High Performance Distributed File System Based on Blockchain

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Ajinkya Rajguru

May 2023

The Designated Project Committee Approves the Project Titled

High Performance Distributed File System Based on Blockchain

by

Ajinkya Rajguru

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

ABSTRACT

High Performance Distributed File System Based on Blockchain

by Ajinkya Rajguru

Distributed filesystem architectures use commodity hardware to store data on a large scale with maximum consistency and availability. Blockchain makes it possible to store information that can never be tampered with and incentivizes a traditional decentralized storage system. This project aimed to implement a decentralized filesystem that leverages the blockchain to keep a record of all the transactions on it. A conventional filesystem viz. GFS [1] or HDFS [2] uses designated servers owned by their organization to store the data and are governed by a master service. This project aimed at removing a single point of failure and makes use of participating users' machines to store data. The implemented file system uses a distributed hash table to evenly store data on multiple machines and efficiently look up, track, maintain, and persist the data. The data stored on the file system is made easily and readily accessible to a user ensuring its soundness. Finally using smart contracts enabled incentives and increased the integrity and reliability of storing data by recording all the transactions. The project functions autonomously by coordinating multiple participant machines in the network to store data and monetize unused storage space on the machines. It supports the basic functionalities of a filesystem which include storing, deletion, and reading a file as desired. The filesystem's performance was tested with files of different sizes and a variable number of nodes. The current file system is able to store files as big as 100 MB within 12 seconds and 300 MB in less than 25 seconds. This is significantly faster than Storj [3] which nearly takes more than a minute for a similar operation. The tests also provided observations on the average gas consumed during transactions.

Keywords - Distributed Filesystem, Blockchain, Chord, DHT, Smart Contracts

ACKNOWLEDGMENTS

I would like to express my gratitude to Professor Chris Pollett for his invaluable support, patience, and feedback. He has played a key role in my progress over the past two semesters. The weekly brainstorming and white boarding sessions with him have been a catalyst in the development and refinement of my project. Thanks for having my back, for always having an answer to all my questions and for guiding me through.

I would also like to recognize the department of Computer Science at SJSU for providing comprehensive courses and resources which helped me achieve the goals of my project. Special thanks to Prof. Ben Reed for introducing me to Distributed Systems, an area of the field which sparked my interest and drove the trajectory of my graduation project.

I am very thankful to my committee members and all other faculty members for their knowledge and expertise. Lastly, I would like to acknowledge my peers and family members for keeping my spirits high and supporting me throughout this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. Introduction

Nowadays with the amount of data consumed and generated by one person it is important to have multiple storage solutions for our changing needs. An average individual stores nearly 500 GB of data on the cloud [5] which includes photos, videos, text documents, music, etc. With evolving internet technologies and availability of cellular plus wireless networks, cloud storage is slowly becoming the conventional way of storing our data.

Cloud storage allows users to access and store their data from any device by paying a minimum fee for the service. Because of this convenience, many personal devices have ample storage space which goes unused. On the other hand, some users run out of storage space on their devices as well as on their free or paid cloud subscriptions. A user with unused storage on their personal computer could rent out some space for other users in need of storage space. This would be a convenient solution for people with a plethora of data to store and for computer owners with multiple gigabytes of vacant hard drives. This project implements a high-performance distributed filesystem which can enable users to rent out storage spaces to others for some incentives.

A distributed filesystem stores data on commodity servers located in different locations, allowing access using a computer network. It unleashes the potential of data by enabling users to share files, remotely access them and be available at all times. While conventional file systems such as Google File System and Hadoop File System offer numerous benefits and a wide range of applications, they still suffer from some drawbacks that cannot be avoided. Huang, et al.[6] have pointed out that these systems lack stability, auditing, and incentive mechanisms and require high-performance master hardware, which can be a single point of failure.

Decentralized peer-to-peer storage systems are emerging as the next generation of distributed filesystems, overcoming a few drawbacks mentioned above. A decentralized system and cryptocurrency can significantly enhance the filesystem by adding an incentive layer, robust auditing, and increased data integrity, among other benefits.

The main aim of this project was to have a distributed filesystem as mentioned above but also be decentralized and include incentives making it fully autonomous, which can run by creating its network of machines. For this purpose, the project uses distributed consistent hashing protocol — Chord [4] along with a smart contract on the Ethereum blockchain using Solidity and gRPC for socket-based interaction between participating machines. This project report follows the following sequence: The upcoming chapter covers the preliminary work and research carried out during the first half of the project and its results in brief. The next chapter then describes the related work in the peer-to-peer systems and file storage field. Following related works, the system architecture is explained along with the implementation strategies used. The results and experiments chapter then overviews the performance testing and other observations drafted from the functioning file storage system. Lastly, the future scope of the project has been mentioned along with the concluding remarks.

II. Related work

Stoica, et al. [1] explain multiple methods to implement Chord along with their advantages and disadvantages. This project implements the modified version of Chord to support concurrent joining and failure of nodes. Unlike the current project, The Swarm team [7] and Juan Benet [8] use the Kademlia Distributed Hash table for efficient lookup and coordination of nodes in a fault-prone system. Benet [8] expands that a Kademlia network with nearly 10000000 nodes only requires 20 hops for a desired lookup. This comes down to an average interaction with $log_2(n)$ nodes. Gnutella and BitTorrent [9] make use of Kademlia for a network of 20 million nodes.

The conventional BitTorrent algorithm used a tit-for-tat strategy which, according to Bram Cohen [10], increases the system's robustness with uniform resource utilization of participating peers. Bram [10] talks about the redistribution of costs of uploading file bits to the downloaders when the downloaders also take part in uploading pieces of the file to other peers requesting the file. This strategy reduces the bandwidth cost that the users bear but does not allow for any scope for direct monetizing. As opposed to this, the currently implemented peer-to-peer filesystem depends on blockchain to add an incentive layer, enabling users to earn real cash for their services.

In the Interplanetary filesystem [8], Merkle trees are used to create a content ID and as a checksum for the file. The Merkle tree-based checksum and identifier have been applied in the current system for efficient lookups and to improve the system's integrity. The current system uses a calculated root hash of a file for lookup, which is identical to IPFS's [8] content ID. Like the implementation proposed in this report, Storj [3] uses gRPC to establish peer-to-peer communication.

III. Preliminary works

The project's initial phase was mainly focused on research work and trying to implement a few essential components of the system. This chapter briefly describes the work carried out during the first half of the project, along with its applications and benefits to the outcome. The chapter includes a description of a basic implementation of Chord, performance testing of existing filesystems, the filesystem structure, and a smart contract implementation using Solidity

3.1 Chord Implementation

To develop a storage system that makes use of blockchain and does not have a single point of failure, it was essential to have a lookup protocol that could regulate the whole flow of the system. In a conventional filesystem like GFS or HDFS, a master service has many responsibilities, including the distribution of data, storing metadata, and working around system failures. Chord is a highly scalable distributed hash table that can also work with constant changes in the network of machines. Chord's primary objective is to map a key to a specific node in a network with minimal effort, even when there are multiple nodes. Chord is designed to be flexible and can adjust to network changes, such as adding or removing nodes.

For the first deliverable, a simple Chord Distributed hash table was implemented in Java to get a deep understanding of all the underlying protocols and learn the fundamentals of its functioning. This implementation consisted of the Chord ring and a client to store key-value pairs on the ring. This implementation could only cater to a few servers ranging between 9000-9127 port numbers. Every server consisted of a finger table with seven entries pointing to the next nodes.

This Chord ring could work with the addition of new nodes and store the keys as desired. This makes the basis of the project as the final project is fully based on a scaled-up version of this Chord with numerous enhancements and improvements.

3.2 Testing Existing Filesystems

The second task was mainly focused on identifying currently existing storage systems that use a crypto incentive layer and are fully decentralized. Four systems were shortlisted for testing after considering their feasibility, similarity to the proposed approach, and availability – IPFS (Interplanetary filesystem) [8], SWARM [7], Storj [3], and SIA [11]. All the systems were tested based on some common criteria and a few exclusive ones based on their different features and purposes. The four filesystems were tested to check the time required to store and retrieve some small and large files. These results helped to determine the expected performance that can be desired from the project. Different features and functionalities offered by the systems helped the project determine the requirements for the filesystem to be developed and to set the project's scope.

3.3 Filesystem Structure

It was essential to have a predefined template for storing files to use the distributed hash table nature of the Chord infrastructure. In a decentralized store, a master can be responsible for tracking all the metadata, tracking files, and nodes serving the files. But in this case, we need a predefined structure to divide a file into multiple blocks, name them, and take care of their reconstruction after retrieval. The third task consisted of developing a filesystem structure inspired by Merkle trees. Before storing any file, it was divided into small chunks of 4 KB each. A chunk object consisted of the data, the hash value calculated from the data, and the root hash value formulated by concatenating all other chunk hashes. Each chunk also stored the last index of the data array, which was necessary for file reconstruction. The root hash helped in determining the integrity of the file after retrieval. This helped create the final product's client-side functions, which were made compatible with the Chord ring.

3.4 Smart Contract Using Solidity

The fourth task was to get acquainted with Solidity and the fundamentals of a Smart contract. The deliverable included learning a new programming language, its data types, and various development environments. To initiate the process, this deliverable aimed to develop a simple gambling game that would run on a test network provided by the Remix IDE. This game allowed anonymous users with some amount of ether to participate for a small fee. The gambling starts after 5 participants enter the game, and finally, two winners are selected based on the randomized algorithm. The contract deployer acts as a dealer and receives some compensation for the gas required. This experience helped develop the smart contract for the filesystem and decide upon the working environment, testing tools, and frameworks required.

