PROJECT REPORT

1. Goals

The goal is to comprehend how peer-to-peer systems like Gnutella, Napster, and Chord are built. We developed a Distributed Hash Table P2P system based on the Chord protocol and evaluated its functionality, latency rate, hash table distribution, and normal performance.

2. Design and implementation.

Our system consists of a chord ring with **n** nodes. Each node is assigned a unique ID (within 2^m (123456 in this case)) by hashing the ip+port of that node by the SHA-1 Algorithm. A node can create a DHT ring, other nodes can join the ring by contacting this node using its ip and port number.

Each node maintains a **peerTable,** which contains the information of every other node present in the network. The peer table has n-1 entries where n is the total number of the nodes present in the chord ring.

Other than information in the peer table. Each node has a known successor and a predecessor, which means each node has information about its successor and predecessor. The successor node and predecessor node play an important role when a new node is added or a node is removed from the chord ring.

The system has a replication factor as a parameter and it will use its successors to replicate it. This is handled by the get feature as well. If the system which has data in it fails, it will query its successor and retrieve the data.

File Architecture:

1. **node.py:** This file defines a structure of a node with the following data structures
    a. Bit length - 2^m bits in the chord virtual ring.
    b. IP - IP of the current node
    c. Port - port on which the current node is listening.
    d. Hash: calculated by hashing ip+port by SHA-1 algorithm
    e. hashTable: contains the key-value pair datastore
    f. Successor's IP, port and hash
    g. Predecessor's IP, port and hash

h. peerTable: a list storing the list of info of each node in the ring(IP, port, and Hash). The information stored in the peerTable is sorted in increasing order of hash to find the closest number greater than the given hash.

Some basic functions to generate a hash, set the IP, port and hash values of the successor and predecessors, and insert values in the hash table and peer table are also defined in this file.

2. **Chord.py:** This file implements the Chord algorithm. Some important functions defined in this file are:
    a. start_server(): This function starts the chord server and accepts the connections of the clients and also performs several operations.
    b. find_neighbours(curr_hash,peer_ip,peer_port): When a new node is added in the chord ring this function finds the predecessor and successor of the node whose hash is passed. It also takes care of the keys that are to be stored in the new node.
    c. getupdate_peerList(self,succ_hash, ip, port): After the node is added, successor and predecessor are set. This function is called to update the new node's peer list.
    d. removeNode(): This function removes the node from the chord server and adjusts the keys in the successor and predecessor.
    e. perform_operations(): All the above operations are performed by the server and these are sent to servers in the form of a query. This function creates a query that is to be sent.

3. Assumptions and justifications of the design.

For the sake of simplicity, we have made the following Assumptions:

1. Key and value with text and space as delimiter.
2. The data is not persistent.

4. Achievements and discussions on underachievement.

a. We are able to implement a chord ring. For the number of resources we have at the moment, we have decided to go for a complete graph approach instead of the finger table to build a university scale peer to peer network.

b. The complete graph approach leads to improvement in the time taken to find a key as there will be only 1 hop. But it is not space efficient as every node is storing data of every other node in the ring. We are using a Binary search algorithm to find the corresponding node in a peer table and we find it in order of log n.
c. All types of error handling is done in the case of adding and removing the new node. Where to put keys after adding or deleting a node is also taken care of.
d. We also implemented replication of the keys and its error handling in case of insertion and deletion of the node.
e. We have not tested the churn of the nodes and its performance during the churn. We have tested for latency, replication, adding new nodes and removing new nodes.

