Distributed Systems CS739
Final Report

P2P DHT Storage using Chord

Madhav Kanbur
Himanshu Pandotra
Lincolnspartacus James
Pavithran Ravichandiran

1 Introduction

Our goal is to build a peer-to-peer decentralized storage system. P2P systems are of great importance given the rising amounts of censorship in today's world. We decided to implement our storage system on top of the well known Chord [1] protocol. We chose Chord for its simplicity and ability to scale well. Some of the salient features of our storage system are the following:

- Replication of Data for High Availability
- Flat namespace w.r.t each client
- Multiuser support with Elliptic Curve Cryptography authentication
- Higher fault tolerance by having a configurable length successor list

In the next section we briefly review the Chord protocol.

1.1 Chord Overview

Chord consists of a collection of nodes connected to each other in a ring (as in Fig 1). Chord relies on consistent hashing to distribute the load across multiple storage host nodes. In consistent hashing, the output range of a hash function is treated as a fixed circular ring. In such a ring, the largest hash value wraps around the smallest hash value. Each node in the system is assigned a hash identifier which is computed by hashing the node's IP address and port number. Further, each data file is identified by a key computed by hashing the combined string of client's public key and file name, and then walking the ring clockwise to find the first node with a position larger than the item's position. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. For example, for a file having the hash computed as 8, we find the successor of this ID as N10 and store the file in it.

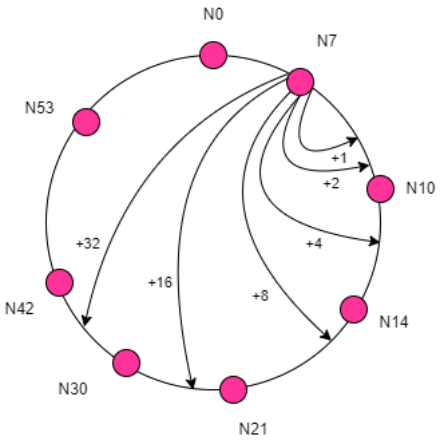

Figure 1: Chord Ring

Entry (i)	Id+2^i	Successor
0	N7+1	N10
1	N7+2	N10
2	N7+4	N14
3	N7+8	N21
4	N7+16	N30
5	N7+32	N42

Table 1: Finger Table for Node 7

To accelerate lookups, Chord maintains additional routing information. Each node has a finger table that acts as a dynamic routing table for node lookups across the chord ring. For an m bit chord ring, we have $2\hat{m}$ chord identifiers. The value of m is 6 in above chord ring. The algorithm proves that having m entries in each finger table performs node lookups in O(log N) time. Identifier entries in the finger table increase exponentially which in turn lets us to search the space in logarithmic time. A finger table entry includes both the Chord identifier and the IP address. Note that the first finger of n is the immediate successor of n on the circle; for convenience we often refer to the first finger as the successor.

The following APIs exposed for clients that use our storage service. Also, the internals APIs of the chord system are also listed. We'll look at these APIs in details in the following sections.

Client Exposed API

- getFile(f) : Returns file F to client
- putFile(f) : Stores file F on the coordinator node

Internal Chord API

- join(n) : Join existing ring via node n
- findSuccessor(k) : Find the coordinator for key k
- notify(n') : Updates predecessor to n' (if conditions are met)
- replicateFile(f) : Replicate file f on this node
- getSuccessorList() : Returns successorList
- stabilize() : Verifies if successor is accurate, calls notify() on successor
- syncSuccessorList() : Maintains the successorList
- fixFingerTable() : Maintains the finger table

Chord Join:

In this section, we'll look at what happens when a node tries to join the P2P chord ring. The steps involved are detailed below. (as shown in Figure 2)

- The new node contacts the bootstrapper to find an entry point in the chord ring.
- We are assuming that Bootstrapper returns any of the available chord nodes randomly.
- The new node contacts the returned node and sends a request to find the new node's successor node.
- The randomly selected node finds the successor of the new node and returns it to the new node.
- Once we've found the successor, the new node updates its successor link to the returned node.

