lattice graph to avoid edge case failure.

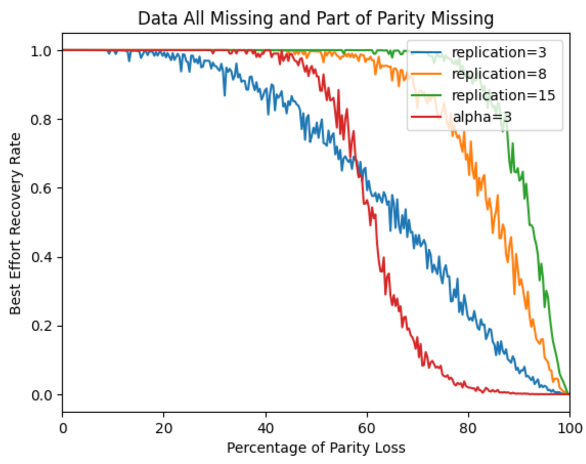
At last, we tested the whole upload and download processes using different sizes of files and designed different cases of data loss regarding downloading. The integration tests ensured the correct functionalities of the system.

6.3 Performance

To evaluate the performance of the system, we consider two main use case scenarios, i) all data blocks are missing, but parity blocks are all available; ii) all data blocks are missing, and some portion of the parity blocks are also missing. These two scenarios would be most commonly seen in our system. For the first one, it could be the case that users upload the file on one machine, and the machine goes offline. Later, the user would like to have the file back, but he would not be able to download the file directly, because the original file is only available on the machine where users upload the file. The only way users could recover their files is via entanglement blocks, which are stored on remote nodes. The second one extends the first use case by the fact that some of the remote nodes could go offline as well. For these two scenarios, we compare our solutions to replication, to check whether our design outperforms the de-facto solution and if yes, by what percentage.



(a) Full file recovery likelihood. For example, recovering the full file 10 times out of 100 tries contributes to a y-value equal to 0.1.



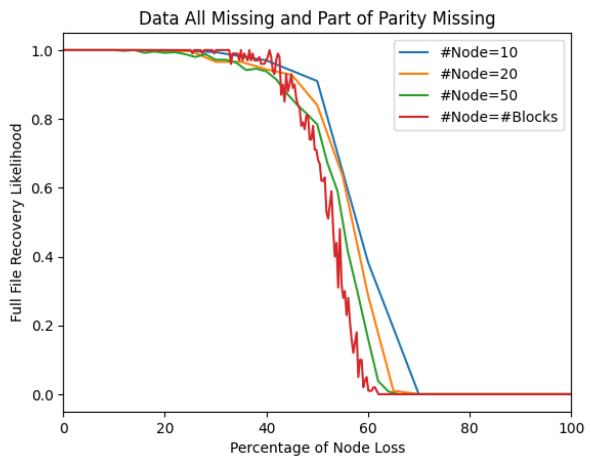
(b) Best effort recovery ratio. For example, recovering 10 blocks out of 100 blocks contributes to a y-value equal to 0.1.

Figure 6.1: The comparison between AE codes and replication in both full file recovery likelihood and best effort recover ratio. All data blocks are missing, while some parity blocks are missing. Both x-axes stand for the percentage of parity loss. 50% parity loss means 50% entanglement blocks are missing for AE codes, and 50% replicated blocks are missing for replication.

