By Ajinkya Rajguru

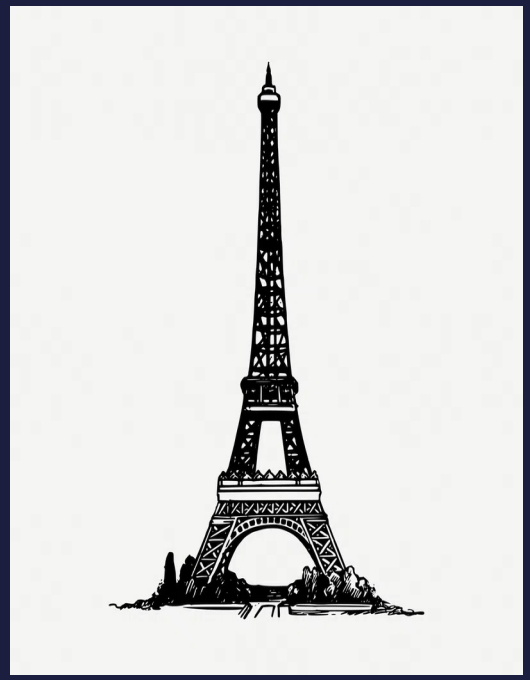
High Performance Distributed File System Based on Blockchain

- Advisor: Dr. Chris Pollett
- Committee: Dr. Ben Reed
- Committee: Sanmesh Bhosale

Agenda

- Introduction
- Background
- Technologies used
- Important Components
- System Architecture
- Workflows
- Results
- Conclusion

Fun Fact



An average person stores 500 GB of data in their cloud storage.

If the physical representation of this data (CDs, papers, photos) were stacked on each other, it would be **three times the size of the Eiffel Tower** [1].

Introduction: purpose

- The amount of data generated and consumed by a user is increasing daily.
- Drawbacks of conventional storage
- Need for alternate storage systems.
- Possibility to make profits from unused storage spaces on personal devices.

Introduction: Goals

- Develop a fault-tolerant decentralized file store.
- Ability to easily store, retrieve, and delete data
- Robust storage along with maintained data integrity.
- Fast storage with no single point of failure.
- Use of blockchain for monetization and transaction transparency.

Background: Distributed File Systems

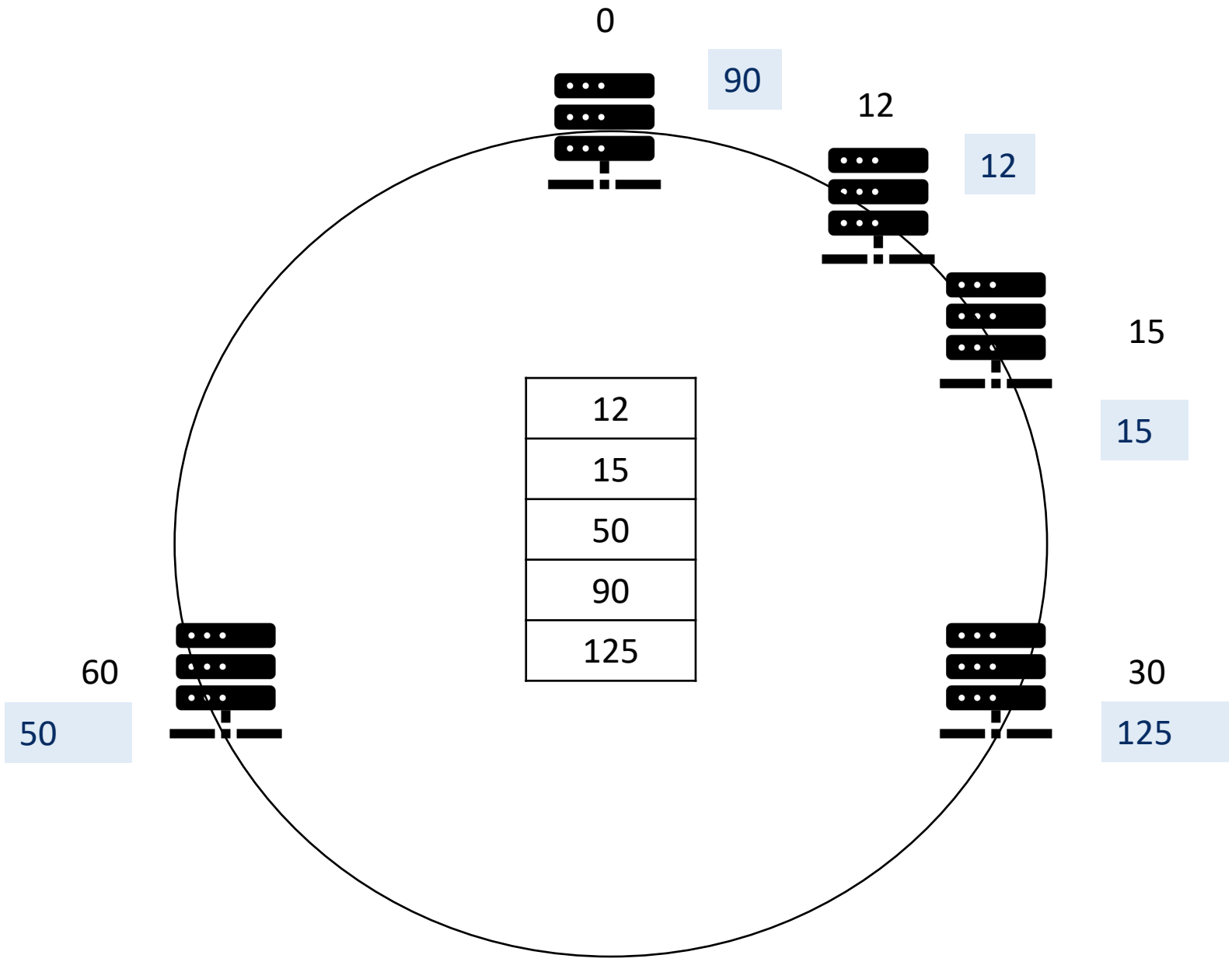
- Allows for storing of data on commodity servers connected using a computer network
- Replication – Increase the reliability of the system by having redundant data.
- Sharding – Makes data more manageable and increases the performance of the system.
- Conventional systems like HDFS [4] and GFS [3] require a master server to manage metadata and commodity servers.

