High Performance Distributed Filesystem Based on Blockchain

A Project Report

Presented to Dr. Chris Pollett

Department of Computer Science

San Jose State University

In Partial Fulfilment

of the Requirements for the Class

Fall 2022: CS 297

By

Ajinkya Rajguru

December 2022

ABSTRACT

Distributed Filesystem architectures have enabled usage of commodity hardware to store data on a large scale with maximum consistency and availability. Blockchain makes it possible to store information which can never be tampered with and allows for incentivization of a traditional decentralized storage system. The goal of this project is to implement a decentralized filesystem which leverages blockchain to keep a record of all the transactions. So, it was important to get myself acquainted with the existing decentralized filesystem architectures and the file system structures. In order to do that we decided upon four different deliverables to be completed over the semester. The first deliverable is to implement Chord which is a protocol for a peer-to-peer distributed hash table. The second deliverable is to deploy and carry out performance testing of a few existing blockchain based filesystems. Third deliverable includes basic designing and development of a decentralized file structure for the system. Finally, the fourth deliverable was to develop a gambling game and deploy the smart contract on Ethereum block chain using solidity.

Keywords - Distributed Filesystem, Blockchain, Chord, hashtable, Smart Contracts, Ethereum, Solidity,

TABLE OF CONTENTS

I. INTRODUCTION

In today's data driven world built on ease of accessibility and constant availability of data, distributed storage systems play a significant role. A distributed filesystem allows access to files and data at any time through a computer network which can be stored on multiple file servers or locations. It unleashes the real potential of file sharing, controlled access of data and data reliability. The conventional filesystems like the Google File System [1] and the Hadoop File System [2] have many advantages and vast array of use cases. As suggested by Huang, et. al [3] they still have a few inevitable drawbacks like instability, little to no auditing and incentive mechanisms and a requirement of a high-performance master hardware system because of a single point of failure.

Due to the drawbacks mentioned above and the ease of maintaining monetary transactions because of cryptocurrency, blockchain based decentralized filesystems are becoming a new generation of distributed peer to peer storage systems. Blockchain is a distributed ledger which stores records that are immutable and sharable. Introducing blockchain while developing a distributed filesystem provides a much-needed boost of increased stability, robust auditing, higher data integrity, etc.

The main objective of this project is to develop a distributed storage system which makes use of blockchain as an incentive layer and for maintaining integrity of transactions. To accomplish this, the main goal of this semester is to carry out a literature survey of the existing systems along with performance testing of in-use filesystems and start with the development of multiple components required for a fully functional decentralized file system based on

blockchain. The following report is organized as follows: The Deliverable 1 consists of a study on the famous distributed hash table Chord and its advantages along with a working example of the same implemented in Java. Then the Deliverable 2 gives an overview of the existing blockchain based filesystems followed by the 3rd Deliverable which consists of the file system structure that can be used in a decentralized file storage for efficient allocation of nodes and metadata. Deliverable 4 briefly discusses about smart contracts and using solidity to develop a gambling game consisting of 5 players. Lastly, the conclusion section summarizes the significance of all the deliverables and the future work to be done during the upcoming semester.

II. DELIVERABLE 1 – CHORD: A DISTRIBUTED HASH TABLE

A distributed hash table works similar to a normal hash table which allows for value mapped to a distinct key. The basic idea behind Distributed hash tables is to provide a strong abstraction for a lookup service in a vast network of devices. Keys allocated to a particular node can be easily redistributed with minimum effort in case of insertion, deletion, or failure of a node. Its robustness can also reduce the necessity of having a Master or a governing service which can prove to be a single point of failure. Stoica, et al. [4] proposed a highly scalable lookup protocol for peer-to-peer systems known as Chord. Its main aim is to efficiently map a key to a particular node such that it can be found with minimum effort in a network of multiple nodes. Chord has the capability to adapt to a changing network where nodes are continuously getting removed and added.

My first deliverable revolved around implementing a simple example of the Chord protocol which could work with multiple nodes and allow for efficient storing and lookup of keys as inserted by the clients. My Java implementation of chord utilizes the gRPC library for interaction between multiple nodes along with Maven repository for required dependencies. The nodes could range between the port numbers 9000 – 9127 which are logically placed in a circular ring one after the other as shown in the figure 1. It also includes a client side which provides a simple switch case menu allowing a client to enter a new key, view the finger table of an existing node, check keys stored on a node and query a node to show all the keys present.