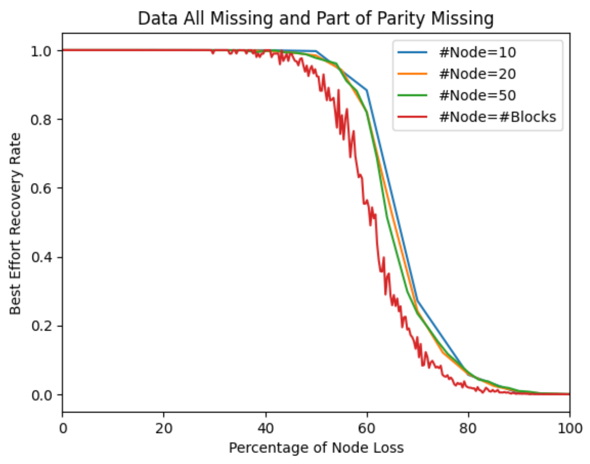
Apart from the two main scenarios, there are also some other dimensions to consider, one would be the number of nodes inside the system. The number of nodes inside a distributed system is always an interesting aspect, it can influence performance in diverse ways. The other would be the size of the uploaded files. The size of files influences the size of the lattice. The larger the file, the larger the lattice would be. The number of internal nodes could also have some implications on how well our system could perform. We discuss the above three dimensions separately in the following part.

6.3.1 Reliability

To attest to the reliability of the system, we compare AE codes to replication. For the first scenario, some data blocks are missing, both AE codes and replication can recover all data blocks. A more interesting scenario would be the second one, which is all data blocks are missing while some "parity" blocks are missing. The party blocks are substituted by replicated blocks, in the case of replication. To begin with, we assume that every loss of block is independent of each other, namely every block is stored on a different node. IPFS uses $256\,\mathrm{kB}$ as the default block size. For the following experiments, we are using files of size $25\,\mathrm{MB}$, which is equivalent to 101 blocks in IPFS. Since all data blocks are missing, we are only considering the "parity" blocks. With $\alpha = 3$, we have the same storage overhead as $replication = 3$. Here $replcation = 3$ means 3 replication of each individual data block is stored inside the IPFS Cluster.



(a) Full file recovery likelihood. The definition is the same as that in Figure 6.1a



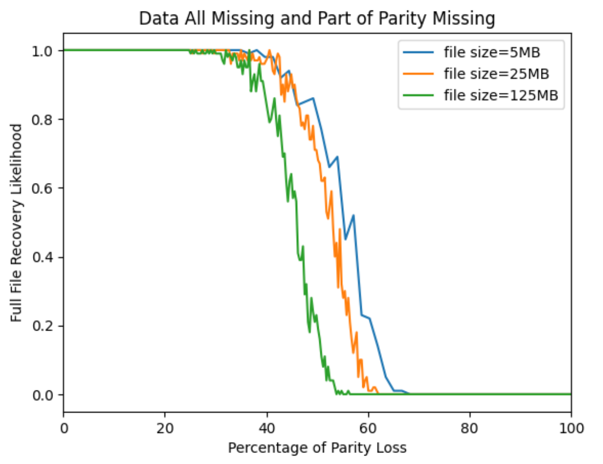
(b) Best effort recovery ratio. The definition is the same as that in Figure 6.1b

Figure 6.2: The influence of the number of nodes on file reliability. All data blocks are missing, while some parity blocks are missing. Both x-axes stand for the percentage of node loss. 50% node loss means 5 nodes missing when $\#node = 10$, 10 nodes missing when $\#node = 20$, and $\#Blocks = 303$.