Background: Distributed Hash Table

- Provides a decentralized lookup based on Key-value pairs.
- Highly scalable
- Fault-tolerant

Background: Chord DHT [5]

Hash function: **mod 100**



Background : Blockchain

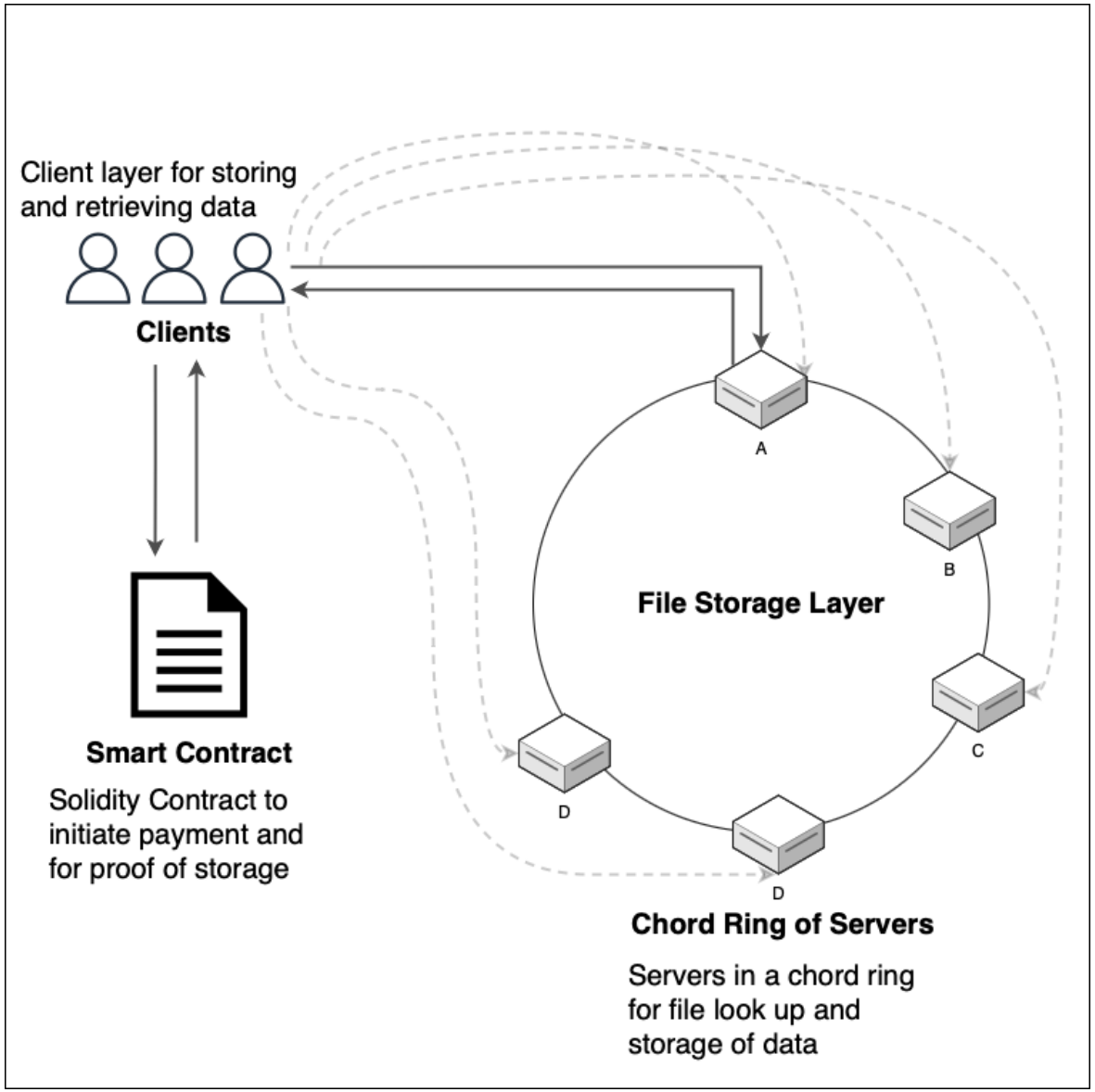
- A decentralized and public digital ledger
- Securely stores transactions
- Unalterable
- Ethereum – decentralized blockchain with smart contract functionality
- Ether - Native cryptocurrency of Ethereum

Background : Solidity Smart Contract

- Programs stored on the blockchain run after predetermined conditions are met [2].
- Solidity – A programming language designed for developing smart contracts that run on Ethereum [3].

Technologies used

System Architecture



Important Components: Client

- To store data on the filesystem
- Data retrieval
- Data deletion
- Encryption
- Divide data into blocks of 64 KB
- Make payments