Each node creates its own finger table with n entries (here 7) as theoretically there could be 2^7 i.e., 128 nodes in the ring. Each entry includes an interval and current node which is serving the floor value of the interval. Along with the finger table every node also stores all the keys that are assigned to

it.

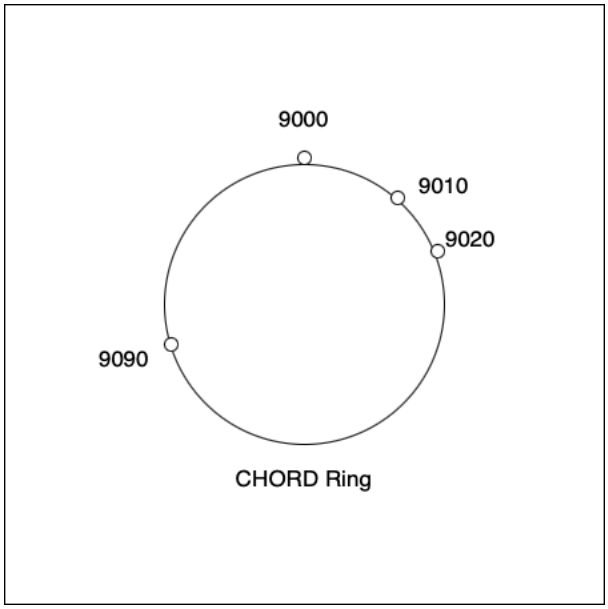

Figure 1: Chord Ring

Interval	Serving Node
91.0 – 92.0	9000
92.0 – 94.0	9000
94.0 – 98.0	9000
98.0 – 106.0	9000
106.0 – 122.0	9000
122.0 – 26.0	9000
26.0 – 90.0	9090

Table 1: Finger Table Example for 9090

As seen in the above example and the implementation results, 9000 is 9090's successor and will serve majority of the keys as there are no other nodes in between them. Similar tables are created automatically for all the nodes which allows for a faster lookup and doesn't require 'n' number of communications amongst the nodes. Once a node shuts down or is removed, all its keys get assigned to the successor of the node and the pointers of its successor and predecessor change and point to one another. This happens without any governing body which is one of the most important factors of having a system without a single point of failure. So, it will be one of the most important components of the decentralized filesystem going ahead.

III. Deliverable 2 – Performance measure of Existing Systems

This deliverable was mainly focused on identifying the existing storage systems and studying their different features along with testing their performance. IPFS (Interplanetary Filesystem) [5], SWARM [7], Storj [8] and SIA [9] are a few of the many others which were shortlisted depending upon their feasibility and availability. All the filesystems leverage a blockchain network to enable their incentive layer for cryptocurrency transactions. This also provides prevention from data tampering as all the transactions are recorded in an immutable ledger. Following is an overview of the four storage systems.

A. **IPFS –** IPFS also known as the InterPlanetary Filesystem is a decentralized storage service which provides storage services for file sharing, storing digital assets, video hosting platform, hosting websites, etc. It uses content-based hashing which helps to ensure the integrity of stored data as well as locate the data on the IPFS network. It has a wide array of solutions for the developer community to host their applications using the filesystem. The installation and storage of data has been made really easy using the user interface and the command line interface provided by IPFS. It makes use of Filecoin [6] to incentivize the blockchain based solution. A timing test performed using the Java API libraries displayed impressive results. The taken to store a file as big as 1 GB took merely 6969 milli seconds to upload followed by 5365 milliseconds to retrieve the whole file.

B. **STORJ –** Storj is a decentralized cloud storage platform which can be accessed via an interactive web-based platform and a command line client. It has a monthly subscription model which allows for 75 TB of storage space and also generously provides monthly free storage of up to 150 GB. It makes use of Storj tokens for monetary transactions. The web platform can be used to create secure buckets which can use a pass phrase for encryption. Once the file is stored in the desired bucket Storj generates a link through which we can access the file contents. It also

displays a global view with highlighted physical location on a map where it is storing shards of our data. Upon testing the command line client with a 400 MB file it took nearly 751 seconds (nearly 12 minutes) to store it. This showed a vast performance difference compared to IPFS which could easily store and access a 1 GB file within 15 seconds.