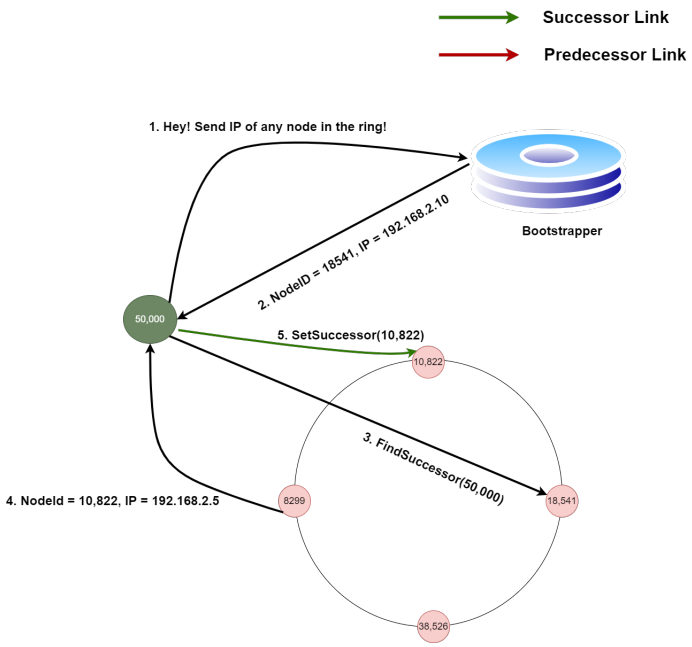

Figure 2: Chord Join

Stabilizing the chord ring: Stabilise function in chord protocol ensures the predecessors and successors nodes are up to date (Figure 3). The function is periodically run in a separate thread on the background on every node at a rate of 3 to 6 second window. The stabilize thread asks the successor about its predecessor and checks if it equals itself. If not, it checks if the new node lies within the key space between itself and to its successor. If yes, it sets the new node as its successor. Then

it notifies the new node about itself, so the new node can set its predecessor correctly (Figure 4).

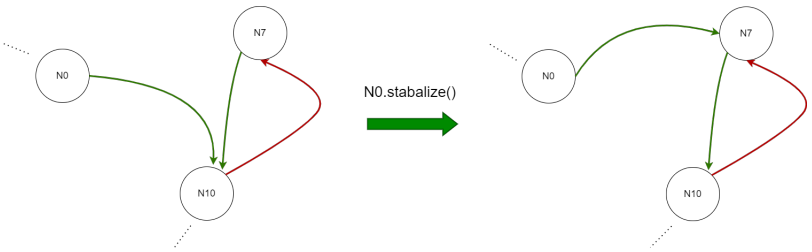

Figure 3: Stabilize

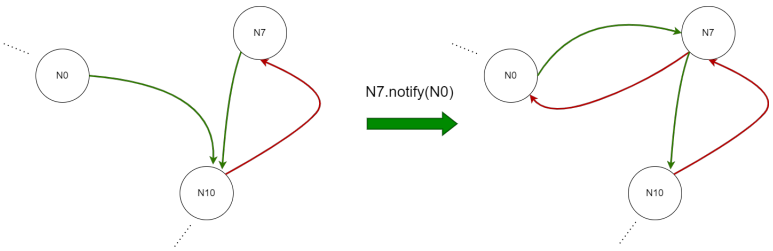

Figure 4: Notify

Successor list: Instead of keeping one successor, we maintain a successor list for fault tolerance. The successor list length is configurable and it also decides the replication factor. If a successor has crashed in the system, it picks the next entry from the successor list and sets it as the new successor. The successor list is kept up to date by periodically running a synchronize successor list thread. The thread fetches the successor list from its own successor every 3-5 seconds once and removes only the last element and appends everything to its own list. (Figure 5)