Important Components: Client

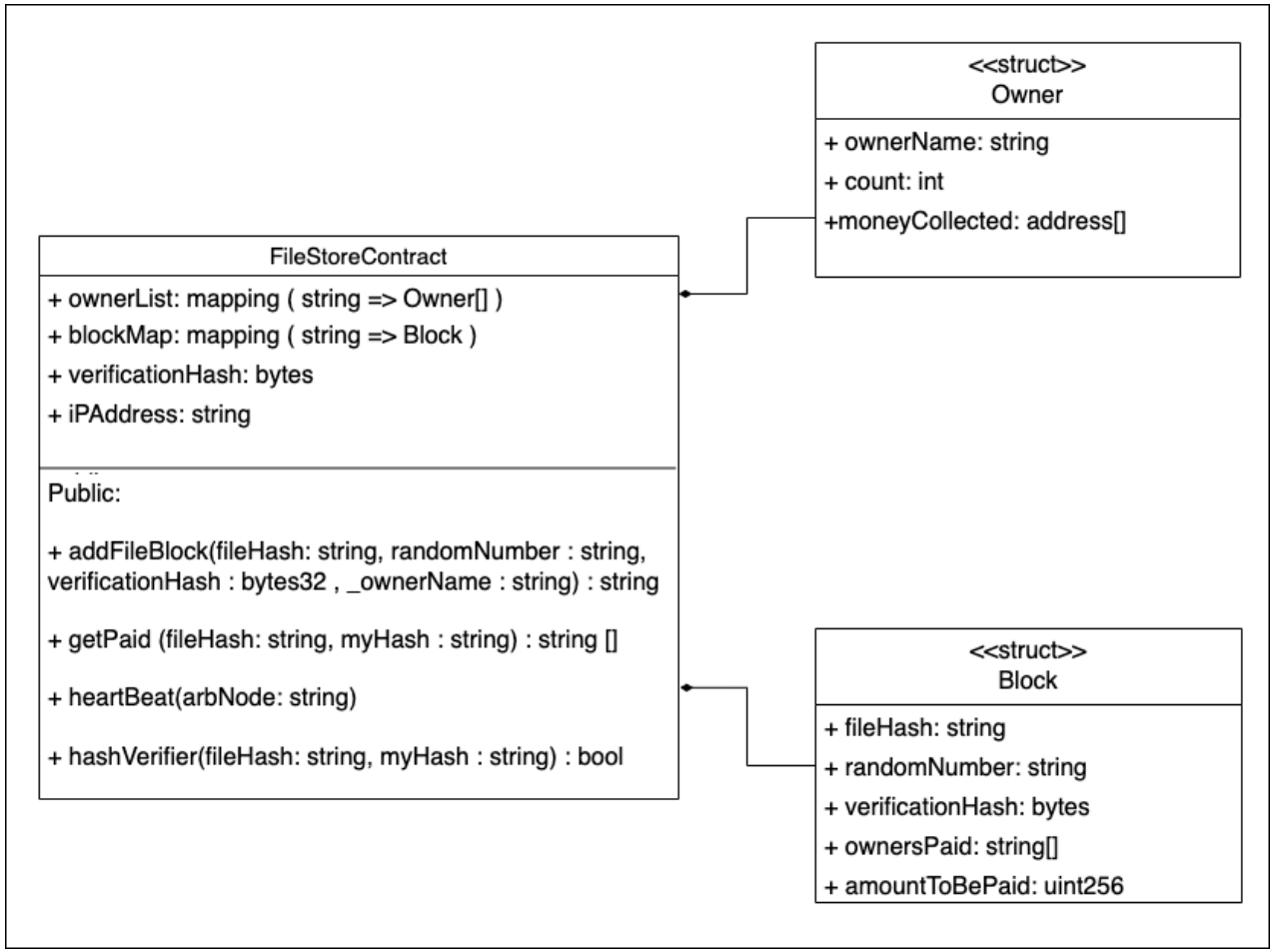
Requirements to participate as a client

1. Java 11+
2. An empty port
3. The IP address of the blockchain network
4. Contract address
5. Crypto wallet (loaded with Ether)

Important Components: Smart Contract

- Introduces an incentive layer
- Transparency of transactions
- Verification for storage
- Payment
- To store/get the IP address of an arbitrary server.

Important Components: Smart Contract



Important Components: Server - Chord

- The hash function used: FNV-1A [6]
- This generates a 32-bit hash for each server derived out of its IP address.
- Significant function: n.findSuccessor(id)
- Threads:
  - FixFingers()
  - Stabilize()

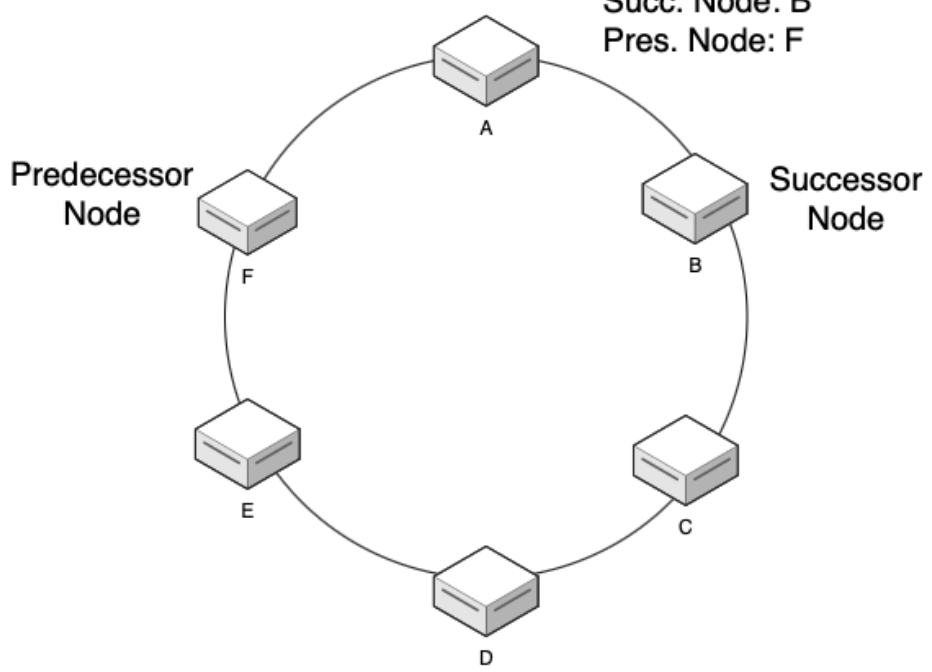
Important Components: Server - Chord

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

Finger Table		
Index	Start	Serving Node
1	1930734897	B
32	4078218544	E

Successor List	
B	C
C	D
D	E

IP Address: 127.0.0.1.8080
Node Hash: 1930734896
Succ. Node: B
Pres. Node: F



Chord Ring

Important Components: Server - Filesystem structure

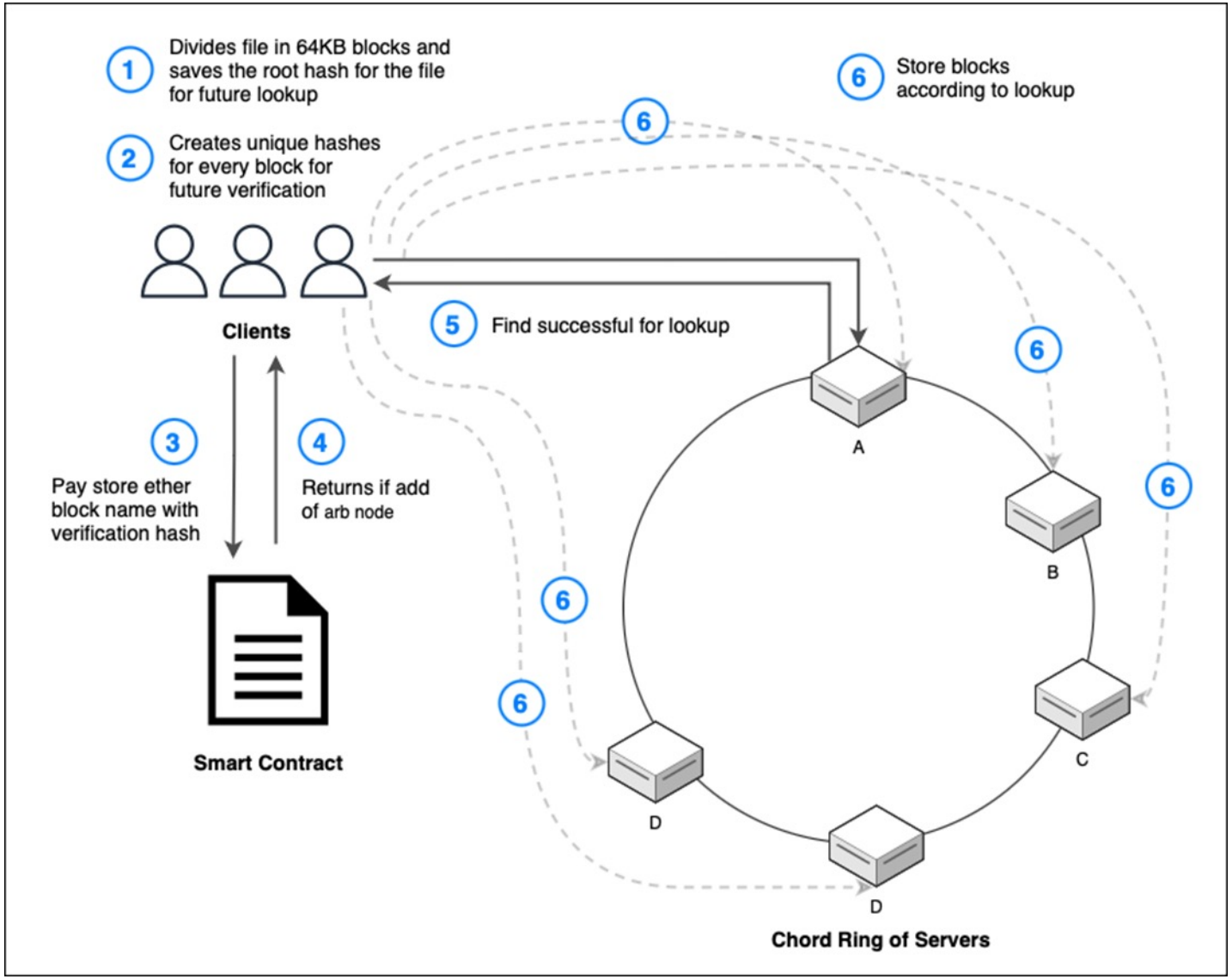
- The blocks received are of two types:
  - A content block
  - File details blocks
- Data is replicated on two other servers:
  - Successor
  - Predecessor
- The stabilizeFileStore thread performs four major functions:
  - Re-distribution of data if a new node is added
  - Garbage collection
  - Replication
  - Payment calls