IV. Architecture and implementation

The file storage system mainly comprises four modules – The Chord ring structure of server nodes, the File Storage layer, a compatible client implementation, and the smart contract layer. The below figure gives an overview of how the components are placed in the current system architecture.

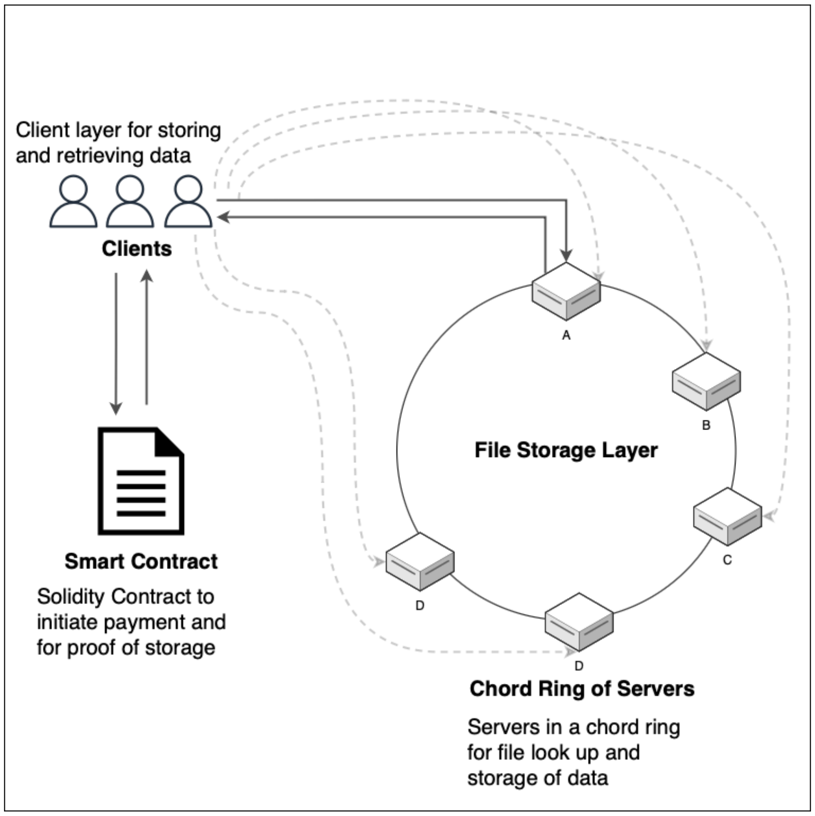

Client layer for storing and retrieving data
Clients
Smart Contract
Solidity Contract to initiate payment and for proof of storage
File Storage Layer
A
B
C
D
D
Chord Ring of Servers
Servers in a chord ring for file look up and storage of data

Figure 1: Filesystem Architecture

The client provides an interface to users for storing, retrieving, and deleting desired files. The smart contract allows clients to pay for their stored data blocks and ensure that the servers requesting ether safely store the respective block. The Chord servers are used for the lookup of nodes to navigate or search for a desired data block. The file store is embedded in the servers participating in the Chord ring.

This chapter elaborates on the architecture and implementation of every layer, starting with the Chord, followed by the File Storage layer on the node servers, the client layer, and the Smart Contract Solidity layer. In the end, a few basic functionalities are explained with the help of descriptive figures.

4.1 Chord Ring and Servers

The servers participating in the Chord ring are responsible for creating a ring-like structure based on the hashes of their IP addresses. This section includes the architecture of the Chord and its implementation.

4.1.1 Architecture

A server can join an active ring anytime by contacting an arbitrary node. It will be placed between two existing servers participating in the ring based on its generated hash and store its successor and predecessor pointers. Every node creates a finger table which has 32 entries pointing toward the subsequent nodes as per the start value generated using the following function:

$$(n + 2^k - 1) \mod 2^m$$

$$\text{where } m = 32$$

$$k \text{ is the index with } 1 \leq k \leq m$$

$$n \text{ is the hash value of a node.}$$

This table helps the node find the successor of a particular value without going to each node, increasing the overall performance. The system is also immune to random failure of nodes. All the nodes promptly remove all the entries pointing towards the deleted server and update it with its successor. For this purpose, every node is also equipped with a successor list which is updated regularly. The following diagram depicts an example of a Chord ring containing six servers. It also displays the information stored by a single node. A finger table consisting of entries from 1 to 32 and a successor list help enhance the overall performance.

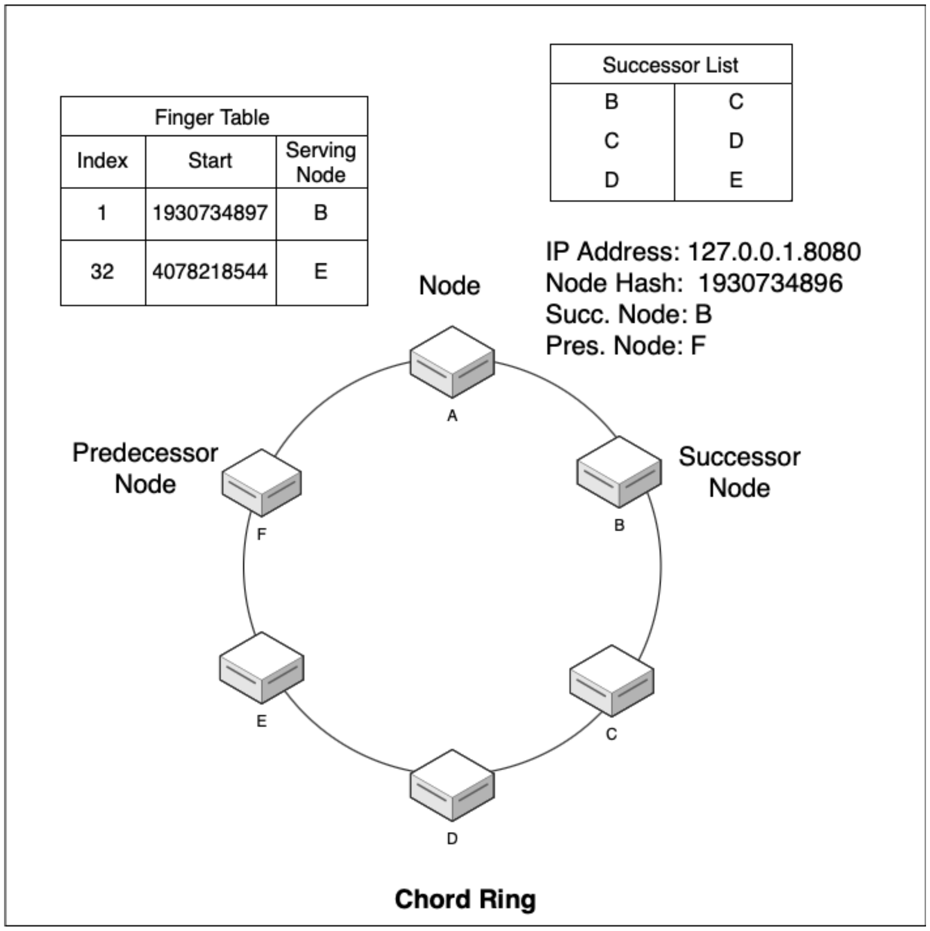


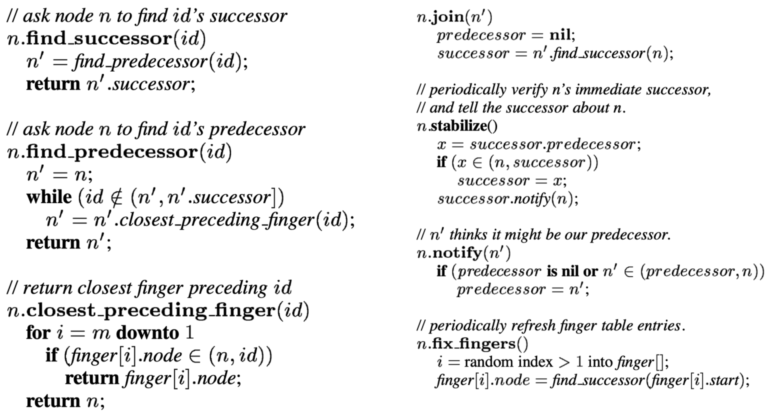
Figure 2: Chord Architecture

Node A's hash value is calculated from its IP address using a slightly modified version of the FNV-1A [12] hashing protocol giving a 32-bit hash value. FNV-1A is an alternate version of the famous Fowler/Noll/Vo (FNV) algorithm, which is designed to be fast with a low collision rate. To receive positive values, the resultant value is passed through a bitwise AND with the largest 32-bit unsigned integer 0xFFFFFFFF.

Initially, the IP address value was hashed using the SHA-256 protocol producing a 256-bit hash. The 256-bit hash resultant required every node to keep a finger table of size 256 per the desired specifications [4]. But as the file store was developed on top of the Chord servers, the performance could have been better. Thus, this version was modified to use the FNV1-A, which provides a faster lookup protocol for hashing and can assign distinct hash values to many servers.