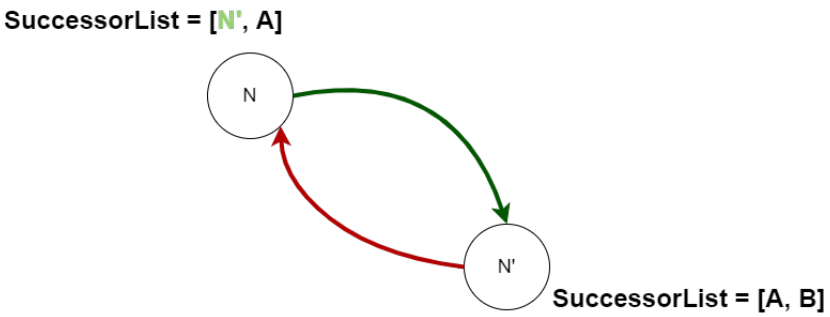

Figure 5: Successor list update

1.1.1 Key Replication for Fault tolerance

For every hashed key, a particular node acts as the coordinator. This is typically the node next to the keyID that is part of the chord ring. Every node maintains a SuccessorList() which contains the address of its N immediate successors where N is the replication factor. Every node also maintains two lists namely OwnerList and Replicated List. The ownerList contains the keys for which the node acts as coordinator and the Replicated List contains the keys which are stored on this node by its predecessors for fault tolerance. The coordinator replicates its keys (and data) to the nodes in its successorList. This helps the system from tolerating node failures without loss of data. For example, in Figure 6, Node N10 is the coordinator node for Key K9. Moreover, it replicates the key K9 to nodes N14 and N21 which are present in its successor List.

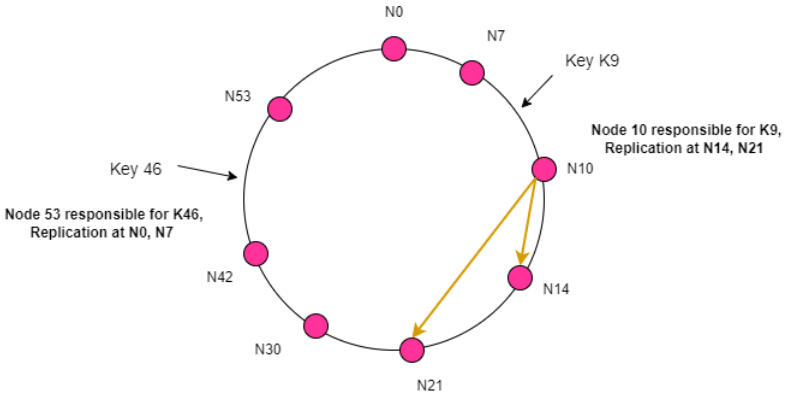

Figure 6: Key 9 with N10 as coordinator node and N14,N21 as replication nodes

1.1.2 Node Crash and Recovery:

Crash: When a node crashes, its immediate successor becomes responsible for the keys previously held by the crashed node. Since this successor node already maintains the data in its replicated List, it just transfers these keys to its owner list and becomes the coordinator node. Moreover, all new put() requests previously handled by crashed node are now handled by its successor. For example in Fig 7, as the node N10 crashes, put requests for key 8 are handled now by node N14.

Recovery: When a node N recovers, its predecessors and successors eventually come to know about its existence. The successor transfers a subset of its keys in ownerList to this new node corresponding to the section in the ring for which this node is now responsible.

Also, the predecessors of this node N replicate its keys to this new node. This is shown in Fig 7(right). Node N10 recovers leading to node N14 transferring the key K9 to node N10. Also, N0 and N7 replicate their respective keys K82, K90 and K3,K6 to node N10. (N10 stores these in replicatedList).

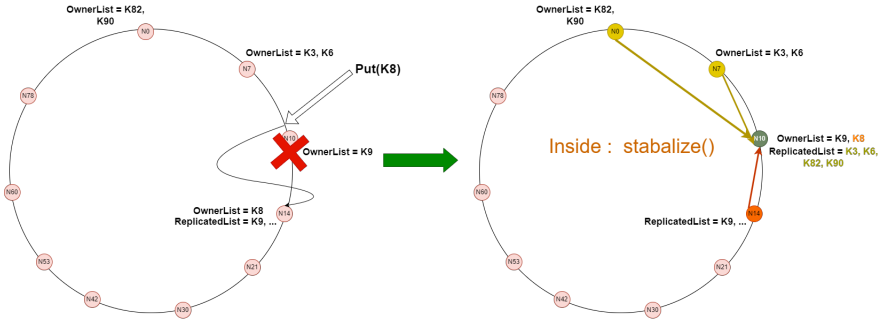

Figure 7: Node Crash and Recovery

1.1.3 GetFile API

In Figure 8, we show the steps involved in getFile() request. The filename is passed as a parameter and SHA-1 hash is used to create the key along with the user's public key. The client will contact the bootstrapper for a node in the chord ring (happens only once, not on every getFile()). After receiving a random node from the bootstrapper, the client will try to contact the chord node and will try to locate the node responsible for it. Using finger table entries associated with each node, the responsible node is found in O(log n) hops. After locating the responsible node, the client will directly contact the node responsible for it. It sends the file hashed value, signature and public key. Client will authenticate this request and streams the file back if the client has required permissions. The chunk size for the streaming request is set to 64MB.