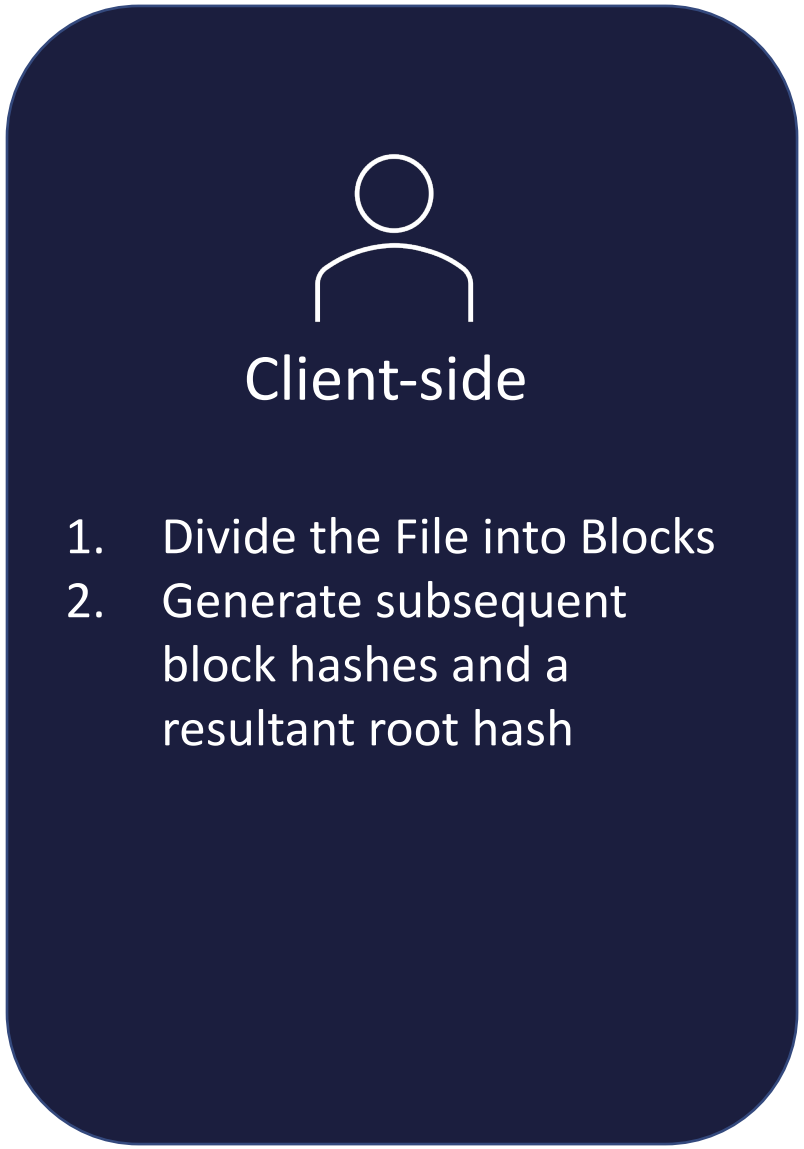
Important Components: Server

Requirements to participate as a server

1. Java 11+
2. An empty port
3. The IP address of the blockchain network
4. Contract address
5. Crypto wallet (loaded with Ether)

Workflow: Store File





Store Sample.pdf on the FS

Sample.pdf

Block 1

Block 2

Block 3

Hash1 = SHA-256(encrypt(byte[],K))

Hash2 = SHA-256(encrypt(byte[],K))

Hash3 = SHA-256(encrypt(byte[],K))

Root Hash = SHA-256(Hash1 + Hash2 + Hash3)

Smart Contract

1. Store Block information on the Smart Contract
2. Send Ip address of an arbitrary node

Block 1

Hash1 = SHA-256(encrypt(byte[],K))
R
Verification_Hash
Owner

Block	**Block hash** **Random Number** **Verification Hash** **Owner** **Amount**

Send the IP address of an arbitrary node to the client.

Client-side

1. Select a Random Number
2. Generate Verification Hash

Block 1

Hash1 = SHA-256(encrypt(byte[],K))
Let arr = encrypt(byte[],K)

Generate a random number **R** between *0 – arr.length*

Rand_byte = Byte representation of **R**

RandByte[] = Replace **Rth** index from *arr* by **Rand_byte**

Initial_Hash = *Keccak***(RandByte**[])

Verification_Hash = *Keccak***(Initial_Hash + R**)

Server: getPaid()

1. Select a Random Number
2. Generate Verification Hash

Block 1

Hash1 = SHA-256(encrypt(byte[],K))
Let arr = encrypt(byte[],K)

Read random number **R** from the contract

Rand_byte = Byte representation of **R**

RandByte[] = Replace **Rth** index from *arr* by **Rand_byte**

Initial_Hash = *Keccak***(RandByte**[])