5. Discussions on changes of plan, if any.

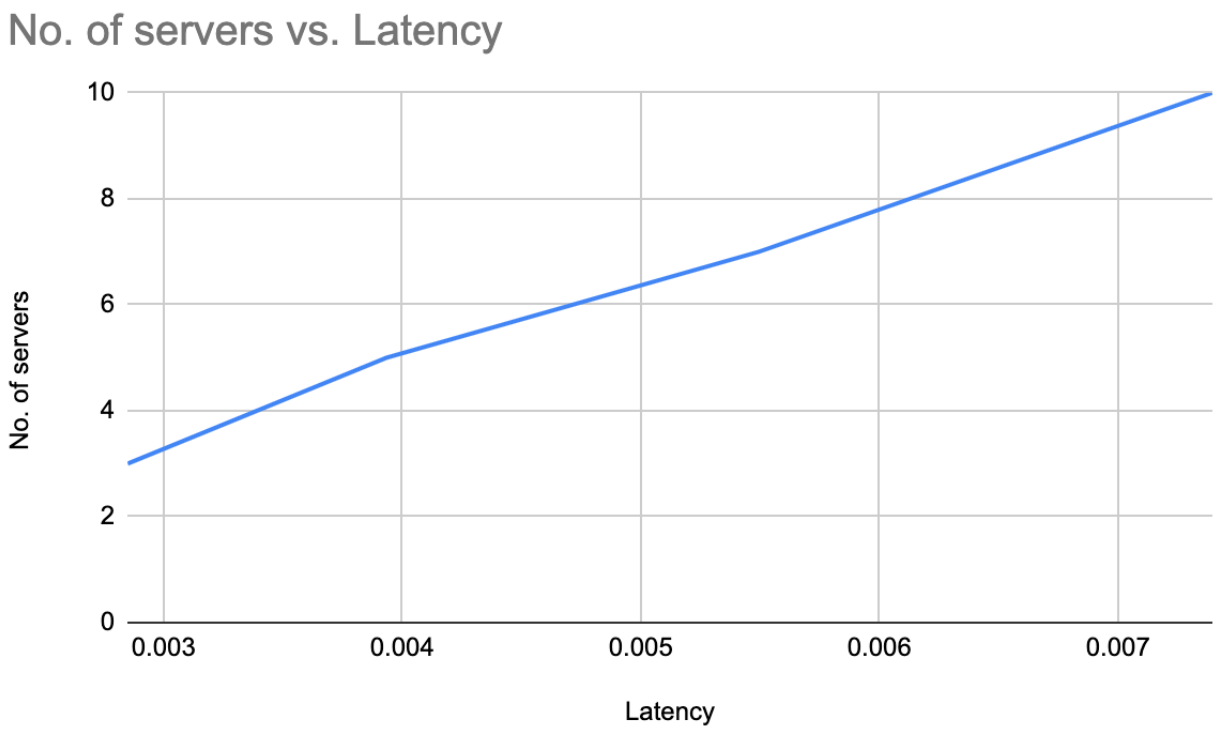
Initially, we planned to replicate the entire Chord architecture but due to constraints in resources and time, we decided to modify some designs and implement a university scale chord like distributed hash table system. We also skipped the idea to implement a cache of hot data on each server finger table to see how it can improve the performance.

6. Evaluation

When we talk about latency here, we mean both read and write. The logic is the same for reading and writing in our implementation and we mean latency for a read/write call.

1. Graph for a circular ring without the Peer table

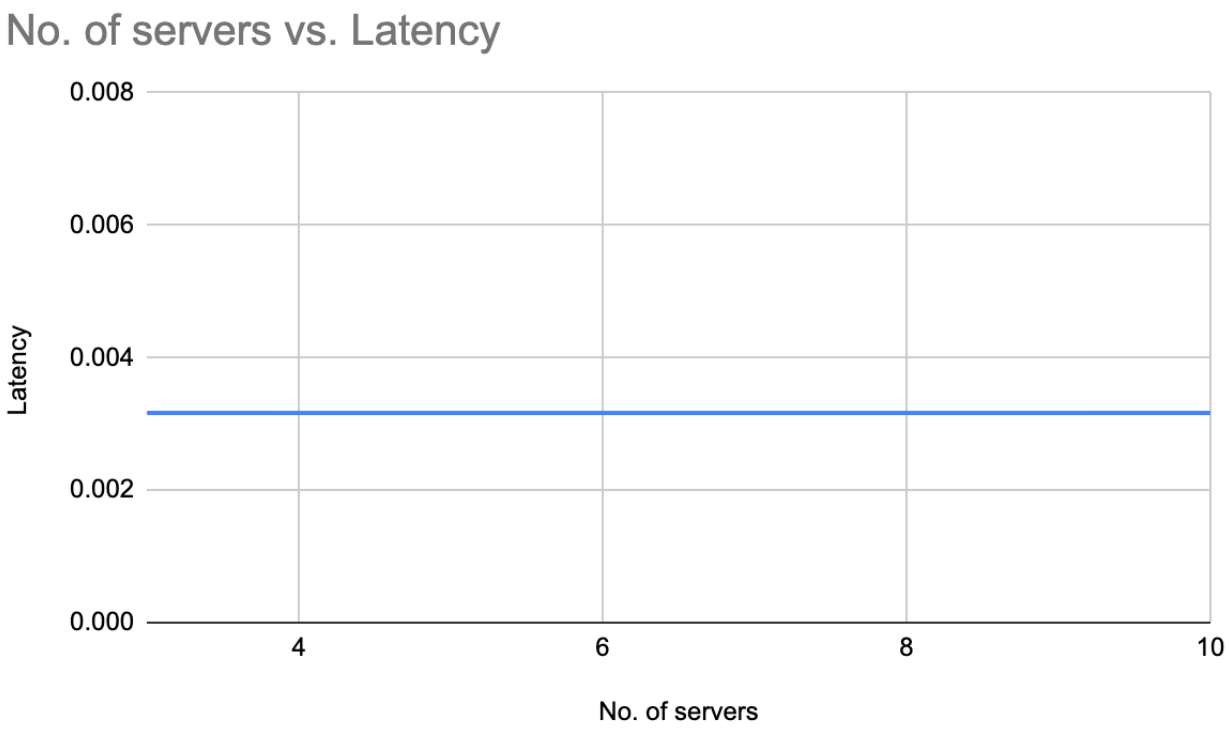
No. of servers	Latency
3	0.002849170923
5	0.003934926748
7	0.005495202303
10	0.007395201302