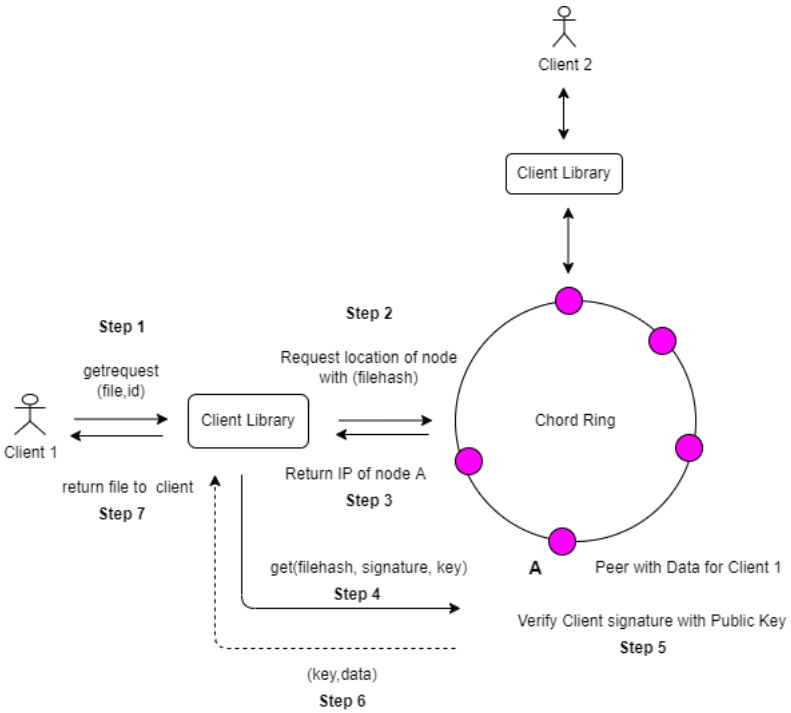

Figure 8: getFile() request overview

2 Demos

We recorded 3 demo videos that showcase the following:

- Basics : In this demo, we perform a Put() operation such that the coordinator node is 18,541 (shown in Figure 9). We then verify that it has been stored on the coordinator node and the 2 successor nodes (for replication) using our RPC Debug interface. Finally we do a Get() on it as a sanity check. (Video Link).

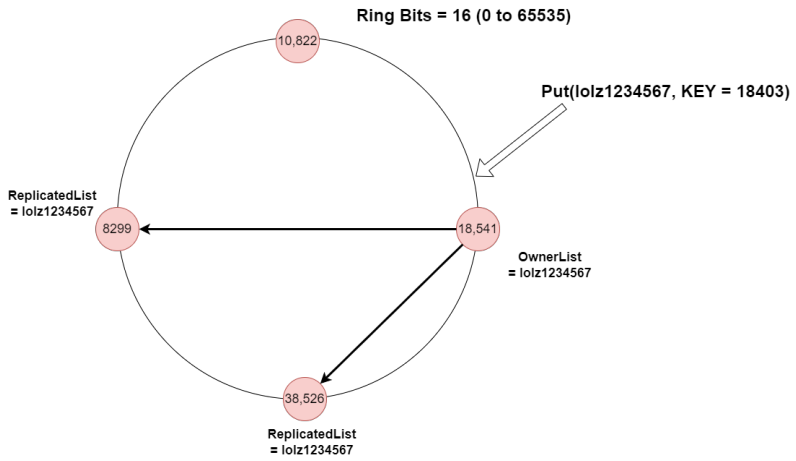

Figure 9: Put() operation Demo. Verify data is stored on coordinator and two successor nodes. (Video Link).

- Availability : Continuing from our previous example, we now try doing a Get() after crashing node 18,541. As expected, the request is routed to 38,526 and it serves the Get(). (See Figure 10)

  (Video Link).