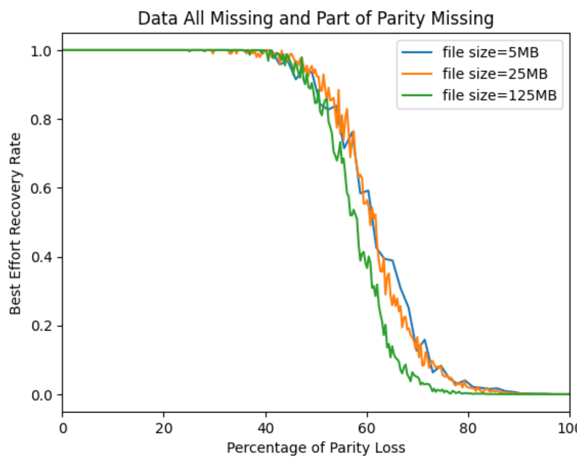
Figure 6.1 shows the results comparing AE codes and replication. The most common use case that we are facing in real life is the results shown in Figure 6.1a. The use case is that we want to recover the file as a whole. When we are talking about the best effort recovery ratio, we are indeed trying to recover as many parts of the file as possible, which is less common. From the results in Figure 6.1a, we could observe that AE codes outperform replication with the same storage overhead, which is $alpha = 3$ and $replication = 3$. With $alpha = 3$, it actually achieves very similar results as $replication = 8$. The storage overhead improvement is around $2.7\times$. Figure 6.1b shows a different trend. We could see that AE codes have a slight advantage over replication when the parity loss is less than 60%. When the parity loss is high, replication indeed performs better. The intuition behind this is simple. With replication, even though most of the replicated blocks are missing, the existing replicas are still valid to recover the original blocks. With AE codes, most of the parity blocks missing means that it is hard to build a connected strand. As a result, the number of recovered blocks is low.

Different Numbers of Nodes

The number of nodes is a core aspect of a distributed system. The same algorithm might behave very differently when there are a handful of nodes and when there are thousands or millions of nodes. In this section, we would like to see how the number of nodes influences the two metrics in Figure 6.1. Ideally, the algorithm should perform similarly no matter what the number of nodes is.



(a) Full file recovery likelihood. The definition is the same as that in Figure 6.1a



(b) Best effort recovery ratio. The definition is the same as that in Figure 6.1b

Figure 6.3: The influence of the file size on file reliability. All data blocks are missing, while some parity blocks are missing. Both x-axes stand for the percentage of parity loss. 50% parity loss means 50% of all entanglement blocks are missing. The number of entanglement blocks is monotonically increasing when the file size increases.

Figure 6.2 shows the results on how the number of nodes influences file reliability. We assume that each node is independent of the other, and the loss of one node does not influence the other. As we could see, when the number of nodes is low, it tends to have a slightly better performance compared to the case when the number of nodes is high. We suspect the results come from the fact that we perform round-robin parity block distribution. For the round-robin distribution, we distribute α strands of the single block in sequence to the next α neighbors. The round-robin pattern is well observable when the number of nodes is low and diminishes as the number of nodes increases. We believe the round-robin fashion together with the independence assumption that we make together contribute to better performance when the number of nodes is low.

Different File Sizes

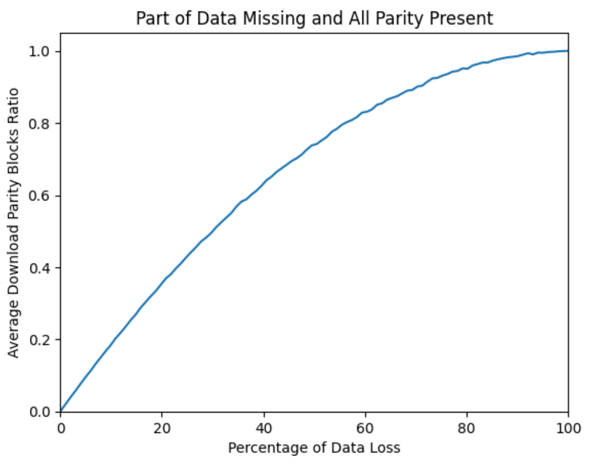
All the above experiments are conducted with $file\ size = 25\,\mathrm{MB}$. However, users are uploading files of different sizes in reality. When the file sizes are different, the number of blocks in Merkle DAG that represents the file differs. More precisely, the number of blocks is monotonically increasing when the file size increase. This comes from the fact that IPFS uses a fixed-size block internally. When the number of blocks is different, the lattice would look different with the same parameters for AE codes. The lattice would be much wider if the number of blocks is higher. The topology of the lattice is the key to the recovery process.