Send the **Initial_Hash** to the contract for verification

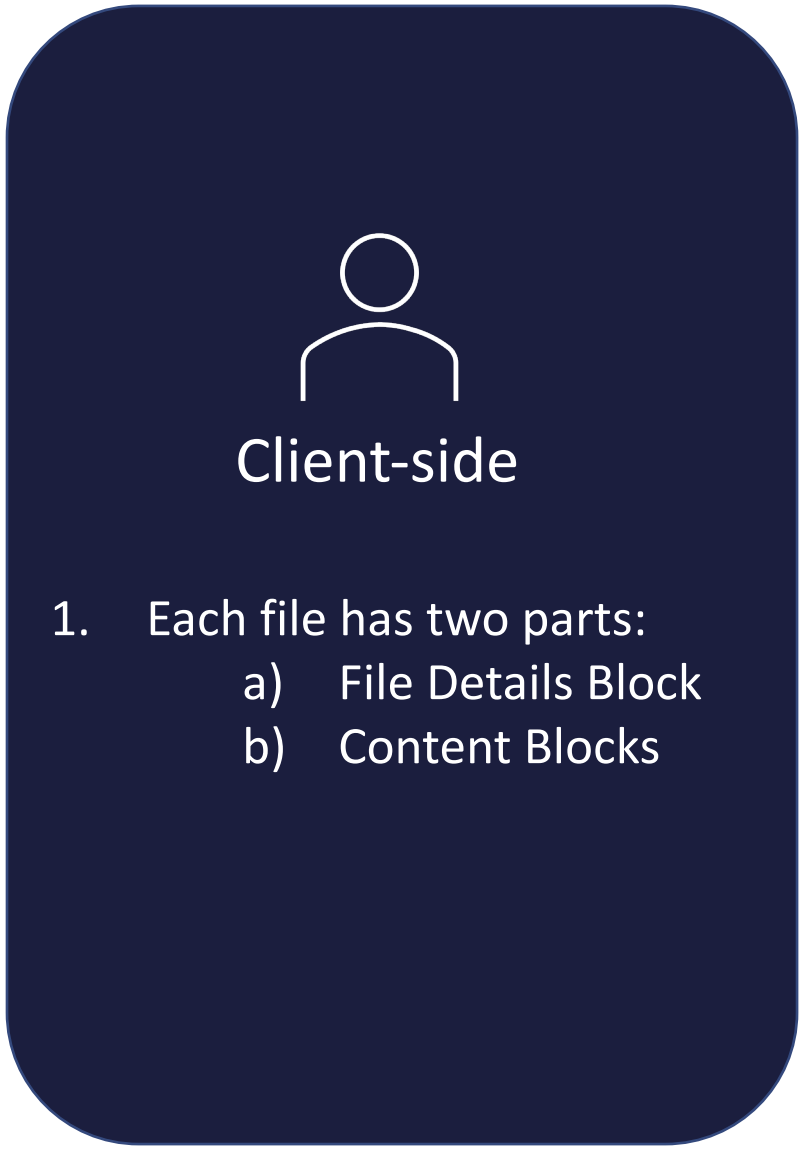
Calc_Verification_Hash = *Keccak***(Initial_Hash + R**)

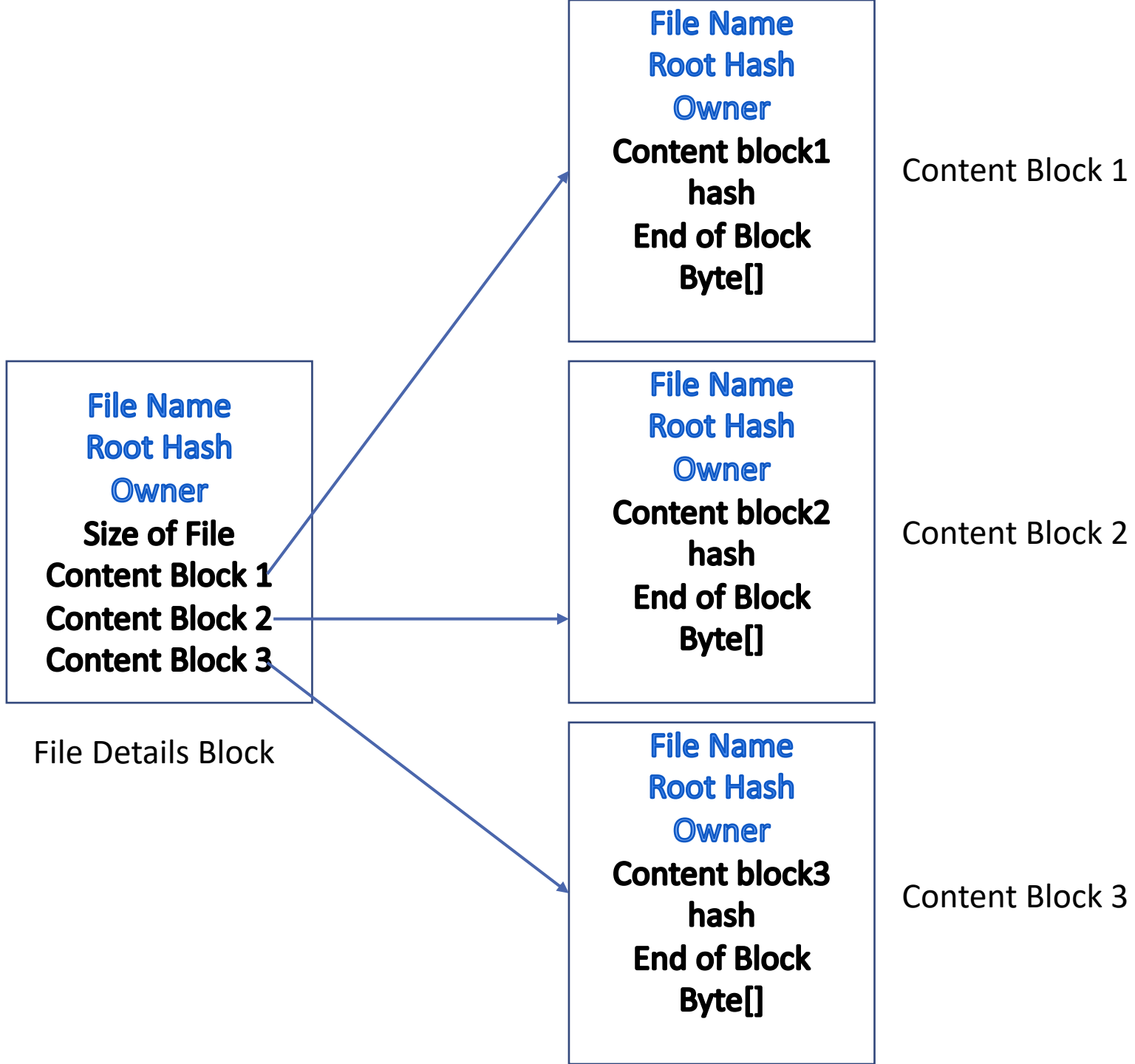
If Verification_Hash == Calc_Verification_Hash then Pay