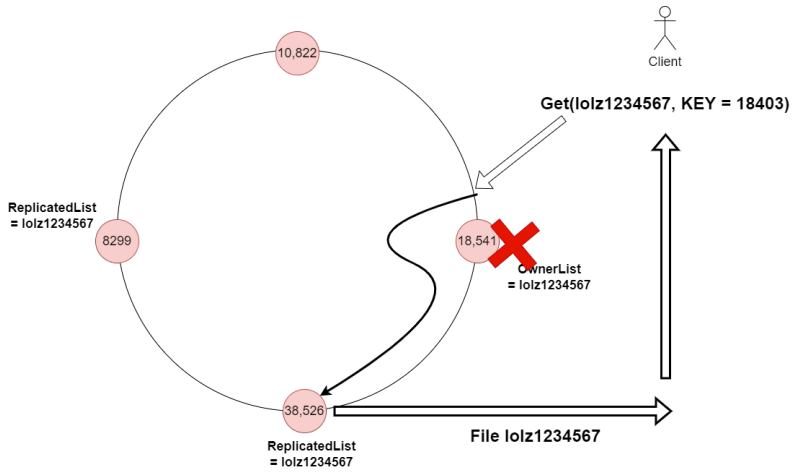

Figure 10: Demo for check Availability of system after node crash. (Video Link).

- Crash Recovery + Key Transfer : With node 18,541 still down, we perform 2 Put() operations such that 2 kinds of transfer take place on recovery (Figure 11) -
  - From Successor's (38526) OwnerList to Node's OwnerList (calvin.mp3)
  - From a Predecessor's (8299) OwnerList to Node's ReplicatedList (test.m)

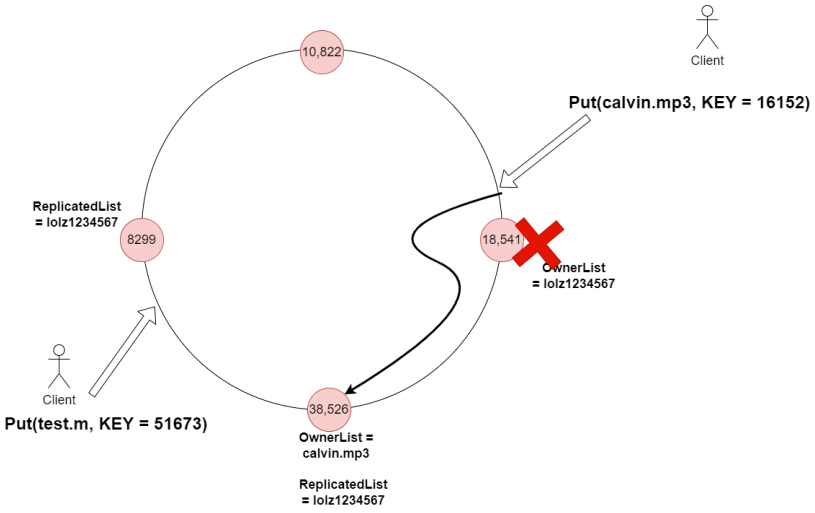

Figure 11: Crash Recovery + Key Transfer

Figure 12 below shows how the keys are transferred when Node 18,541 boots up. Node 8299 notices that its SuccessorList has changed and initiates a transfer of all keys in its OwnerList (Yellow Arrow). This happens inside the stabilize() method of Node 8299.

At the same time, Node 38,526 gets notified by 18,541 of its presence. Node 38,526 scans through its OwnerList to find all keys belonging to (10,822, 18,541] and initiates transfer of those keys (Red Arrow).

(Video Link).

2.1 Implementation

We implemented Chord from scratch in Python and used gRPC for communication. Contrary to the popular belief of things being easy to code up in Python, we found it more challenging owing to the non-statically typed nature of Python, inconceivable run-time exceptions etc. Long live C++!