4.1.2 Implementation

This project implements a few fundamental algorithms proposed by Stoica, et al. [4] to implement the Chord lookup protocol. The pseudo codes for the critical functions that form the foundation of the Chord distributed hash table are shown in the following figure.



```
// ask node n to find id's successor
$n$.find_successor($id$)
    $n' =$ find_predecessor($id$);
    return $n'$.successor;

// ask node n to find id's predecessor
$n$.find_predecessor($id$)
    $n' = n$;
    while ($id \notin (n', n'.successor]$)
        $n' = n'$.closest_preceding_finger($id$);
    return $n'$;

// return closest finger preceding id
$n$.closest_preceding_finger($id$)
    for $i = m$ downto $1$
        if ($finger[i].node \in (n, id)$)
            return $finger[i].node$;
    return $n$;

$n$.join($n'$)
    $predecessor =$ nil;
    $successor = n'$.find_successor($n$);

// periodically verify n's immediate successor,
// and tell the successor about n.
$n$.stabilize()
    $x = successor.predecessor$;
    if ($x \in (n, successor)$)
        $successor = x$;
    $successor$.notify($n$);

// $n'$ thinks it might be our predecessor.
$n$.notify($n'$)
    if ($predecessor$ is nil or $n' \in (predecessor, n)$)
        $predecessor = n'$;

// periodically refresh finger table entries.
$n$.fix_fingers()
    $i =$ random index $> 1$ into $finger[]$;
    $finger[i].node =$ find_successor($finger[i].start$);
```

Figure 3: Chord Pseudocode [4]

The stabilize() and fixFingers() functions run periodically on a separate thread to maintain the successor pointer and to rectify older finger table entries. The current implementation uses socket-based data transmission using gRPC (Google Remote Procedure Call) to establish communication between two nodes.

The diagram below illustrates the gRPC services and its attributes used within the fundamental functions listed above.

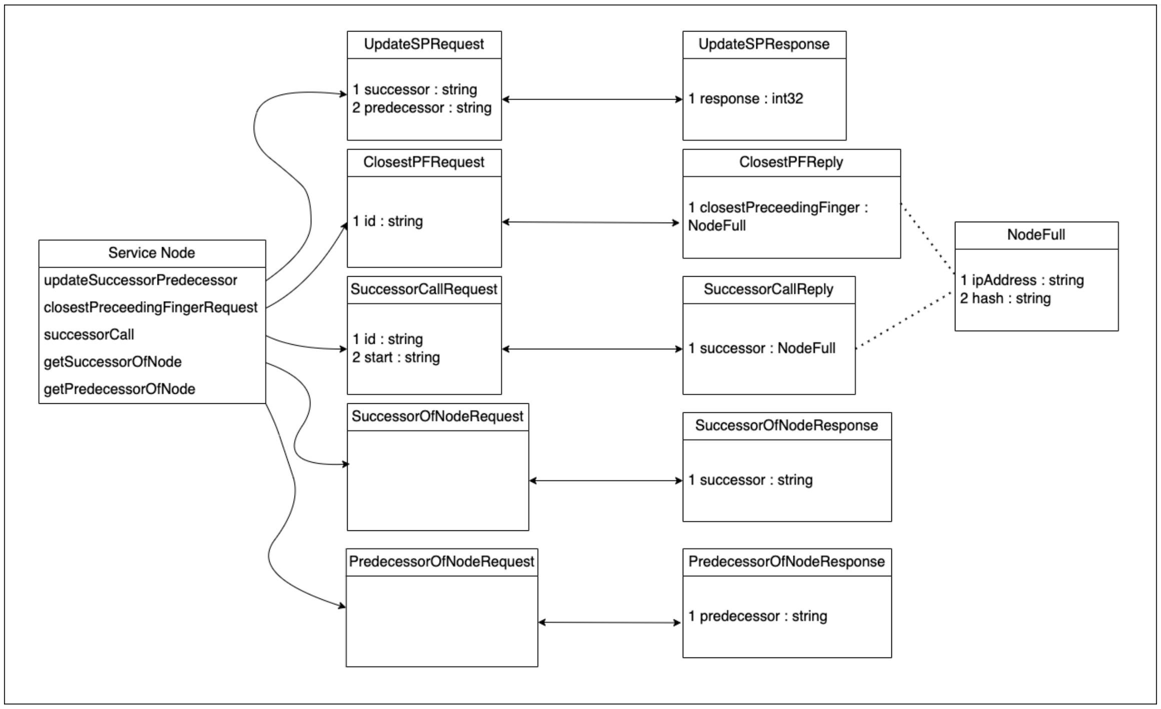


Figure 4: Chord gRPC Services

4.2 File Storage Layer

Every server comprises a file storage layer on top of the Chord architecture, which supports basic functionalities for a file system. Each server listens on its designated port for client or server calls. The architecture and implementation of the file store are explained in the below section

4.2.1 Architecture

When a server receives a file block, it stores it with its necessary metadata. The current file system divides a file into two types of blocks :

1. File Details block

2. Content blocks

The file details block is the primary metadata file that stores a list of all the data blocks that comprise the actual file bytes. This list in the file details block is the most significant part of the file that helps navigate the counterparts. The content block contains the original data bytes of the client's file. Usually, a single file will only have one file detail's block storing information regarding its contents. The maximum size of the File Details block can be 1 MB after which another block is created. Files greater than 256 MB have more than one File Details block.

The file name follows a particular naming convention allowing the Chord ring to locate or redistribute it. The content block files in the system are named after calculating the hashes of the whole data bytes. SHA-256 is the hashing algorithm used for this purpose. This makes file lookup easy and helps determine the integrity of the data bytes in the future. A Merkle tree structure is formed by concatenating hash values of all the blocks of a file; the root hash of this tree is used to name the file details block. As Benet [8] suggests in the IPFS implementation, using a Merkle tree structure has many valuable properties. He expands that it can be used to uniquely identify files for checksum to ensure data integrity and deduplication of data. The SHA-256 hashed file name is rehashed for file lookup using the FNV-1A [12] algorithm as the Chord ring supports such 32-bit hashes.

To safeguard files from any failures, the system stores two replicas, one on the successor node and the other on the predecessor node. If a node gets deleted, its successor caters to the deleted node's file requests, eventually becoming the primary node. Hence, every node stores data blocks for its successor and predecessor. To maintain the default replication factor in order to avoid losing any client data, every node sends its list of block files to its successor and predecessor. In case of missing replicas, the designated replica node requests the primary for it. The current implementation replicates newly stored data blocks and periodically maintains the replica count by interacting with its adjacent nodes.

The following diagram explains the replication strategy used by the project implementation. Node A stores its replicas - A1, A2, and A3 on B and F. Similarly, B and F replicate their primary files on their respective successor and predecessor node

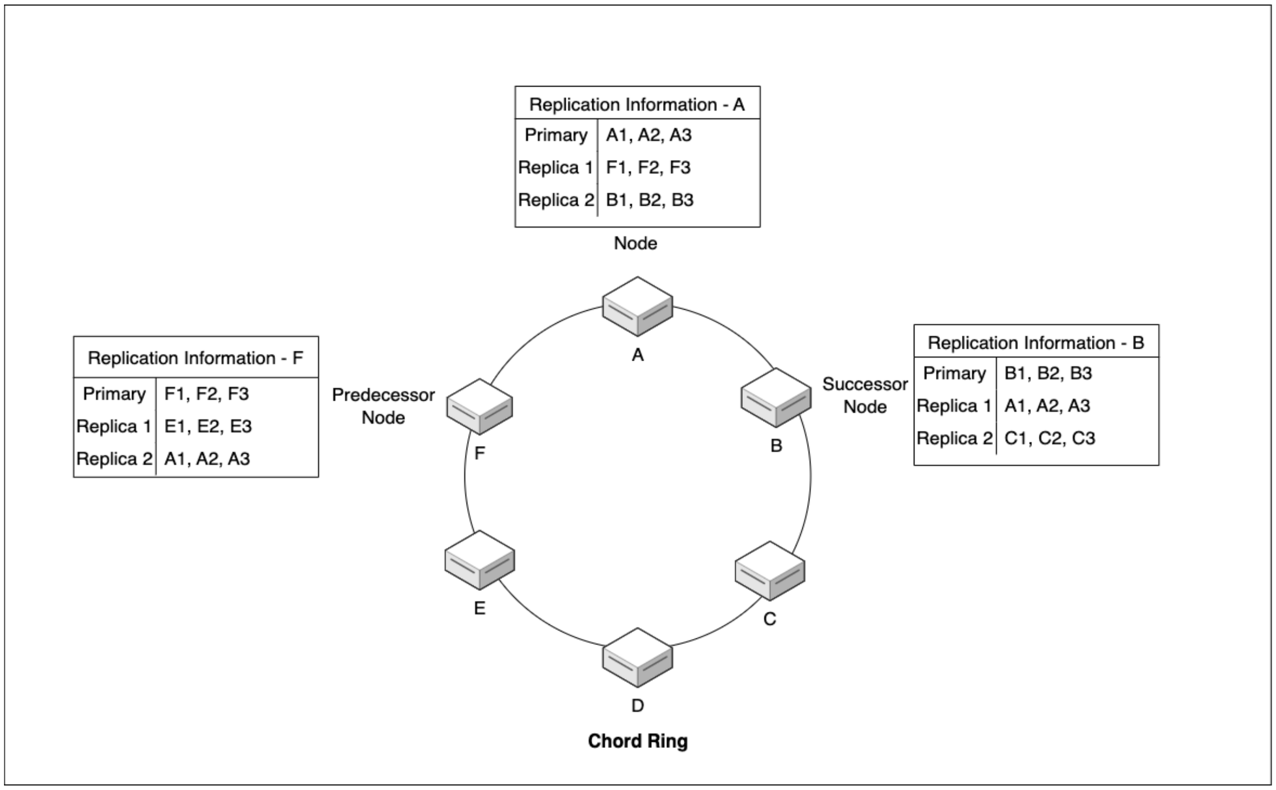


Figure 5: Replication Strategy

4.2.2 Implementation

When a primary node receives a file block, it comes with a Boolean value stating its type, file details, or content. This helps to organize the metadata, which helps classify the file during retrieval or deletion. Every file consists of a few mandatory attributes like the root hash, name of the owner, name of the original file, and its content hash, along with data or names of the content blocks which make the file depending on the file type.