Client-side

1. Each file has two parts:
    1. File Details Block
    2. Content Blocks

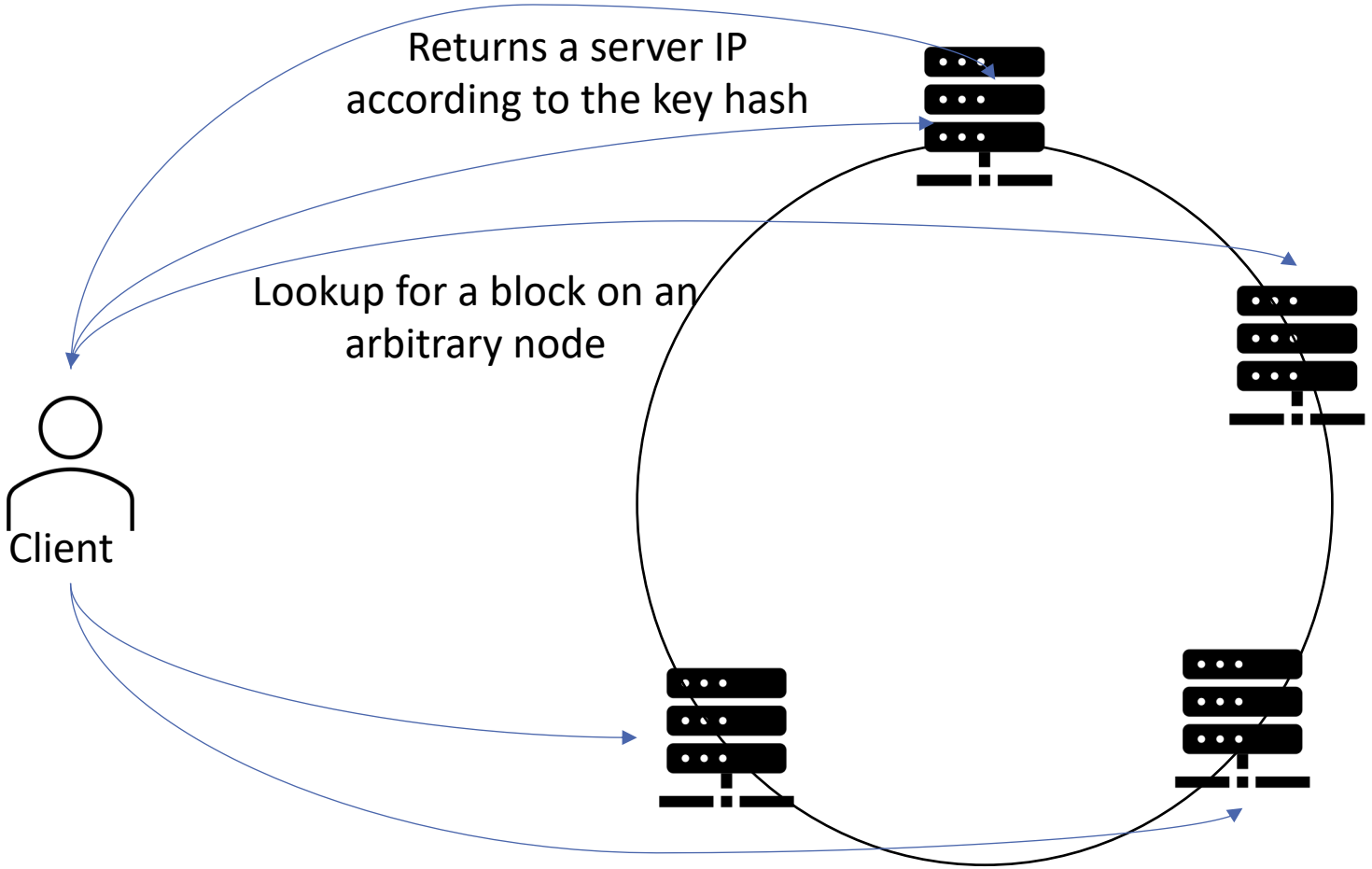
File Details Block **(Primary metadata file)**	**File Name** **Root Hash** **Owner** **Size of File** **Content Block List**
Content Blocks **(file containing data bits)**	**File Name** **Root Hash** **Content block hash** **End of Block** **Byte[]**


Client-side
1. Each file has two parts:
a) File Details Block
b) Content Blocks


File Name
Root Hash
Owner
Size of File
Content Block 1
Content Block 2
Content Block 3
File Details Block
File Name
Root Hash
Owner
Content block1
hash
End of Block
Byte[]
Content Block 1
File Name
Root Hash
Owner
Content block2
hash
End of Block
Byte[]
Content Block 2
File Name
Root Hash
Owner
Content block3
hash
End of Block
Byte[]
Content Block 3