2.2 Testing/Verification

Our Chord interface exposes a debug() RPC for observing the current state of nodes in the ring. It can be used to verify the values of all pointers, keys, finger table entries etc. We used this interface to verify the correctness of our implementation in

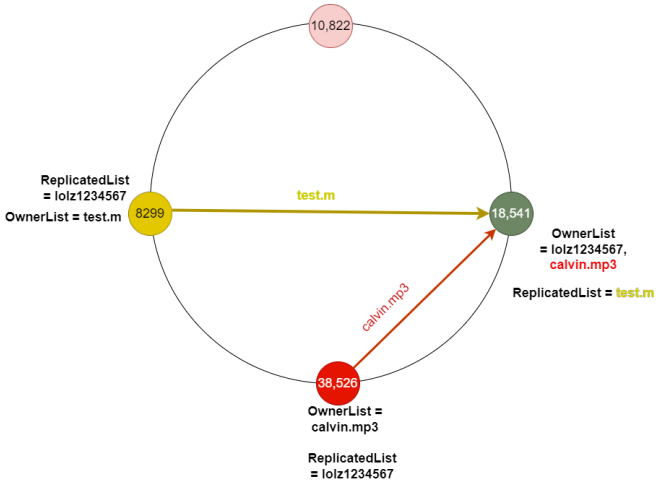

Figure 12: Recovery after Bootup. (Video Link)

various situations such as node crashes, node recovery etc. We observed that after a node crashes, there is a tiny window during which the crashed node is still present in the finger tables of various nodes, leading to failed lookup() requests until the ring stabilizes again. However, this failure is masked by our client library's retry mechanism.

3 Experiments

3.1 Latency

We measured the effectiveness of the chord algorithm. Initially, we measured the find successor latency without the finger routing table. Meaning we only had the circular links which routed the node lookup in linearly time proportional to the number of the nodes in the chord ring.

We observe from the Figure 13 that as the number of peers/nodes in the ring increases, the node lookup delay increases linearly without the finger table but it remains constant with the finger routing table. This shows that the node lookups are performed with logarithmic number of hops efficiently according to the chord protocol.

We also measured how the read and write latencies increase as we increase the client file size. The relevant results can be found at Figure 14.

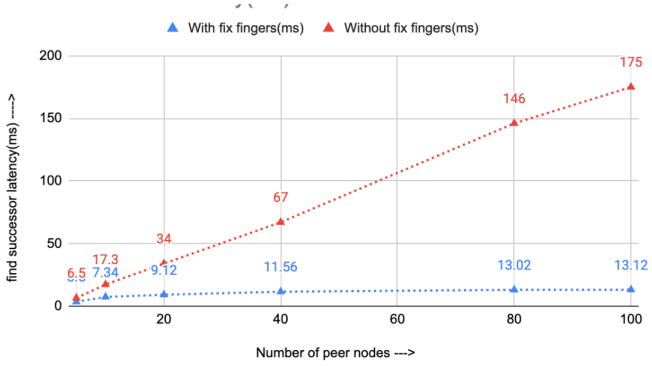

Figure 13: Find successor Latency(ms) vs No. of Nodes

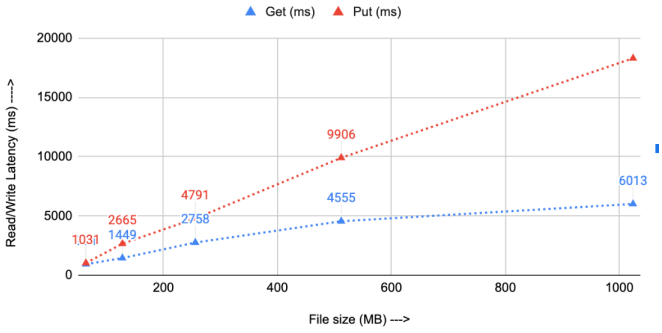

Figure 14: Read Write Latency vs File Size

3.2 Throughput

We observe the overall performance of the chord ring by measuring the combined throughput of all read and write requests that the system handles at a time. First, we fix the number of concurrent clients accessing the chord nodes and measure the throughput of simultaneous read requests distributed over the ring. For this, we spawn K ring nodes and N client processes and write 32MB data files(per client) per client to the ring. We vary K and N(5 and 10) and observe the write throughput in Fig 1. Similarly, N clients simultaneously read data from the ring via get() requests and we measure the system throughput as shown in Fig 15. We observe from Fig 15 and Fig 15 that as the number of peers/nodes in the ring increases the read and write throughput also increases. This shows that the keys in the ring are being evenly redistributed as the nodes increase leading to better efficiency of processing the requests and load balancing.