Inference - Since there is no PeerTable present and we depend on only the successor and predecessor for connections, the latency increases linearly as the number of servers in the peer to peer network grows.

2. Graph for a circular ring with the Peer table with no replication

No. of servers	Latency
3	0.01392449784
5	0.01336341286
7	0.003165023565
10	0.003165023565



Inference - Since the number of Hops will always be 1, the latency remains the same and is not dependent on the number of servers in the system. The size of the PeerTable also increases linearly with this design choice. The aim here was to build a university scale system that can store data and work efficiently in real time.

7. Lessons learned.

Following are the lessons learned:

a. Understanding the architecture of Peer to Peer systems.
b. Understanding of the Distributed hash table and chord in particular.
c. How to set up communication between several servers to form a chord ring.
d. Improving our thinking on edge cases that can happen in a system and thinking on how to handle those cases.
e. Learning the system and its nuances by implementing a toy version of the system.

8. Instructions on how to run the program.

We need to use the main.py script to start the chord process on a server. We can use multiple servers and run them or can run multiple servers on different ports of the same instance as well.

python3 -u main.py --Ip <IP> --Port <PORT> --BitsLength <2^m> --PeerIp <PeerIP> --PeerPort <PEERPORT>

This is the structure and parameters to run the chord server. The peer ip and the peer port are optional.

For the first server, we should not provide the peer ip and peer port because this is the only server that will be responsible for the entire chord Ring. For the next servers, we need to pass any server part of the ring as the peer ip and peer port.

Example commands for running chord with 4 servers in the ring on local:

```
nohup python3 -u main.py --Ip Amiths-MacBook-Pro.local --Port
15010 --BitsLength 12345 >
/Users/amithganesh/khoury/distributed_systems/CS7610-Chord/1.out
&

sleep 2

nohup python3 -u main.py --Ip Amiths-MacBook-Pro.local --Port
15011 --BitsLength 12345 --PeerIp Amiths-MacBook-Pro.local
--PeerPort 15010 >
/Users/amithganesh/khoury/distributed_systems/CS7610-Chord/2.out
&

sleep 2

nohup python3 -u main.py --Ip Amiths-MacBook-Pro.local --Port
15012 --BitsLength 12345 --PeerIp Amiths-MacBook-Pro.local
--PeerPort 15010 >
/Users/amithganesh/khoury/distributed_systems/CS7610-Chord/3.out
&
```

```
sleep 2

nohup python3 -u main.py --Ip Amiths-MacBook-Pro.local --Port
15013 --BitsLength 12345 --PeerIp Amiths-MacBook-Pro.local
--PeerPort 15010 >
/Users/amithganesh/khoury/distributed_systems/CS7610-Chord/4.out
&
```

Here the ip and peer ip are the same and the above example is for running on the local machine.

9. Individual contributions

Setup Virtual Ring - Amith
Setup finding neighbors - Amith
Setup put and get commands - Aayushi
Work on exit command - Amith
Setup peerTable - Amith and Aayushi
Error Handling - Amith and Aayushi
Testing on local and Khoury Systems - Amith and Ayushi
Replication - Amith
Report Preparation - Amith and Ayushi