C. **SWARM –** Similar to IPFS and Storj, SWARM provides a desktop application to run our own node and store data on its network. It has mainly been designed to develop decentralized applications around it and provides a JavaScript library and the BEE API for accessing and monitoring data with http endpoints. It uses Ethereum tokens and a chequebook contract to secure payments based on the negotiated storage incentives. In order to interact with the SWARM client, it requires a wallet setup. It does not provide any free tokens, in the beginning it provides a simple gateway interface on its website to store a file up to 10 MB. Upon storage it provides a swarm hash and a weblink to access and download the file.

D. **SIA –** SIA provides a desktop application containing a dashboard to store and retrieve the files. For around 1 TB storage space a user needs up to 1250 SIA coins. That nearly comes down to 3-4 $ per month which is comparable to the conventional cloud storages. The SIA dashboard also provides an option to host a node which can store data for other remote users. This allows for users with free disk space to earn some amount by renting it out to remote users. The cryptographic proofs i.e., smart contracts are used to make sure that the data will be stored safely.

From the above remarks we could conclude that IPFS performs better than Storj when it comes to uploading and retrieving a file. SIA provides an easily accessible functionality to host a node for storage and earn some tokens and SWARM is great when it comes to developing applications blockchain and supports the use of Ethereum which is a widely used cryptocurrency and provides support for a great

compile engine for authoring smart contracts in the form of solidity. Depending upon the desired functionalities and support studied in this survey could induce the final filesystem.

IV. DELIVERABLE 3 – FILESYSTEM STRUCTURE

In a conventional filesystem a master node can be used to keep a track of all the metadata related to one file and the nodes serving the file. The master can continuously interact with its servers to remain updated and serve client queries. Without a master it could be a challenging task to locate a file. For this purpose, in this deliverable I worked on implementing a file system structure which will be useful for locating multiple chunks of file irrespective of how small or big the file is.

A Merkle tree is a hash-based data structure which is also used in blockchains to create a digital fingerprint of all the set operations in a transaction for maintaining the total integrity of the data. Inspired by this approach the deliverable 3 will server three purposes by having the Merkle tree like file system structure:

a. It will allow for a decentralized storage architecture based on DHT to easily locate and store the file and its metadata.
b. It will ensure the integrity of data objects/chunks.
c. It can be applicable for files with variable size.

The algorithm first divides a file into multiple chunks of size 4096 bytes (4 KB) simultaneously creating a chunk object for every 4 kb object divided. The chunk object will contain multiple attributes like the Hash of all the bytes present in the object. The start and end of file pointers which will be important while recreating a file. The original bytes of data of the file and the hash its root which will be created after the Merkle tree gets generated going ahead. After the file gets divided into multiple chunks it gets sorted in an increasing order of their end of file counter. I have used a

SHA256 hash function to encode the data bytes. Further this list of chunk objects is used to create a Node object which has some additional attributes like the list of all the child nodes, hash of the node itself, a Boolean flag explaining its status on the tree (whether it's a root, or a leaf or neither) nor data. A bottom-up approach is used to create the Merkle tree structure. Each node could have at the most 128 child nodes. Firstly, all the leaf nodes which contain the actual data are calculated. A parent node is created by concatenating all the hashes of the leaf nodes and then applying the Sha256 hash function on it. In this way the tree gets created finally returning the root hash. This root hash will be useful in determining whether any data block is missing or has been tampered with. Following is an example of the Merkle tree structure.