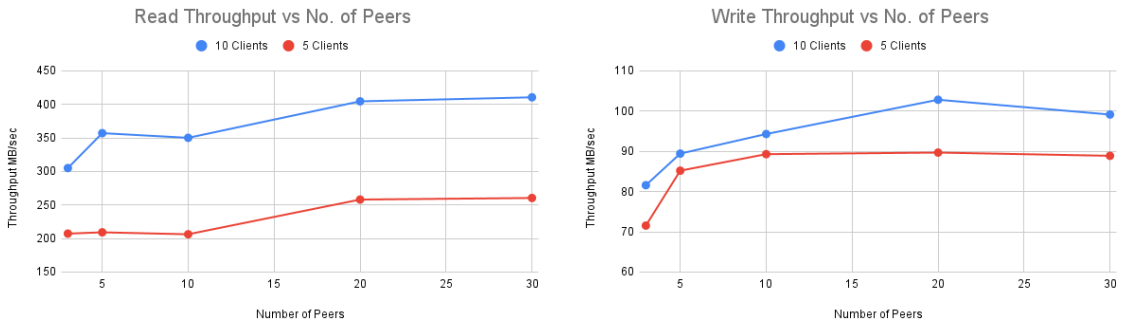

Figure 15: (a) Read Throughput vs Number of Peers. (For 5 and 10 clients) (b) Write Throughput vs Number of Peers. (For 5 and 10 clients)

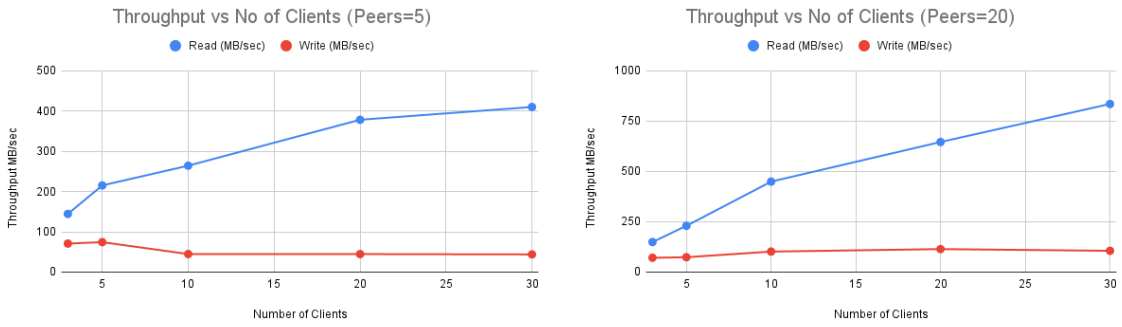

Figure 16: (a) Throughput vs Number of Clients (Peers/Nodes=5) (b) Throughput vs Number of Clients (Peers/Nodes=20)

In Fig 16 and Fig 16, we plot the change in Throughput with the number of clients at a fixed ring size. As the number of concurrent clients increases, the system Read throughput increases showing that the system has enough capacity to serve more clients. This is true for both cases when Peers = 5 and Peers = 20. Though the Read

rate is higher when more nodes are there in the ring i.e Peers=20 as expected for a larger ring.

But, we also observe that the write Throughput does not increase with more clients as expected but instead remains constant. We attribute this behavior to the cost of Put() in our implementation. Since we fsync the data to the node on every call, the number of fsync's increases as the number of clients increases. Thus the total data movement is limited by slow fsync's for data persistence.

3.3 Node Churn

The design is tested for its accuracy with having the peer nodes joining and leaving at different rates as shown in Figure 17. The chord ring is initiated with 15 peer nodes. The rate R of 0.05 indicates one node joining and leaving every 20 seconds. The number of lookup failures are observed by varying the node join/leave rate. As expected, the number of lookup failures continues to grow when the churn due to node joining/leaving the ring increases.