Once a file is received, it is stored by the primary server and replicated asynchronously. The ‘StablizeFileStore’ thread runs separately after every fixed interval, which is responsible for ensuring the server is storing the necessary files and for regular garbage collection. It performs four major tasks: Firstly, it verifies the files persisted on the server by making findSuccessor() call for each file hash. This helps in redistributing misplaced files and ensuring a server's current state. Then it interacts with the two adjacent nodes to ensure all of its primary data has been replicated. After that, it removes unnecessary files tagged for garbage collection by the filesystem. Lastly, it interacts with the smart contract and tracks payments made towards a particular block by its owner.

The respective gRPC services and methods handle the file retrieval requests. The node containing the file details block forwards the request to the respective servers holding the content block. The requested data bytes are read from the content blocks and are transmitted to the requesting client.

Deletion of the file uses the same gRPC request as the File Retrieval. Instead of returning the file bytes, it deletes the blocks and sends an acknowledgment after successful deletion. The StabilizeFileStore thread deletes the replicas after a certain period.

The following diagram gives an idea of the gRPC services the file store uses.

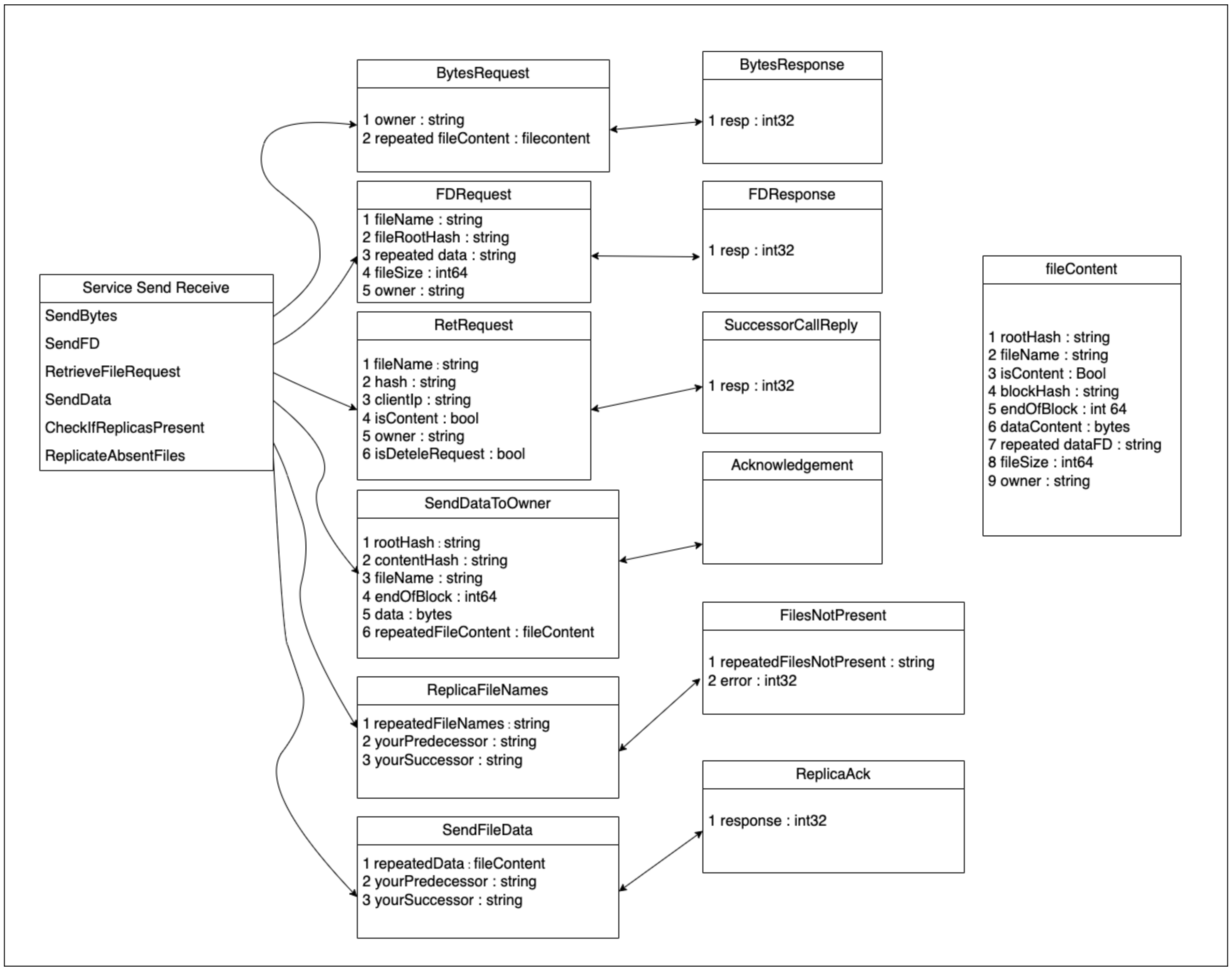


Figure 6: File Store gRPC Services

4.3 Client Layer

The client layer performs important tasks for secured storage of the desired file on the system. It is responsible for dividing the file into adequate blocks, making periodic payments, and checking for any file tampering after retrieval. The architecture and implementation of the client layer has been described below.

4.3.1 Architecture

The client can store files on the file system by paying some ether on the smart contract. For every block, the client sends a random number, a verification hash, and some amount on the contract to initiate file storage. Then it receives an IP address of an arbitrary node from the contract, which is used for further storage. For file retrieval, the client maps a root hash to the file name desired. An arbitrary node is then contacted for file retrieval to make a findSuccessor() call for the root hash value. Then the root hash is sent to the resultant server received by completing the findSuccessor() call. The root hash points to the file detail's block that stores navigating information regarding the content blocks. These content file names are then searched on the Chord and sent to the requesting client. File deletion works similarly. In file retrieval, all the blocks received from the servers check the resulting root hash to ensure their integrity. The client also takes care of hourly pay for every file block to maintain its files stored.

4.3.2 Implementation

While storing a file client calls a helper function which converts the file into blocks. Each block can be 64 KB if the file is large enough. A file is transformed into a FileDetails object which contains a file name, size, root hash, and a list of Content objects. Content stores the encrypted data bytes, the hash of the block, and the last index of the byte, which helps reconstruct the file in the future.

After this, every content object is passed through a function to store the necessary information on the contract. The byte[] array shown in Figure 7 is encrypted with the key provided by the user using the AES algorithm.

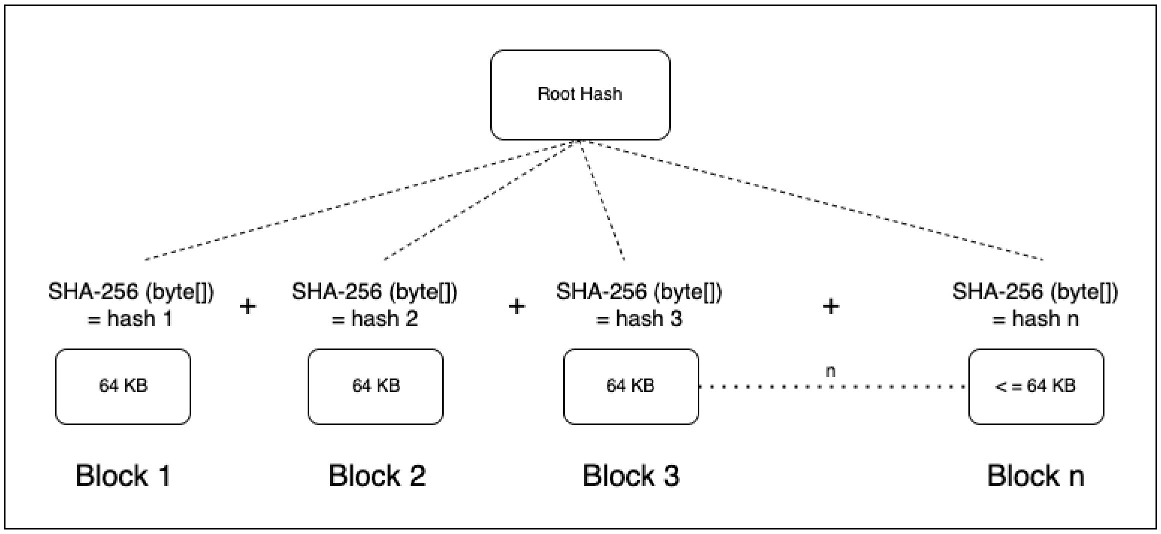

Root Hash
SHA-256 (byte[]) = hash 1
+
SHA-256 (byte[]) = hash 2
+
SHA-256 (byte[]) = hash 3
+
SHA-256 (byte[]) = hash n
64 KB
64 KB
64 KB
n
< = 64 KB
Block 1
Block 2
Block 3
Block n

Figure 7: File Distribution into Blocks

The below algorithm describes the method used to calculate a verification hash for a block and store that data on the smart contract.