Figure 6.3 shows how file sizes influence file reliability. The results suggest a negative trend. The larger the file size, the less likely that you will be able to either recover the whole file or recover partial files. The reason behind this comes from two factors. First, the larger the files, the wider the lattice would be. AE codes are irregular codes. The fault tolerance of irregular codes is not fixed, but it depends on the pattern of the data loss [10]. There are some irrecoverable patterns that prohibit users from recovering the whole file. The possibility of the occurrence of these patterns increases with the lattice getting larger. Therefore, if we have a $5\times$ larger lattice, we supposedly have $5\times$ more likely on average to have an irrecoverable pattern. Second, the number of internal nodes is different. When $filesize = 5\,\mathrm{MB}$ and $filesize = 25\,\mathrm{MB}$, there is only one internal node, the root. The irrecoverability of the root will render the cascading failure of recovering all its descendants. When $filesize = 125\,\mathrm{MB}$, there are 4 internal nodes, including the root. The irrecoverability of any of these 4 nodes will render the cascading failure of recovering all its descendants. We need at least to recover all 4 internal nodes to proceed with full file recovery. In Figure 6.3b, we could observe the difference between $filesize = 5\,\mathrm{MB}$ and $filesize = 25\,\mathrm{MB}$ is small. The reason behind this could be that they have the same number of internal nodes.

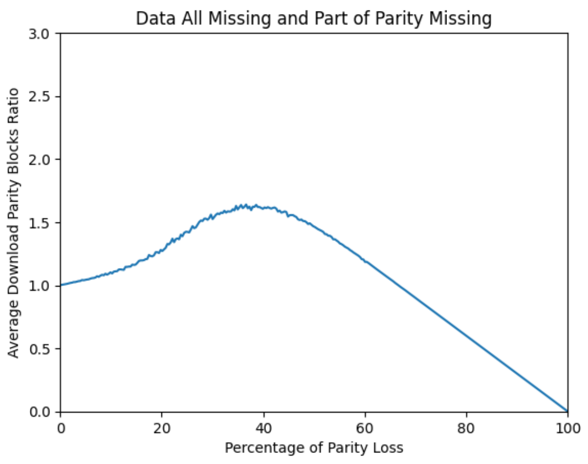
6.3.2 Download Overhead

We measured how AE codes influence the download overhead, i.e., how many blocks we have to download from the Internet. It is worth measuring because network bandwidth could be the bottleneck to recovering the file. In our first scenario, we have some data blocks missing only, we need to download some parity blocks to recover the data blocks. In our second scenario, all data blocks are missing and some parity blocks are missing as well, we also need to download some parity blocks.

Figure 6.4 shows curves denoting the download overhead for our two scenarios. Figure 6.4a illustrates the download trend with the increase in data loss. In the beginning, the slope is around 2. This comes from the fact that AE codes use XOR two blocks for recovering. As the data loss increases, the slope drops and the curve is upper-bounded by 1. This comes from the fact that we need to only download one strand to recover data only loss. The number of blocks inside a strand is the same as the total number of blocks in the original file. If we are using replication instead of AE codes, the curve will turn into a straight line, with the same starting and ending points. The download overhead is linear increasing with the data loss, and the slope is 1. Figure 6.4b shows how the download overhead relates to the parity loss. The curve starts at point $(0, 1.00)$. This is a continuation from Figure 6.4a's endpoint, where all data is missing and all parity is present. With the parity loss increasing, we need to download more parity blocks. This comes from the fact that we need to use parity blocks to repair parity blocks, and finally, repair the original data blocks. The curve summits around 40% of parity loss, with around 1.7 download overhead. The curve then drops roughly linearly as it is dominated by the parity loss. The high parity loss prevents us from downloading more parity blocks, thus, the download overhead is dropping monotonically. If we are using replication instead of AE codes, the download overhead



(a) Partial data blocks missing. The download overhead curve is upper-bounded by 1. We have to download at a maximum of one strand of parity to recover the whole file.