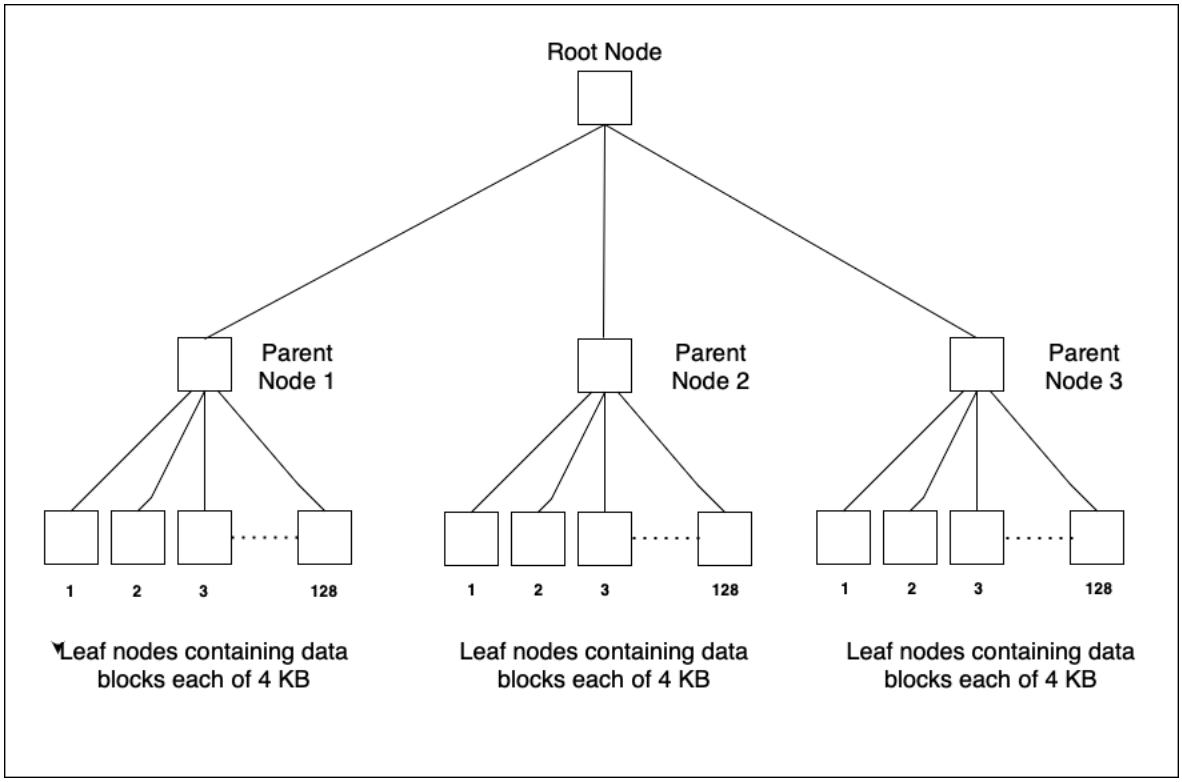

Figure 2: Merkle Tree structure for a 1.5 GB file

The above representation shows division of leaf nodes into 128 of size 4 KB to form 3 separate parent blocks and a root node as a result of the 3 parent blocks. This is a simple representation for a file as big as 1.5 GB. Depending upon the size of file the depth of the tree will vary.

Along with this I implemented a client which creates the Merkle tree structure and sends out the metadata file and data blocks over a socket connection and reconstructs the file and root to ensure its integrity. This can be implemented alongside the chord protocol or any other DHT for storage and retrieval of files.

V. DELIVERABLE 4 – SMART CONTRACT USING SOLIDITY

The fourth deliverable was focused on understanding smart contracts and using solidity to create and deploy a contract. Smart contracts are a contain transaction protocols which automatically executes and document events and actions as per the requirements. As suggested by Wang et al. [10] Smart contracts are computer protocols which ensure negotiation, automatic transaction facilitation and verification without any presence of central authorities. It can be deployed on a blockchain network for others to interact with and used to build interactive applications on a block chain. A smart contract can sit on a blockchain which is similar to a user account but controlled by a code. A contract account can have components like its Balance, Storage, and the code.

Solidity [11] is a programming language invented for authoring smart contracts which makes execution of a contract easy. Solidity uses the Solidity compiler to generate two files – the byte code and the Application binary interface which is used to interact with the deployed contracts. I generated a basic solidity contract to understand the syntax and to learn how to use the Remix development environment. Remix provides us with a Ethereum virtual machine, existing testing networks and access to multiple accounts with 100 ethers assigned to each account. This makes it easy to learn and deploy numerous contracts without worrying about our own crypto wallet. Remix has a graphical user interface which makes it easy to interact

with the created contract using different accounts and lets a user play with the parameters such as the Gas limit, value and its unit.

To get a better understanding of smart contracts a complex contract with a gambling game was developed for this deliverable. In the game 5 players are allowed to bet on a random number. The minimum amount required to enter the game is set to 500000 Wei (Wei is a smaller unit of ether). A user account inputs a value greater than 500000 and a random number and calls the enterPlayer() function to enter the game. The game waits for 5 players to enter and when the 5^{th} player enters it initiates the 'gamble' function which generates a random number and does further processing to choose the winner. The number selected by the users which is closest to the generated random number wins the game and the second closest number is considered as the runner up. Additional 250000 is transferred to the fifth player who enters the game as after initiating the gamble function it requires more gas to perform the operations which should not be unfairly charged to the 5^{th} player. 50000 wei is transferred to the dealer who deploys the contract as a commission. The 70% of the remaining balance is taken by the winner and 30% by the runner up.