Algorithm 1 $storeBlockInformationOnSolidityContract(content, ownerName, contract)$

```
*bytes* ← data bytes from *content*
*randomNumber* ← A random integer between *0 - bytes.length*
*randomNumberByte* ← byte representation of *randomNumberByte*
*modifiedBytes* ← Put *randomNumberByte at index randomNumber*
*hash1* ← *keccakHash(modifiedBytes)*
*finalVerificationHash* ← *keccakHash(hash1 + randomNumber)*
*store(content.hashId, randomNumber, finalVerificationHash, ownerName)*
```

The Keccak [13] hash generates a 256-bit hash value from the given input byte array. It returns a hexadecimal string representation of the hash value. The final verification hash ensures that a server stores the whole block without tampering.

Every client stores the root hash for every file mapped to the file name. When a client puts in a retrieval request for a file name, it gets the respective root hash and forwards it to the respective node to retrieve all the content blocks mapped to the root hash. On receiving the blocks, the client continuously calculates the root hash to check if the whole file has been retrieved. Along with the data blocks, the servers also send the last indexes of the byte arrays, which help reconstruct the file.

4.4 Smart Contract Layer

The smart contract layer consists of a fully deployed solidity smart contract. This contract helps monetize and secure the storage system by recording necessary transactions. The architecture and implementation of the contract are given below.

4.4.1 Architecture

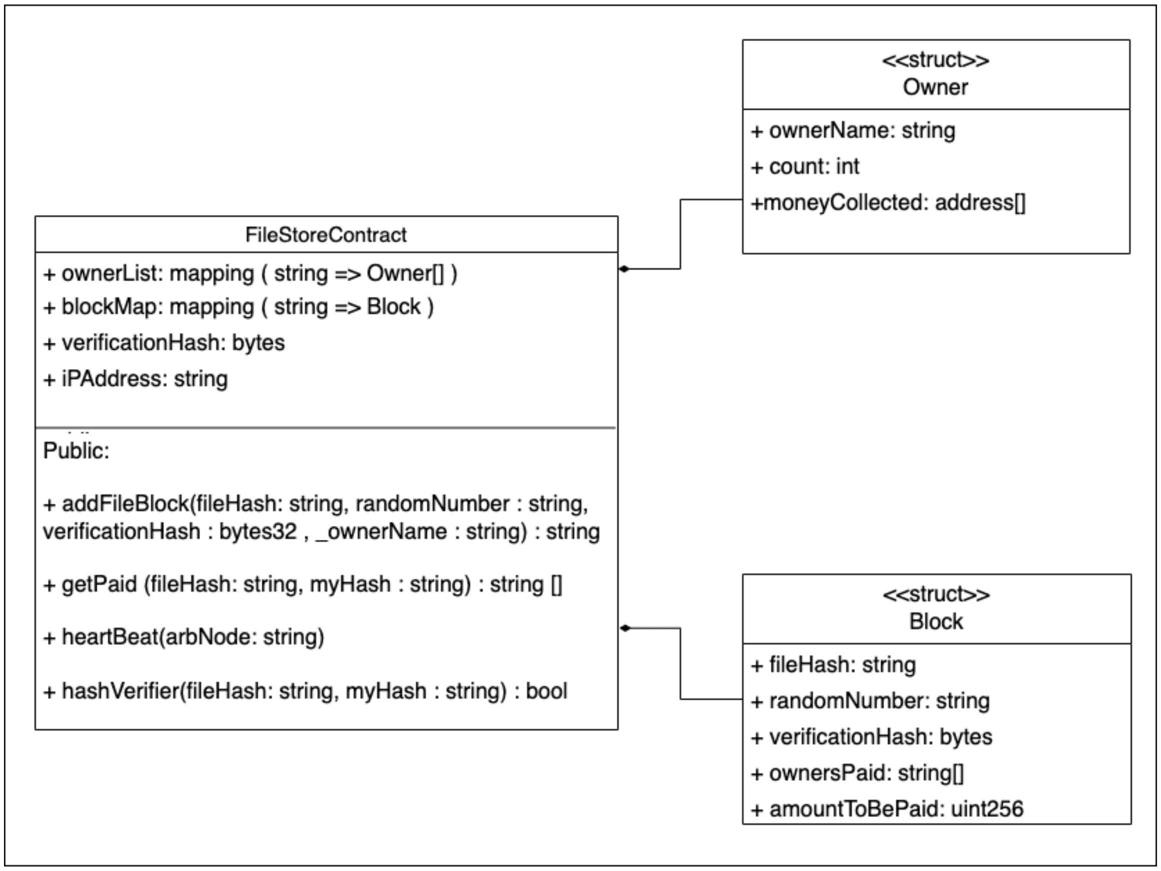
The FileStoreContract acts as a middleman for the server nodes participating in the Chord ring and the clients wanting to store data. A highly trustworthy digital ledger has been added to avoid any monetary disputes. It plays a vital role in making sure the system holds the client's data without fail.

4.4.2 Implementation

The smart contract has been integrated into the filesystem for two significant reasons – to introduce an incentive layer for users participating in the Chord ring as storage servers and to persist and verify proof of storage on servers so that all the transactions are recorded and verified, which improves the integrity of the whole system.

The contract has a few client-facing functions and a couple of server-facing parts. The client calls the addBlock() function to initiate storage of a file block by sending its verification hash as derived in Algorithm 1. This, in return, calls the getIpAddress() function to get an arbitrary node in the system. This arbitrary node makes a findSuccessor() call on the Chord for lookup and then sends the data block. The most significant function for a server is getPaid(). Every server periodically calls this function to get paid for storing a block of data for an hour. On every getPaid() call by a server, the contract sets that particular server as the arbitrary IP address. According to the contract, the current actively running node can cater to client requests. A new object is created every time a client pays for a block. It also provides security such that no single server can make more than one getPaid() call for one block in an hour. The system aims to evenly pay the primary and the replicas the desired amount without malpractices from faulty servers.

The diagram below shows the user-defined data types used in the Solidity smart contract to store information regarding blocks and verification information.



<<struct>>
Owner
+ ownerName: string
+ count: int
+moneyCollected: address[]
FileStoreContract
+ ownerList: mapping (string => Owner[])
+ blockMap: mapping (string => Block)
+ verificationHash: bytes
+ iPAddress: string
Public:
+ addFileBlock(fileHash: string, randomNumber : string, verificationHash : bytes32 , _ownerName : string) : string
+ getPaid (fileHash: string, myHash : string) : string []
+ heartBeat(arbNode: string)
+ hashVerifier(fileHash: string, myHash : string) : bool
<<struct>>
Block
+ fileHash: string
+ randomNumber: string
+ verificationHash: bytes
+ ownersPaid: string[]
+ amountToBePaid: uint256

Figure 8: FileStoreContract.sol : Class Diagram

4.5 Filesystem Architecture Flow

Here, we look at three scenarios experienced while operating the file system. This section includes descriptive figures for storing a file, deleting a file and retrieving a file.

4.5.1 Store File

A basic flow of the file system architecture for storing a file has been demonstrated below.:

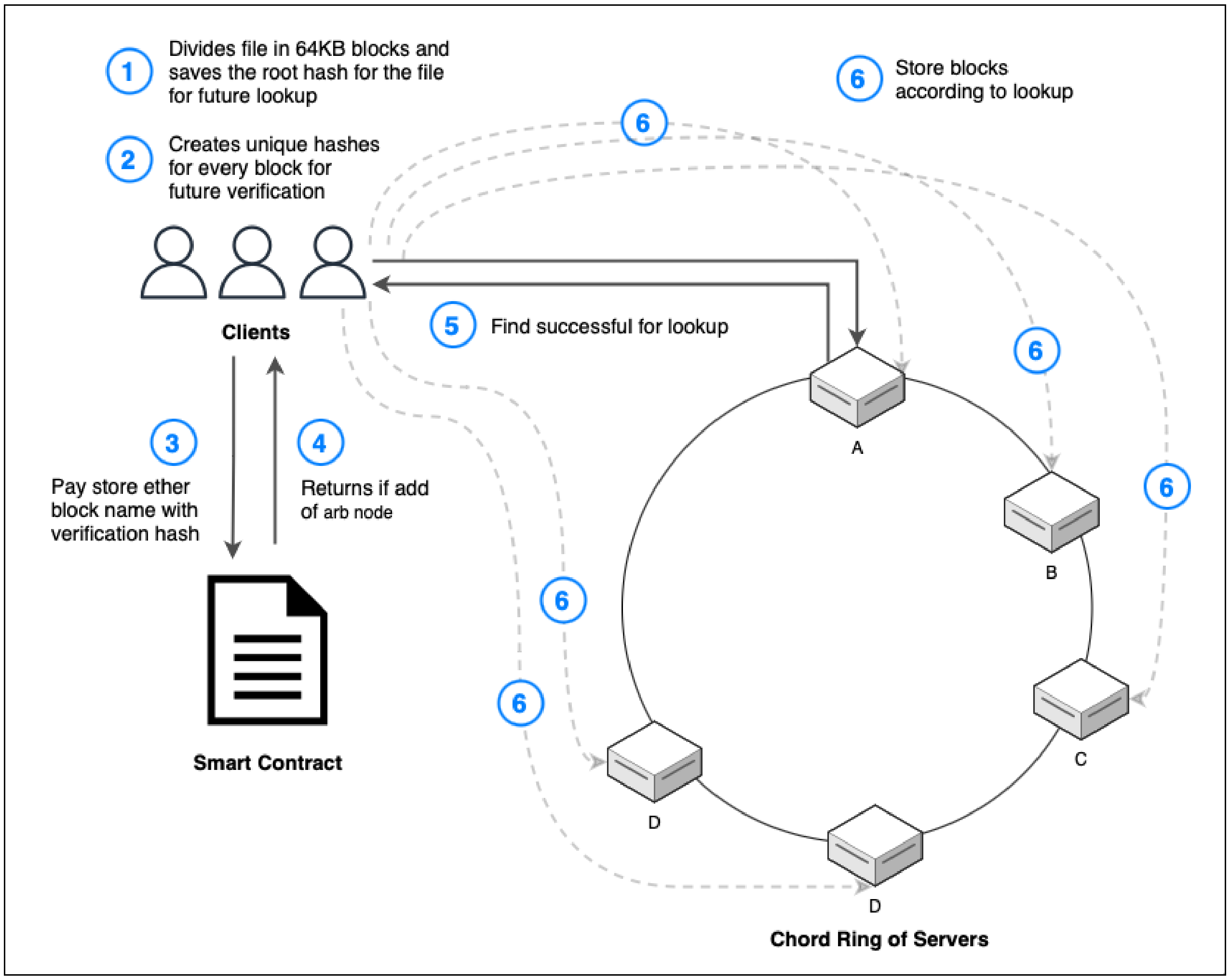

1
Divides file in 64KB blocks and saves the root hash for the file for future lookup
2
Creates unique hashes for every block for future verification
6
Store blocks according to lookup
Clients
5
Find successful for lookup
3
Pay store ether block name with verification hash
4
Returns if add of arb node
Smart Contract
6
A
B
C
D
D
Chord Ring of Servers