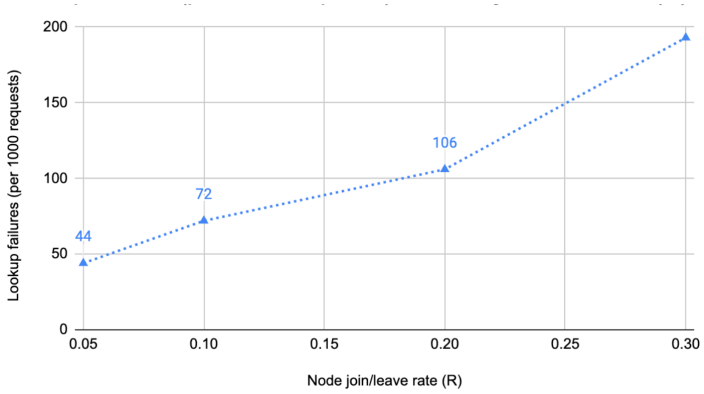

Figure 17: Lookup Failures (per 1000 requests) vs Node Join/Leave Rate

3.4 Load Distribution

To measure the load balancing properties of our ring, we inserted 510 keys into our ring of 100 nodes. The process of picking keys was as follows - we created 510 files name abc1.txt, abc2.txt ... and so on. Since changing even one byte in SHA1 should generate a different hash as shown in Figure 18 below because of the avalanche effect, we expect the hashes of abc_i.txt to be different and hence uniformly distributed on the entire ring.

However, we noticed the standard deviation of our key distribution to be 3.82720848 (Normalized value = 0.75) for an average number of 5.1 keys per node, which is unexpectedly higher. Intuitively, from the Figure 19, it looks like there are way too many files getting mapped to the range 10000-32000 instead of being uniformly distributed in 0-65535.

- `SHA1("The quick brown fox jumps over the lazy dog")`
  - Outputted hexadecimal: `2fd4e1c67a2d28fced849ee1bb76e7391b93eb12`
- `SHA1("The quick brown fox jumps over the lazy cog")`
  - Outputted hexadecimal: `de9f2c7fd25e1b3afad3e85a0bd17d9b100db4b3`

Figure 18: SHA-1 Avalanche Effect

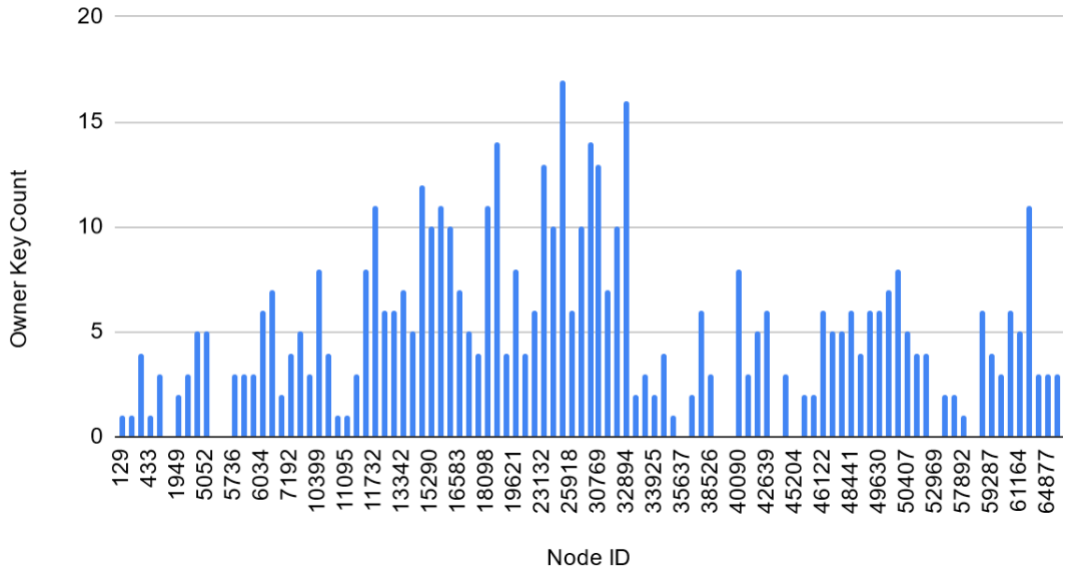

Figure 19: Key Distribution, Nodes=100, Keys=510

4 Conclusion

We have successfully implemented the peer to peer file storage based on chord routing protocol. Each node had a couple of threads like stabilise, replicate, fixfingers, etc running asynchronously in the background which lead to some interesting debugs in our design. Overall, it is a good learning experience on how to design peer to peer systems from the scratch.