As observed in the deployed game it is important to keep a track of number of operations to always minimize the operations cost. In a smart contract every transaction is always accounted for. Solidity provides inbuilt functions to track the cost of every bet and the current balance of the contract. It also allows for easy transfer of ether units to desired accounts. It has support for datatypes to store user account addresses and a mapping datatype similar to a HashMap in Java. Modifiers can be used to add any common condition statements

used by multiple functions to increase code reusability. I have made use of the 0.5.0 compiler version to develop the contract. Every version has seen a few syntax changes over the period.

Web3 library is used to interact with the Ethereum world using any other programming languages. A JavaScript framework called Mocha can be used for unit testing of the contracts and deploy them on a local network. This will help in incorporating the blockchain and adding an incentive layer to the filesystem which will allow users to trade ether tokens for storing data.

VI. CONCLUSION

For any blockchain based filesystems it is important to have an architecture which can sustain on its own without any leader to overcome having a single point of failure. This semester's deliverables were mainly focused on implementing a few important components in a decentralized filesystem. Starting with a distributed hash table as seen in the Deliverable 1 that can be used to make sure it is easily possible to locate any files based upon the key hash and the nodes address hash vale. It was also important to test the existing systems which serve similar purpose in order to get a sense of additional functionalities being offered, their performance and their shortcomings. Deliverable 2 was followed by developing a filesystem structure which could be used with various Distributed Hash Table algorithms going forward and also take care of the metadata along with ensuring that the data has not been tampered with. Finally deliverable 4 was significant to get familiar with smart contracts and work on developing a simple gambling game using Solidity. The research and implementation work for the final deliverables was crucial to dive deep into the working of a decentralized filesystem and to understand the basic flow and functionalities of the existing systems. All the four

deliverables will be used as building blocks to implement the whole file system to serve the main objective of the project.

Lastly this project report is a consolidated record of all the four deliverables carried out during the semester and will serve as the starting point for CS 298. The final aim for the project would be to develop a fully functional decentralized filesystem based on blockchain which could be used to store files of various sizes and access them from any node in the network using a specific identifier.

REFERENCES

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

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” presented at the - 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010, pp. 1–10, doi: 10.1109/MSST.2010.5496972.

[3] H. Huang, J. Lin, B. Zheng, Z. Zheng and J. Bian, "When Blockchain Meets Distributed File Systems: An Overview, Challenges, and Open Issues," in *IEEE Access*, vol. 8, pp. 50574-50586, 2020, doi: 10.1109/ACCESS.2020.2979881.

[4] I. Stoica *et al.*, “Chord: a scalable peer-to-peer lookup protocol for Internet applications,” vol. 11, no. 1. pp. 17–32, 2003.

[5] welcome to the ipfs docs,” *IPFS Docs*. [Online]. Available: https://docs.ipfs.tech/ . [Accessed: 07-Dec-2022]

[6] Filecoin, “Filecoin Docs,” *Filecoin*. [Online]. Available: https://filecoin.io/build/#usp . [Accessed: 07-Dec-2022]

[7] Swarm team, “SWARM: Storage and communication infrastructure for a self-sovereign digital society,” Jun. 2021 [Online]. Available: https://www.ethswarm.org/swarm-whitepaper.pdf

[8] “Storj: Documentation overview,” *Storj DCS Docs*. [Online]. Available: https://docs.storj.io/ [Accessed: 07-Dec-2022]

[9] “Introduction to these docs,” *Introduction to these docs - Sia Docs*. [Online]. Available: https://docs.sia.tech/get-started-with-sia/intro . [Accessed: 07-Dec-2022]

[10] [2020] A Detailed and Real-Time Performance Monitoring Framework for Blockchain Systems. P. Zheng, Z. Zheng, X. Luo, X. Chen and X. Liu, 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2018, pp. 134-143.

[11] [2021] Blockchain smart contracts: Applications, challenges, and future trends. Khan, S.N., Loukil, F., Ghedira-Guegan, C. et al. Peer-to-Peer Netw. Appl. 14, 2901–2925 (2021). https://doi-org.libaccess.sjlibrary.org/10.1007/s12083-021-01127-0

[11] *Solidity*. [Online]. Available: https://docs.soliditylang.org/en/v0.8.17/. [Accessed: 07-Dec-2022].