Figure 9: Storing a File on the Filesystem

The file storage process starts at the client layer. Every file is divided into small encrypted blocks, where 64KB is the maximum size for one block. For every block, the client calculates 256-bit hashes from the data. These content block hashes are used to calculate the root hash of the file and are important to locate the respective contents of a file going ahead. Then the client creates a File Details block which stores the list of all the content blocks for future lookup. The File details block is mapped to the root hash. After this, a random number is generated, which is used to calculate a verification hash. While the client pays for a block, it stores the hash value of a block, a random number, and a verification hash on the smart contract every time. After paying for all the blocks, the smart contract returns the IP address of a random node from the filesystem. Using this IP address, a lookup call is made for all the blocks, and they are sent for storage on the respective servers.

4.5.2 Retrieve File

Here we see how a file is retrieved by combining blocks from multiple servers:

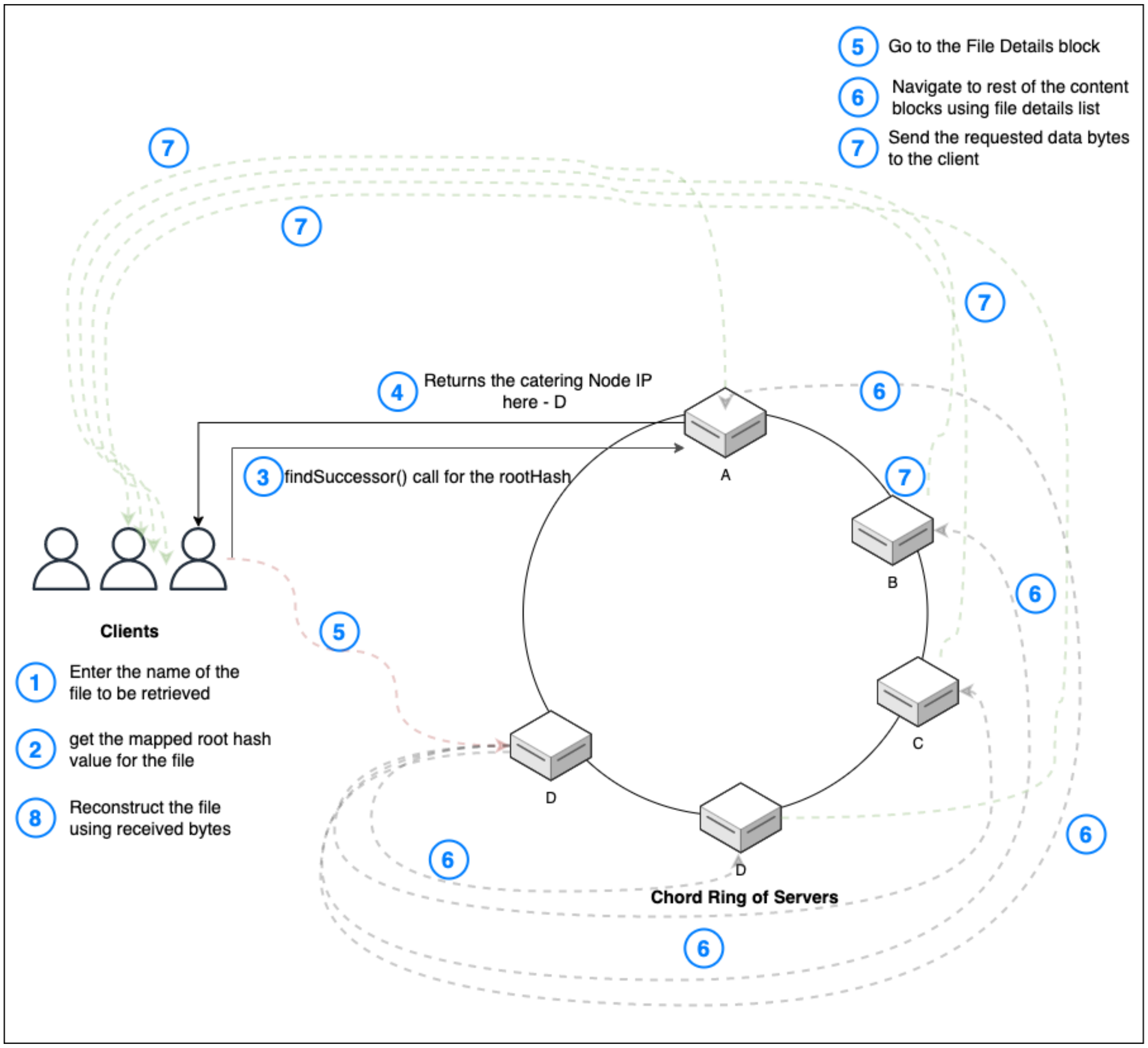

5 Go to the File Details block
6 Navigate to rest of the content blocks using file details list
7 Send the requested data bytes to the client
7
7
7
4 Returns the catering Node IP here - D
6
3 findSuccessor() call for the rootHash
A
7
B
6
Clients
5
1 Enter the name of the file to be retrieved
2 get the mapped root hash value for the file
8 Reconstruct the file using received bytes
D
C
D
6
6
Chord Ring of Servers
6

Figure 10: Retrieving a File from the Filesystem

To start with the file retrieval, the client makes a Chord lookup call using an arbitrary node for the desired file's root hash value which it maintains. On receiving the respective IP address, a retrieve request is sent to that server with the root hash. The server then reads the metadata mapped to this hash, which lists the hashes of all the file's content blocks. This server now makes a lookup for all the content blocks using the hashes and receives IP addresses for the catering nodes for each block. Then the retrieve request is forwarded to those servers. After receiving the request for the content blocks along with the IP of the client, the servers send the desired blocks to the client. The client then reconstructs the file, decrypts it using the same key, checks for integrity by recalculating the hashes, and stores it locally.

4.5.3 Delete File

The deletion process of a file has been illustrated in the figure below :