(b) All data blocks missing and partial parity blocks missing. The download overhead curve is upper-bounded by 3. The maximal parity blocks you could have is α strands, where $\alpha = 3$ in our case.

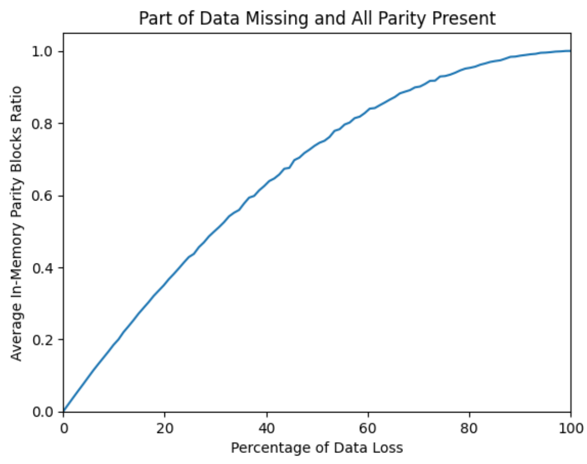
Figure 6.4: Download overhead of AE codes. Both y-axes stand for the percentage of downloads compared to the total number of data blocks. $y = 0.5$ means that we have to on average download 50 blocks if the original file has 100 blocks.

will be upper-bounded by 1. Similar to the trend shown in Figure 6.4b, it will gradually drop to the same endpoint.

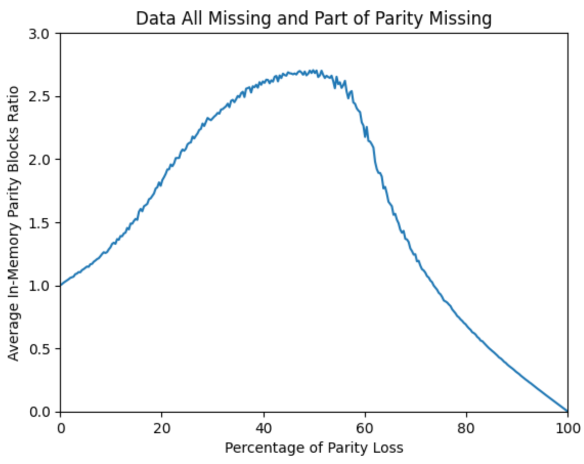
6.3.3 Memory Overhead

The memory overhead is different from the download overhead. To recover the data blocks, we have to reconstruct the lattice. Some parity blocks are missing and could not be downloaded, but it is possible to recover them and store them in memory. Therefore, the memory overhead is presumably much larger than the download overhead. For example, you download 5 parity blocks, but using these 5 parity blocks, you could recover 7 parity blocks. These 7 parity blocks have to be stored inside memory, until the termination of the recovery process. To be precise, the memory overhead is lower-bounded by the download overhead, i.e., the memory overhead is at least the same as the download overhead. In the case of a super large file, if the memory consumption exceeds the RAM capacity, it could slow down the process quite a lot, involving page swapping. Therefore, it is interesting to see how large the memory overhead would be in addition to the download overhead.

Figure 6.5 shows the results of memory overhead. When there is only data loss, we could see that the trends in Figure 6.4a and Figure 6.5a are very similar. When there is only data loss, the parity blocks are only downloaded if needed and stored in memory. There is no parity to repair, therefore, the download overhead curve matches the memory overhead curve. When all data



(a) Partial data blocks missing. The memory overhead curve is upper-bounded by 1. We have to keep in memory at a maximum of one strand of parity to recover the whole file.



(b) All data blocks missing and partial parity blocks missing. The memory overhead curve is upper-bounded by 3. The maximal parity blocks you could have is α strands, where $\alpha = 3$ in our case.