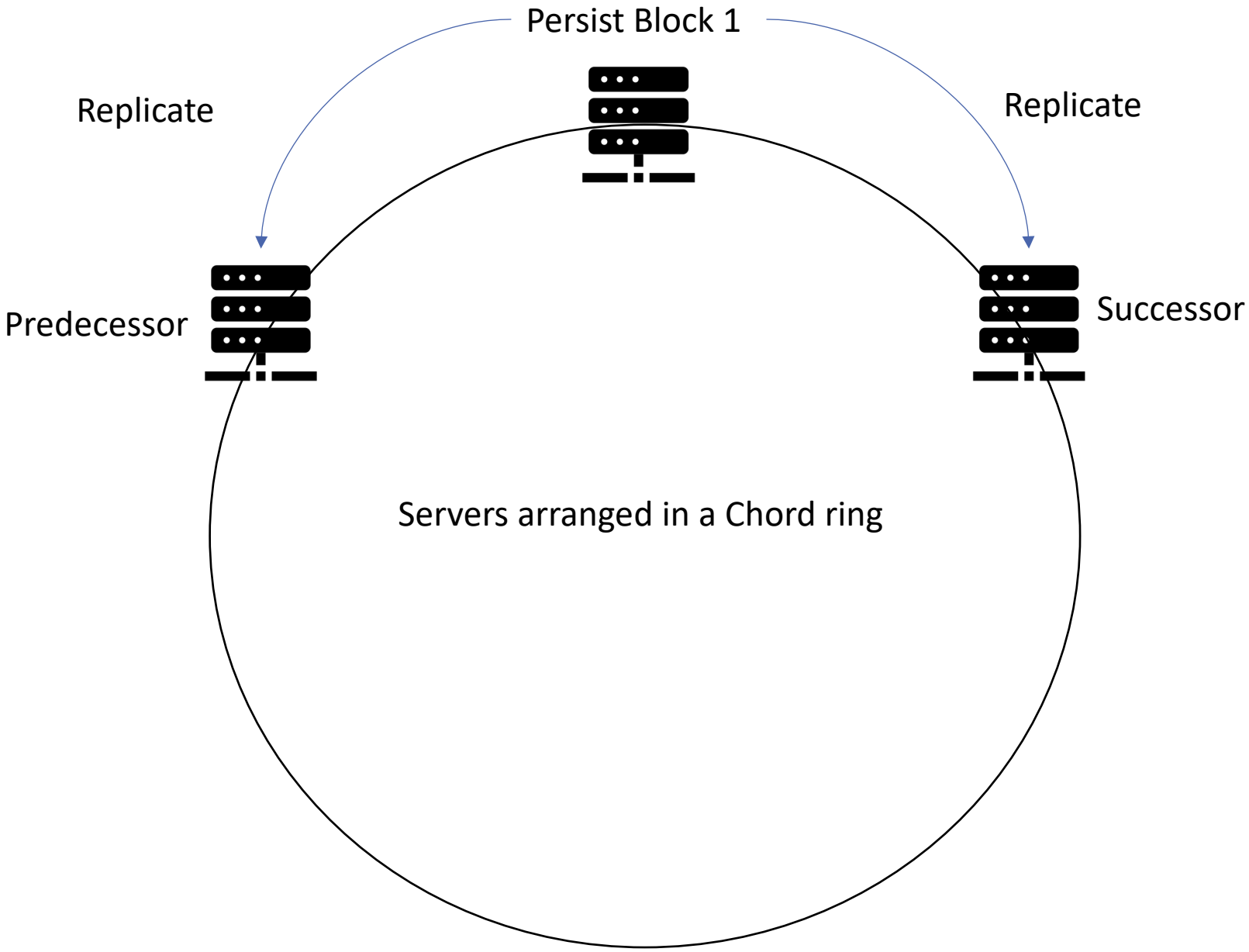
Workflow: Store File

1. Make a DHT lookup call for each block.
2. Send the blocks for storage on their respective servers according to the lookup