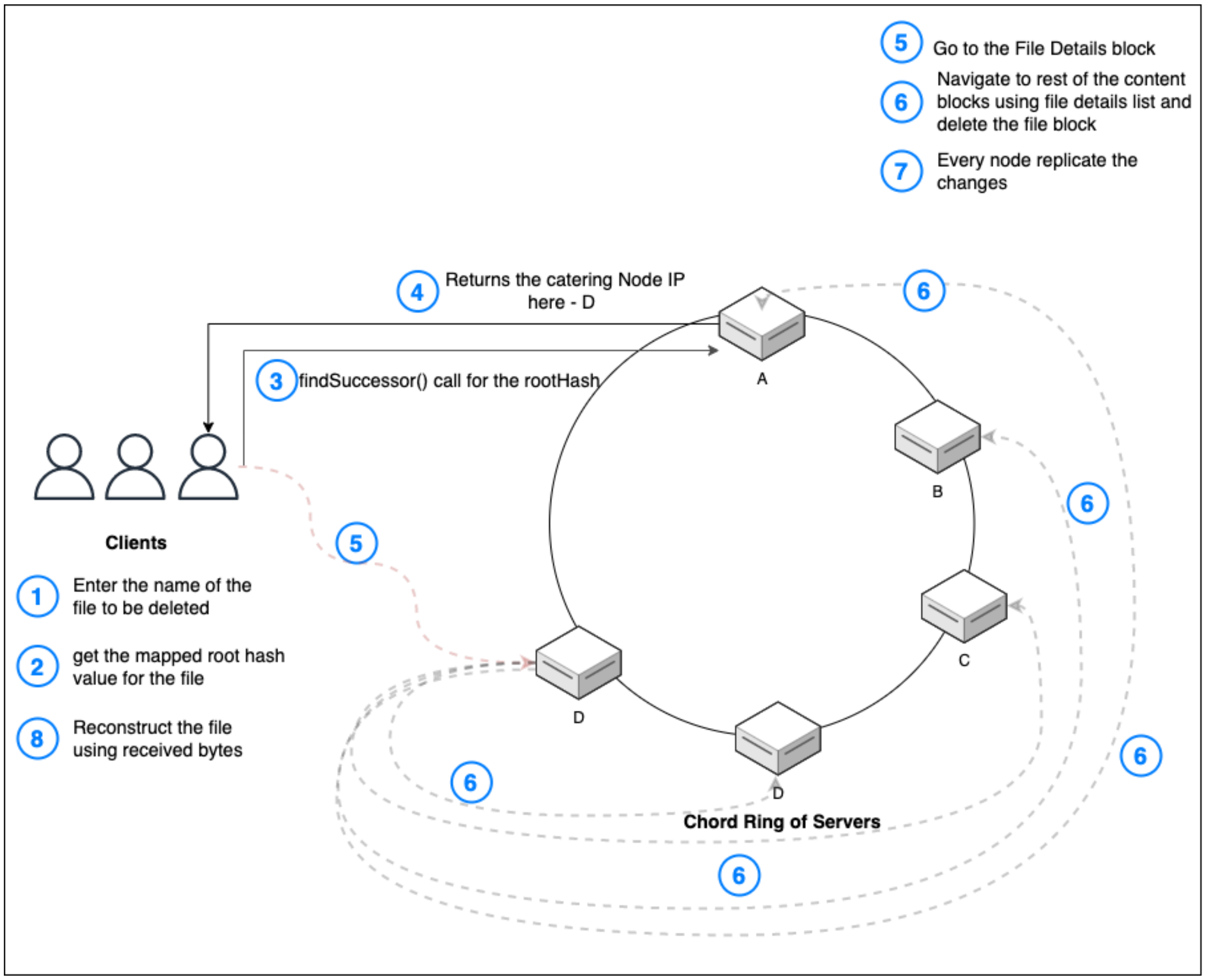

5 Go to the File Details block
6 Navigate to rest of the content blocks using file details list and delete the file block
7 Every node replicate the changes
4 Returns the catering Node IP here - D
3 findSuccessor() call for the rootHash
Clients
1 Enter the name of the file to be deleted
2 get the mapped root hash value for the file
8 Reconstruct the file using received bytes
5
6
6
6
6
6
A
B
C
D
D
Chord Ring of Servers

Figure 11: Delete a File from the Filesystem

Deleting a file follows a similar procedure to that of the retrieve request. In the end when all the content blocks for the file are found, instead of sending the blocks back to the client they are permanently deleted from the server. The replicas get tagged for timely garbage collection.

V. Experiments and results

The clients can use the implemented file storage to store as small as a few kilobytes and as large as 100 megabytes. As a distributed file storage, the performance can depend on the number of nodes currently serving requests and the number of lookups required in our distributed hash table. Every file first gets divided into blocks on the client side, and then the block names get stored on the smart contract along with the verification hash. After this, the blocks are stored on the Chord ring.

The client and the Chord servers have been implemented in Java and make use of gRPC to establish communication amongst the instances. Web3J library has been used to deploy the smart contract and to interact with it as needed.

Any user can run a client or earn ether by acting as a server on its machine if the below requirements are met:

1. Java 11

2. An empty port for gRPC services to run

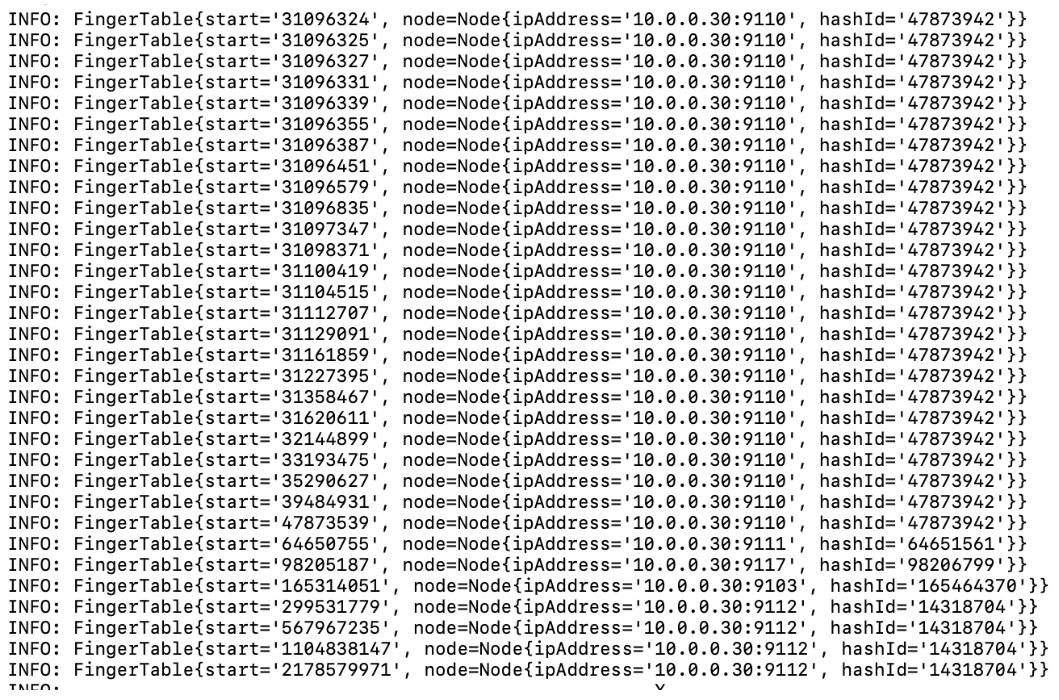
3. IP address of the blockchain network and the deployed contract's address.

4. Crypto Wallet (loaded with Ether)

The experiments have been performed on a test chain provided by the Ganache tool.

The Chord servers run two parallel processes to maintain the correctness of the Chord and to keep the finger table updated as nodes get added and deleted constantly. To maintain the files stored on the servers, a separate process keeps track of all the blocks. It re-distributes the blocks if required, does garbage collection, replicates data, and takes care of the payment.

Every node contains a finger table for easy lookup. Below is an example of a finger table created on a node running on port 9012 with its successor and the following nodes shown in the example below.



```
INFO: FingerTable{start='31096324', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31096325', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31096327', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31096331', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31096339', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31096355', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31096387', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31096451', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31096579', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31096835', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31097347', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31098371', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31100419', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31104515', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31112707', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31129091', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31161859', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31227395', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31358467', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='31620611', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='32144899', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='33193475', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='35290627', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='39484931', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='47873539', node=Node{ipAddress='10.0.0.30:9110', hashId='47873942'}}
INFO: FingerTable{start='64650755', node=Node{ipAddress='10.0.0.30:9111', hashId='64651561'}}
INFO: FingerTable{start='98205187', node=Node{ipAddress='10.0.0.30:9117', hashId='98206799'}}
INFO: FingerTable{start='165314051', node=Node{ipAddress='10.0.0.30:9103', hashId='165464370'}}
INFO: FingerTable{start='299531779', node=Node{ipAddress='10.0.0.30:9112', hashId='14318704'}}
INFO: FingerTable{start='567967235', node=Node{ipAddress='10.0.0.30:9112', hashId='14318704'}}
INFO: FingerTable{start='1104838147', node=Node{ipAddress='10.0.0.30:9112', hashId='14318704'}}
INFO: FingerTable{start='2178579971', node=Node{ipAddress='10.0.0.30:9112', hashId='14318704'}}
INFO:
```

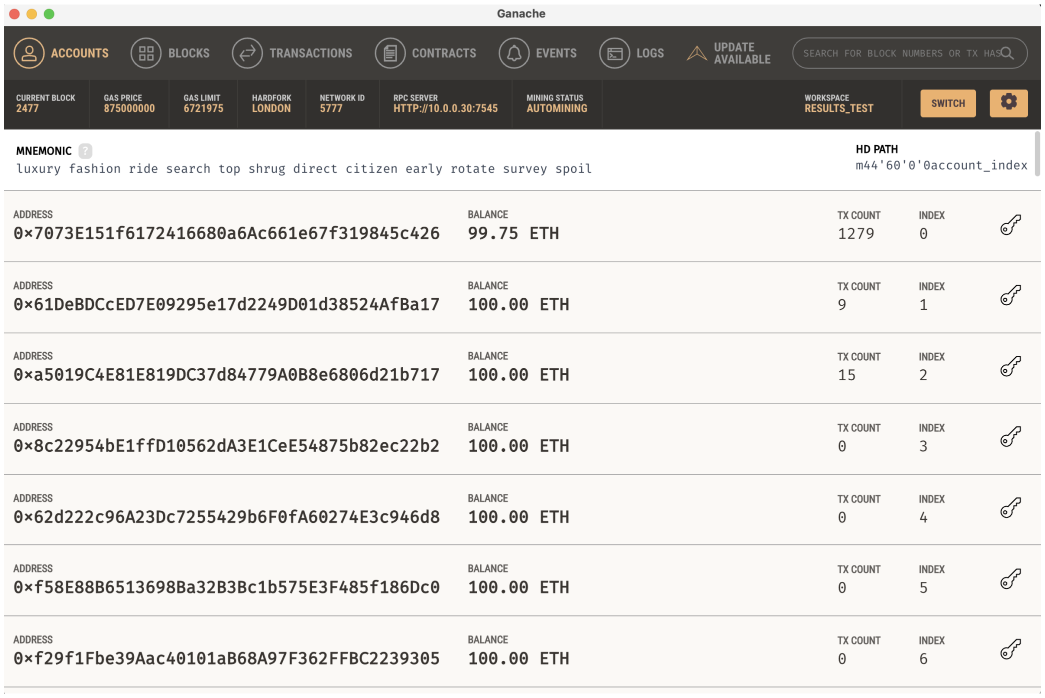
Figure 12: Example of a Finger Table

From the above figure, we can confer that 9110 is the successor node of the server running on port 9012. 9110 is followed by another node which servers entries having hash values between 47873942 and 64651561, which is 9111. Similarly, 9117, 9103, and 9112 serve keys in the succeeding Chord ring. This pre-calculated table will make any lookups made on 9012 quick and improve the overall performance. Every node has 32 entries calculated using a similar function as the one shown above.