Figure 6.5: Memory overhead of AE codes. Both y-axes stand for the percentage of in-memory blocks compared to the total number of data blocks. $y = 0.5$ means that we have to on average store 50 blocks in memory if the original file has 100 blocks.

blocks are missing and some parity blocks are missing, the trend is quite different as shown in Figure 6.4b and Figure 6.5b. The memory overhead is noticeably much higher than the download overhead. The peak memory usage reaches around 2.7. While the download overhead's peak is around 1.7. The gap between the two comes from the fact that there are some recovered parity blocks inside the lattice.

6.4 Security

IPFS Community is a system built on top of IPFS and IPFS Cluster. Therefore, it is unreasonable to talk about its security if we ignore the security of IPFS and IPFS Cluster.

IPFS is a fully decentralized file system. This nature solves the trust problem that centralized systems face and makes IPFS less vulnerable to attacks or censorship compared to centralized ones. It uses cryptographic hashes to identify files, which ensures file integrity. It also provides confidentiality by encrypting files and enabling access control. However, it also faces several threats, for example, the attacker might make use of hash collisions to generate a valid hash for a malicious file, or they may control an IPFS node and tamper with the content being shared through that node. In addition, IPFS does not provide an incentive mechanism for nodes to pin files, which means there might be an availability problem with IPFS files. Users might need to keep their IPFS node always online to make their files available, or they may make use of third-party services, which again introduces the trust problem, In addition, since most files are not widely pinned by others, there would also be a reliability problem once the original copy of the file is corrupted.

IPFS Cluster offers a solution to increase file availability by forming a group of nodes that pin files for each other. It runs a consensus protocol so that all nodes have the same view of the global pinning set. Others will help to pin the content when the original pinning node goes offline, and the pinned content will still be accessible. However, this solution does not solve the incentive problem, and the cluster assumes all nodes are honest and will perform what is requested to be done. There is no defense mechanism if a node in the cluster refuses to pin files for other nodes.

Our solution IPFS Community relies heavily on IPFS and IPFS Cluster. It inherits all advantages IPFS and IPFS Cluster have, such as no central authority, censorship resistance, file integrity, fairness, etc. In addition, it also provides plausible file availability and reliability through AE codes and IPFS CLuster with relatively low storage overhead. However, it does not solve the problem IPFS has, such as hash collision or the break of on-path nodes. It does not solve the incentive problem and is more useful in scenarios where family and friends form a small cluster or a company owns its own cluster.

Chapter 7

Conclusion

In this paper, We present IPFS Community, a novel architecture built on top of IPFS and IPFS Cluster, that manages to improve IPFS file availability. It makes use of AE codes and successfully provides better file reliability with less storage overhead compared to the native IPFS solution of replicating files across IPFS nodes. Users are able to customize the redundancy factors for their own needs and balance between the redundancy level and the storage overhead. In addition, the architecture is designed with pluggable modules, and can, therefore, be easily migrated to other distributed file systems by adjusting the connector component.

Extensive experiments are done to evaluate the effectiveness of the system. The storage overhead of IPFS Community is reduced by $2.7\times$ compared to that of replication, given the same file redundancy level. We also examined the bandwidth overhead and memory overhead and concluded that they are bounded by $1.7\times$ and $2.7\times$ accordingly, for $\alpha = 3$.

Future work

Our solution IPFS community is the beginning step of applying AE codes to IPFS. Several future improvements are possible.

First, this code is a proof of concept implementation to see how AE code helps in IPFS. Therefore, only a simple switch between sequential and parallel recovery is implemented. The repairer now views each block repair separately, and the dynamic switch happens individually in each repair process with a given switch depth. However, the heavy data loss situation is actually a global situation inside the lattice, and it would be better if the dynamic switch could be controlled globally if a certain amount of missing happens. In addition, experiments could be done to find the best switch time to balance between the throughput efficiency and the parity download overhead.