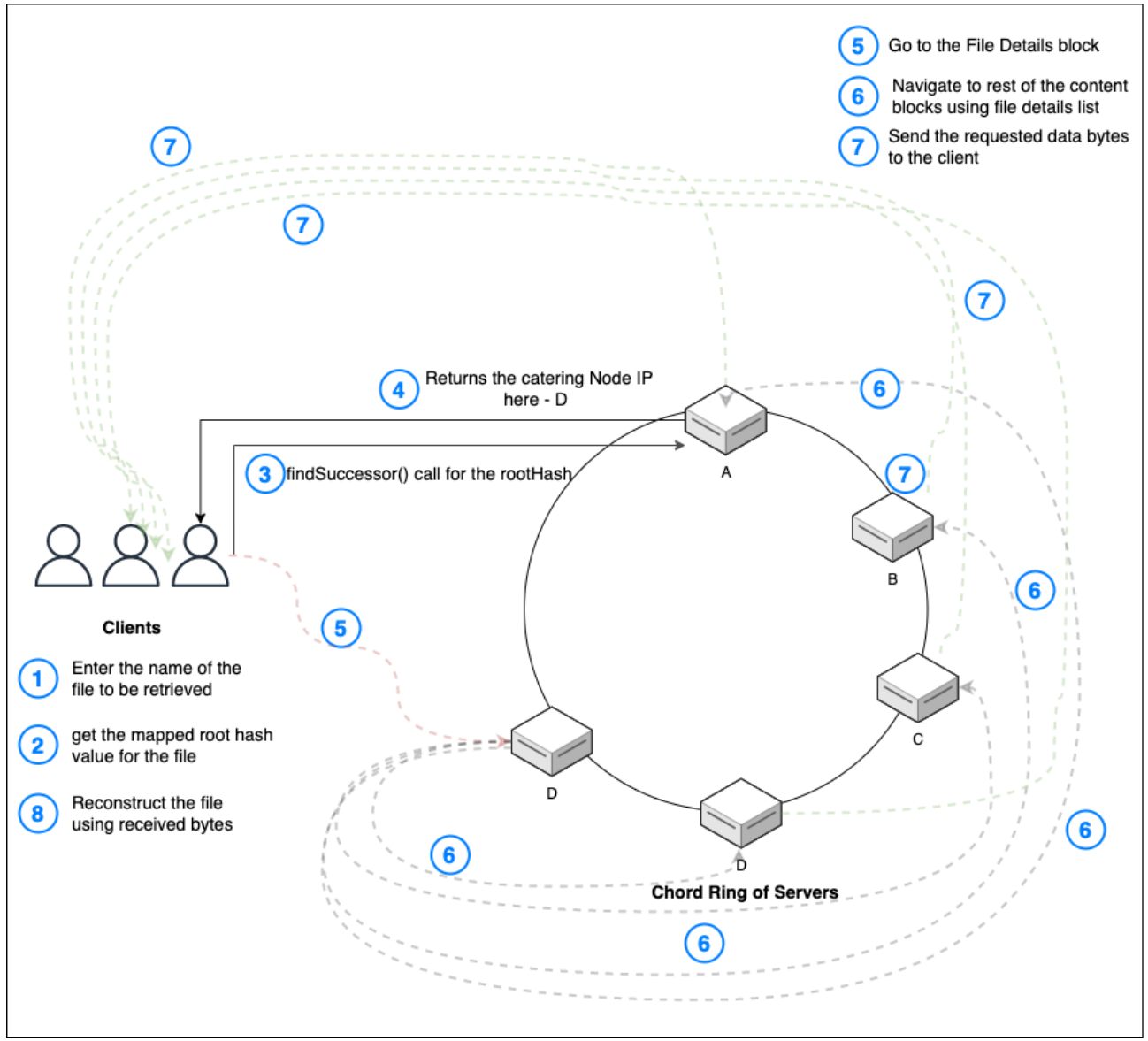


Server Node:
File System and Chord

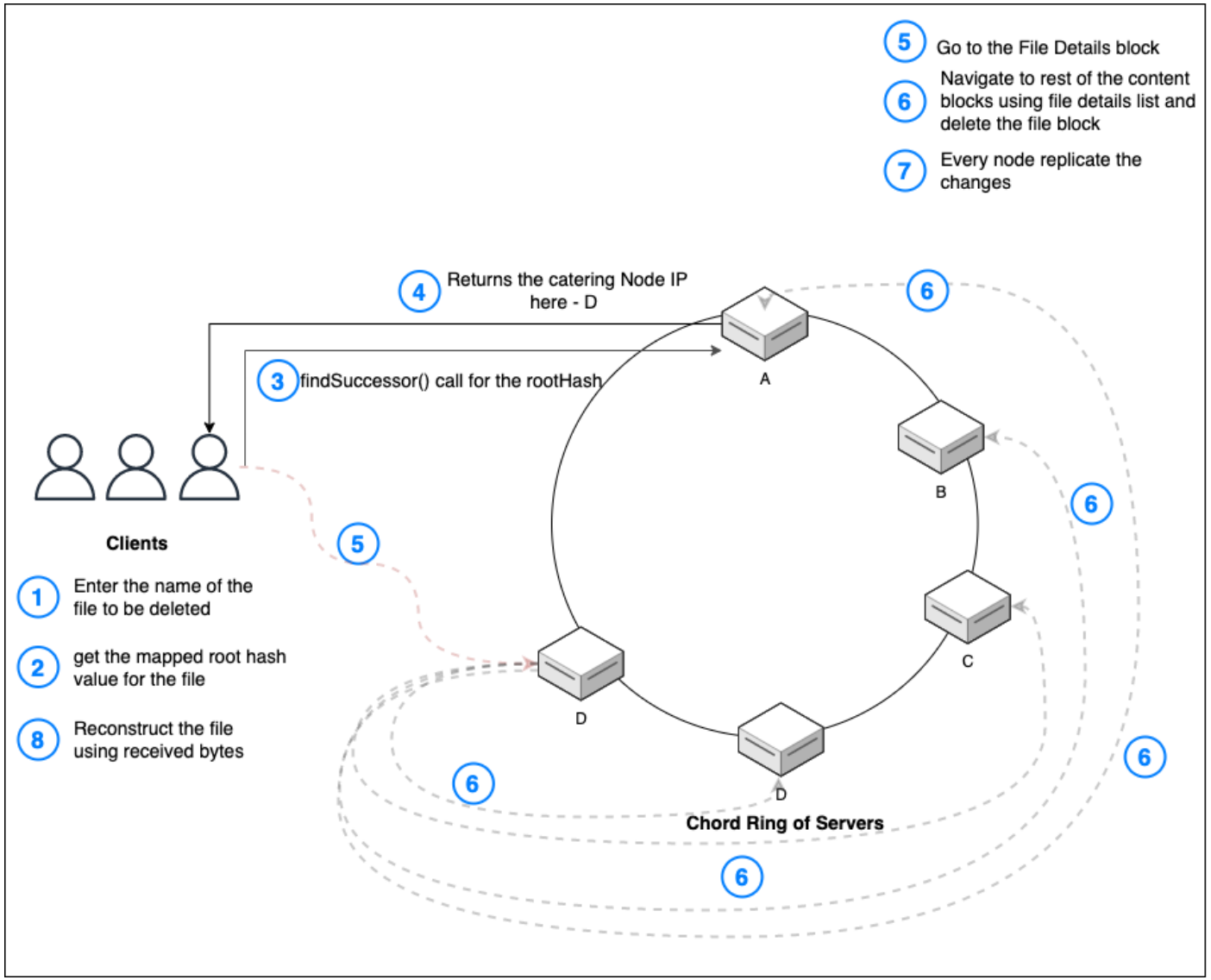
1. Persist the block
2. Replicate it on successor and predecessor nodes



Workflow: Retrieve File



Workflow: Delete File



Results

Cumulative gas consumption to store file data 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

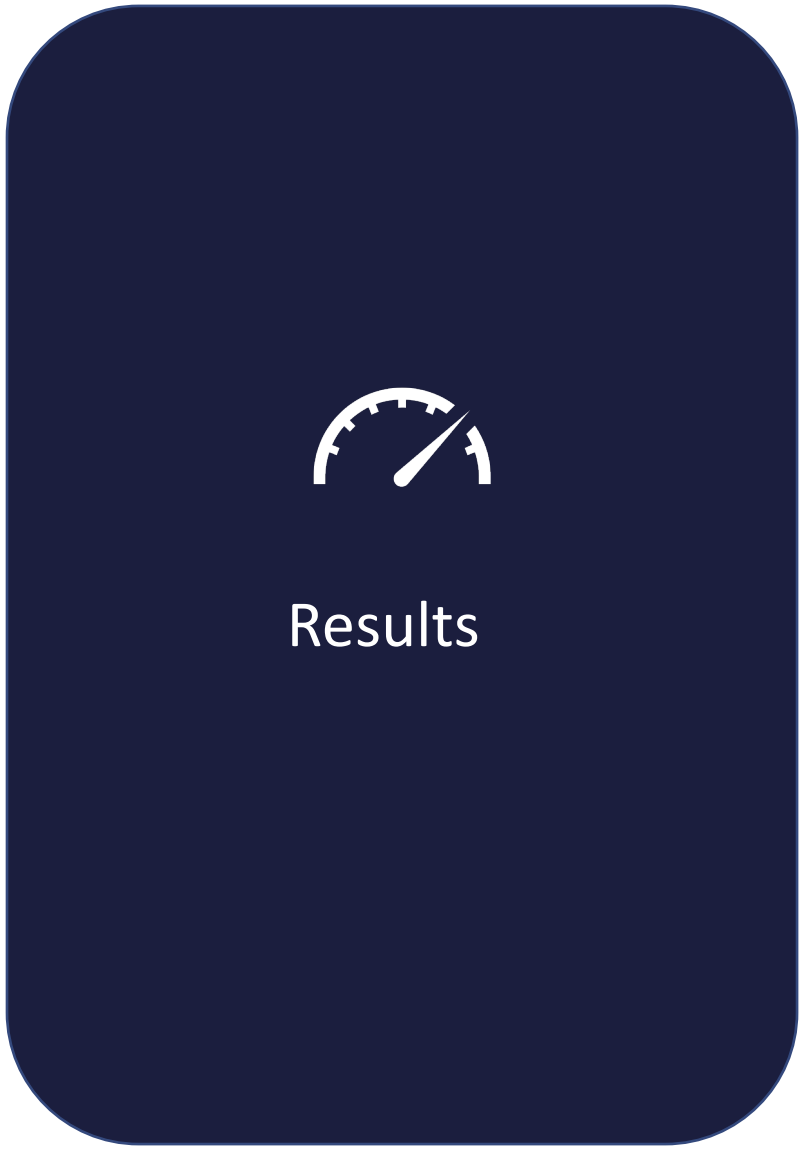
A server requires on average **360000** gas while calling getPaid()

Results

The client requires an average of 200 milliseconds for every block of 64 KB

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



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	22770 ms	48279 ms

Table 3: Performance Testing on 15 Nodes

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

Conclusion

The developed file system boasts impressive benefits:

- Great performance
- Lucrative rewards
- Robust storage verification
- Effortless replication
- Secure
- An enticing option for users seeking to maximize their storage capabilities and make the most of their available space.

Thank you

References

[1] 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).

[2] Smart Contract https://www.ibm.com/topics/smart-contracts

[3] 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.

[4] 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.

[5] 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.

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

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