The Ganache tool allows developers to create a blockchain workspace with a testing network for Ethereum-based solidity smart contracts. The configuration used for testing includes randomly generated 100 addresses with a default of 100 ether each in every account's wallet. The server is running on port 7545 of the local machine.

For testing purposes, Ganache creates a test network Hardfork - London. The gas price is set to the minimum possible value of 875000000 Wei. The gas limit is set to 6721975 Wei. It is crucial to find a balance between these values to make sure limited ether is spent for every transaction.

The figure below shows the created workspace to test transactions by deploying the smart contract.



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
2477	875000000	6721975	LONDON	5777	HTTP://10.0.0.30:7545	AUTOMINING	RESULTS_TEST

MNEMONIC
luxury fashion ride search top shrug direct citizen early rotate survey spoil

HD PATH
m44'60'0'0account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0×7073E151f6172416680a6Ac661e67f319845c426	99.75 ETH	1279	0
0×61DeBDCcED7E09295e17d2249D01d38524AfBa17	100.00 ETH	9	1
0×a5019C4E81E819DC37d84779A0B8e6806d21b717	100.00 ETH	15	2
0×8c22954bE1ffD10562dA3E1CeE54875b82ec22b2	100.00 ETH	0	3
0×62d222c96A23Dc7255429b6F0fA60274E3c946d8	100.00 ETH	0	4
0×f58E88B6513698Ba32B3Bc1b575E3F485f186Dc0	100.00 ETH	0	5
0×f29f1Fbe39Aac40101aB68A97F362FFBC2239305	100.00 ETH	0	6

Figure 13: Ganache : Blockchain Testing Tool

The first call made by the client stores the following information for every block - verification Hash, Owner name, Some unit of ether value sent by the client, and a random number. To store this information, the client requires an average of 200 milliseconds for every block of 64 KB. The storage of files begins after these blocks are stored on the contract. The average gas required for the above transaction is 178500 units. Every transaction requires a fee so that a miner can compute the operation on the blockchain. This fee is called gas. The amount of gas required for different computations performed is fixed for a solidity contract. But, the cost of gas keeps fluctuating, similar to our real-world gasoline prices.

The getPaid() function requires an average amount of 360000 gas for a single server. This is the transaction cost paid to compute the required functions to receive some unit of ether. As the gas limit has been set to 6721975 Wei, The total cost of gas needed to perform the transaction can be calculated using the following formulae -

$$\text{Total Fee} = \text{Gas Units(limit)} \cdot \text{Gas Price Per Unit}$$

Here, the gas unit is 6721975 even though the average gas required is only 360000. After the transaction has been completed, the user receives the difference amount that was not required to compute operations.

$$6721975 - 360000 = 6361975$$

If the gas limit is set to 200000, less than the gas required, it won't perform any transactions and will throw an error.

According to the current test scenario, the gas price is set to 875000000 Wei, which is equal to 0.000000000875 ether.

For the server to make some profit, the default value received per block should at least break even the transaction amount, which comes to:

$$0.875 \text{ Gwei} \cdot 360000$$

The below table shows the cumulative gas required to store file details on a smart contract:

Cumulative Gas Consumption	
File Size	Cumulative Units of Gas Consumed
35 KB	133544
35 MB	70509504
120 MB	426215339

Table 1: Cumulative Gas Consumption

The File system was tested using a client to store files of multiple sizes. The performance test results presented are an average of three trial rounds for every case. The first performance test was done with five nodes running in parallel as part of the Chord ring. The following table describes the average results for five nodes.

Results on 5 servers		
File Size	Store File	Retrieve File
35 KB	194 ms	57 ms
35 MB	2716 ms	5190 ms
120 MB	7000 ms	17647 ms
300 MB	18592 ms	37852 ms

Table 2: Performance Testing on 5 Nodes

After testing with five nodes, another set of tests was carried out with 15 nodes with multiple operations on varying data sizes. Following is the average of the observed result for the same:

Results on 15 servers		
File Size	Store File	Retrieve File
35 KB	312 ms	63 ms
35 MB	4363 ms	8412 ms
120 MB	12251 ms	21321 ms
300 MB	20150 ms	48279 ms

Table 3: Performance Testing on 15 Nodes

In addition, similar operations were performed on a larger number of servers participating in the distributed filesystem. The table below shows the performance testing of a filesystem with 25 nodes running at once.

Results on 25 servers		
File Size	Store File	Retrieve File
35 KB	317 ms	75 ms
35 MB	4334 ms	7842 ms
120 MB	9587 ms	20426 ms
300 MB	22770 ms	50215 ms

Table 4: Performance Testing on 25 Nodes

As observed from the above results, we can see that the time to store and retrieve a file increases by increasing the number of servers. But, the time does not increase linearly as we increase the servers. The time is increased only by a few seconds, even with a difference of 20 servers serving in the filesystem. But, as the file size increases, the file storage time increases linearly.

From the above table, it takes approximately 10 seconds to store all the blocks of a 100 MB file in the system. The time of retrieval is nearly 20 seconds for the same.

A file smaller than 64 KB, consisting of only one block, will require more time to store than retrieve. It requires less than 50 milliseconds on average to retrieve files smaller than 64 KB. Thus, the implemented system is very efficient against relatively smaller files.

All the file blocks are replicated on two other servers within 20 seconds of receiving an acknowledgment for a successfully stored file.

VI. Conclusion

The developed file system boasts impressive benefits, including great performance, lucrative rewards, robust storage verification, effortless replication, and a remarkable fault-tolerant design. It presents an enticing option for users seeking to maximize their storage capabilities and make the most of their available space.

It provides a high-performing, flexible, and transparent storage facility for people in need of extra storage in the form of a client. Users with empty storage spaces can rent them out to earn some profit by participating as Chord servers. Every user requires a blockchain wallet to make use of the incentive layer.

The file store can work with multiple nodes and tolerate faults because of constant data replication. It currently implements a pay-on-the-go approach with an hourly payment strategy adopted by the system. It can support multiple nodes simultaneously for better distribution of data blocks.

This filesystem can be utilized on an organizational level as well. Organizations that provide work machines to their employees or stakeholders can embed the current file system. A running instance of the current file system with an internal incentive layer can help organizations reduce cloud storage costs. All the resources can be used at their maximum potential level.

VII. Future work

The current system can be improved, more robust, and user-friendly by applying a few modifications and enhancements. The enhancements required on different layers of architecture are given below:

7.1 Client layer

The current client allows users to store and retrieve a file using a root hash mapped to the file name. But, a folder structure still needs to be integrated into the client for better user interaction with the filesystem. Having folders will make the system more user-friendly and allow users to work with a standard setup. Storj [3] enables users to store files in a single-layer directory structure, bucket. Adding a folder structure will also allow users to classify their files better.

7.2 Smart Contract Implementation

The current computations performed while calling the getPaid() function are relatively costly. This is to have a secure payment gateway for every server and client by ensuring no server tries to receive compensation more than once an hour for a block. Also, to ensure the server stores the data block by computing verification hash. A lesser complex computation requiring lower gas could be implemented to maximize the amount received for a block and reduce the gas cost incurred by a client. Additionally, every user must contact the server hourly to retrieve the file, store the random number and a random hash, and pay some amount for the block. This has a few advantages: a user can be assured of the block being stored on the system, but a lot of computation is required at the client's end. This strategy could be improved going further.

LIST OF REFERENCES

[1] S. GHEMAWAT, H. GOBIOFF, and S.-T. LEUNG, "The google file system," *vol. 37, no. 5, pp. 29–43, doi: 10.1145/1165389.945450*, 2003.

[2] K.Shvachko, H.Kuang, S.Radia, and R. Chansler, "The hadoop distributed file system," *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010, pp. 1–10, doi: 10.1109/MSST.2010.5496972*, 2010.

[3] I. Storj Labs, "Storj: A decentralized cloud storage network framework," https://www.storj.io/storjv3.pdf, Oct 2018.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *vol. 11, no. 1. pp. 17–32*, 2003.

[5] I. Dimitrov, "Stacks of storage: How much space does your data take up?" https://blog.pcloud.com/stacks-of-storage/, Dec 2020, (Accessed on 05/02/2023).

[6] H. Huang, J. Lin, B. Zheng, Z. Zheng, and J. Bian, "When blockchain meets distributed file systems: An overview, challenges, and open issues," *IEEE Access, vol. 8, pp. 50574-50586, 2020, doi: 10.1109/ACCESS.2020.2979881*, 2010.

[7] S. team, "Swarm - storage and communication infrastructure for a self-sovereign digital society," https://www.ethswarm.org/swarm-whitepaper.pdf, Jun 2021.

[8] J. Benet, "Ipfs - content addressed, versioned, p2p file system," https://docs.ipfs.tech/concepts/further-reading/academic-papers/#democratizing-content-publication-with-coral.

[9] J. A. Johnsen, L. E. Karlsen, and S. S. Birkeland, "Peer-to-peer networking with bittorrent," https://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf, Dec 2005.

[10] B. Cohen, "Incentives build robustness in bittorrent," https://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf, May 2003.

[11] D. Vorick and L. Champine, "Sia: Simple decentralized storage," https://sia.tech/sia.pdf, Nov 2014.

[12] L. Noll, "Fnv hash," http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-1.

[13] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "The keccak reference," https://keccak.team/files/Keccak-reference-3.0.pdf, Jan 2011.