Second, the recovered data is now re-uploaded to the IPFS network to obtain its CID, which is compared with the expected CID to ensure recovery correctness. However, in the original design, we would like to offer an option for users to decide whether they want to do the re-uploading. Hence, it would be better if it could compute the CID locally in the future version of the IPFS Community,

Third, although the IPFS Community does not solve the incentive problem directly, it provides a possible way to do it. This could be done by creating an entanglement with multiple files belonging to different people. In this case, the entanglements could be pinned by several nodes and the reliability of files could be maintained by several parties. To obtain files of different owners, the entanglement components need to be integrated into IPFS nodes inside its DHT. Though the IPFS Community is currently an application built on top of the IPFS network, integration is quite possible as the eDAG-coder and the repairer are reusable by just replacing the IPFS connector and block-getter components.

Bibliography

[1] Fernando Almeida, José D Santos, and José A Monteiro. "e-commerce business models in the context of web3. 0 paradigm". In: *arXiv preprint arXiv:1401.6102* (2014).

[2] Russell Belk, Mariam Humayun, and Myriam Brouard. "Money, possessions, and ownership in the Metaverse: NFTs, cryptocurrencies, Web3 and Wild Markets". In: *Journal of Business Research* 153 (2022), pp. 198–205.

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

[4] Juan Benet, David Dalrymple, and Nicola Greco. "Proof of replication". In: *Protocol Labs, July* 27 (2017), p. 20.

[5] Nazanin Zahed Benisi, Mehdi Aminian, and Bahman Javadi. "Blockchain-based decentralized storage networks: A survey". In: *Journal of Network and Computer Applications* 162 (2020), p. 102656.

[6] Longbing Cao. "Decentralized ai: Edge intelligence and smart blockchain, metaverse, web3, and desci". In: *IEEE Intelligent Systems* 37.3 (2022), pp. 6–19.

[7] Yongle Chen, Hui Li, Kejiao Li, and Jiyang Zhang. "An improved P2P file system scheme based on IPFS and Blockchain". In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 2652–2657.

[8] Erik Daniel and Florian Tschorsch. "IPFS and friends: A qualitative comparison of next generation peer-to-peer data networks". In: *IEEE Communications Surveys & Tutorials* 24.1 (2022), pp. 31–52.

[9] Vero Estrada-Galiñanes, Ethan Miller, Pascal Felber, and Jehan-François Pâris. "Alpha entanglement codes: practical erasure codes to archive data in unreliable environments". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2018, pp. 183–194. DOI: `10.1109/DSN.2018.00030`.

[10] Kevin M Greenan, Ethan L Miller, and Jay J Wylie. "Reliability of flat XOR-based erasure codes on heterogeneous devices". In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE. 2008, pp. 147–156.

[11] Barbara Guidi, Andrea Michienzi, and Laura Ricci. "Data persistence in decentralized social applications: The ipfs approach". In: *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE. 2021, pp. 1–4.

[12] Peter Lorentzen. “China’s strategic censorship”. In: *American Journal of political science* 58.2 (2014), pp. 402–414.

[13] Racin Nygaard, Vero Estrada-Galiñanes, and Hein Meling. “Snarl: entangled merkle trees for improved file availability and storage utilization”. In: *Proceedings of the 22nd International Middleware Conference*. Middleware ’21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 236–247. ISBN: 9781450385343. DOI: 10.1145/3464298.3493397. URL: https://doi.org/10.1145/3464298.3493397.

[14] James S Plank. “A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems”. In: *Software: Practice and Experience* 27.9 (1997), pp. 995–1012.

[15] Yiannis Psaras and David Dias. “The interplanetary file system and the filecoin network”. In: *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE. 2020, pp. 80–80.

[16] Viktor Trón. *The Book of Swarm - v1.0 pre-release 7 - worked on November 17, 2020*. https://www.ethswarm.org/The-Book-of-Swarm.pdf. Accessed: 2022-09-21.

[17] Shermin Voshmgir. *Token economy: How the Web3 reinvents the internet*. Vol. 2. Token Kitchen, 2020.