

Distributed Hash Table

Ashish Agarwal	Brooke Dalton
San Jose State University	San Jose State University
016621709	015486315
<code>ashish.agarwal@sjsu.edu</code>	<code>brooke.dalton@sjsu.edu</code>

Aravind Rokkam
San Jose State University
016592212
`aravind.rokkam@sjsu.edu`

Abstract

In this research project, we implement Chord, a scalable peer-to-peer lookup protocol to provide Key-Value store service using Distributed Hash Tables. A Chord cluster consists of 2^m possible node IDs, where server nodes map their server name (or IP) to a node ID. Both of our server and client applications are implemented with NodeJS and gRPC.

Our experiment and evaluation are done locally and on AWS EC2 with different numbers of servers. To prove our implementation is correct, we use the web application framework ExpressServer to show our key-value store works as expected.

We are looking to solve fast lookup within a distributed environment. Chord is able to solve locating a data item in a collection of distributed.

1. Introduction

A Distributed Hash Table (DHT) is a decentralized distributed system that manages key-value stores by providing a lookup service based on keys. The nodes that are distributed are assumed to retrieve the value associated with the key efficiently and the set of participants causes minimal disruption. Ideally, a DHT can handle a large number of nodes and handle the continuous arrivals and departures of nodes.

This report first talks about the advantages of using a DHT, goals and techniques we implement in our Chord application, implementation details, design decisions, challenges faced, performance, analysis, related work,

how we tested our system, a small demonstration, and future work we would like to implement.

1.1. Advantage of a Distributed Hash Table

The main advantage of a DHT is that nodes can be added or removed with minimal work around re-distributing keys. A DHT is a great design for distributed systems since it can provide hash table-like functions such as put and get to manage large amounts of distributed data. The put function puts data into a node with a key and the get function grabs the data using the key. These two functions work well in a variety of instances and help provide stability and efficiency, especially in large-scale systems [1].

2. Goals and Techniques

This proposal provides specifications needed to create a working distributed hash table. Our distributed hash table implementation contains an agreement protocol so when a node value is updated or a new node is added the finger table has the correct value. A distributed shared memory protocol, so that all nodes are aware of the ids and IPs stored in the finger table even though they are not stored in the same location. And lastly, a resource discovery protocol is also put into consideration since we must locate, retrieve, and advertise server information for all nodes to communicate effectively.

2.1. Agreement Protocol

An agreement protocol is needed when nodes in a distributed system compete or cooperate to achieve a common goal. In many cases, some form of agreement needs to be established for the system to function as expected. Without an agreement protocol, a faulty node might send unexpected results or values to other nodes, resulting in disruption of the entire communication and thus the goal.

Our system uses an agreement protocol to decide which nodes store which keys. The nodes also arrive at an agreement about the finger entries such that the combination of entries includes all the nodes in a cluster. Finally, when nodes join or leave, key entries are copied and deleted from necessary nodes based on hash identifiers.

2.2. Distributed Shared Memory Protocols

Distributed Shared Memory protocols ensure that data is shared among different nodes in a cluster, and no one node stores all data. Memory management is ensured using Remote Procedure Calls (RPCs). Every node allows data read and write to other nodes in the cluster while assuring consistency. This model provides a larger memory space and aids with the agreement of the nodes that store the right replicas.

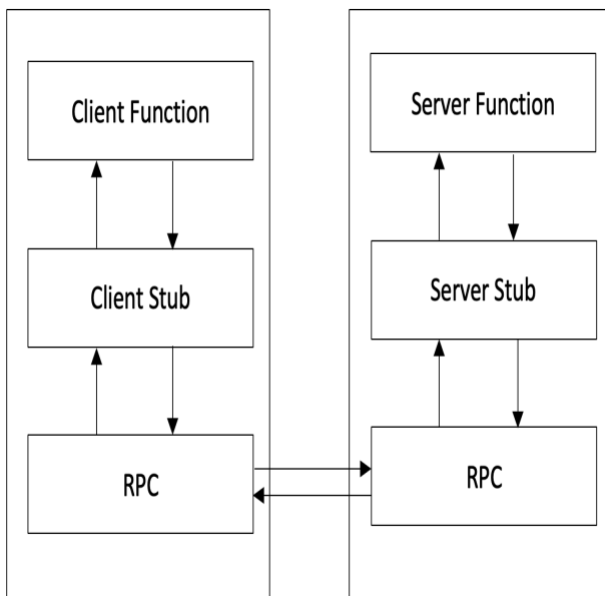


Figure 1: RPC Diagram

The distributed hash table implemented by us stores key-value pairs in different nodes based on the key's hash identifier. A given key is stored at and always retrieved from the immediate successor node of the key. For example, assume a cluster having nodes with hash identifiers 1, 2, and 7. A key with hash identifier 5 will always be stored and retrieved from the node with identifier 7.

2.3. Resource Discovery

Effective communication is the first step for a distributed system to achieve the goals of the given system. In order to do this, nodes must first know about the other existing nodes in the cluster. At the least, nodes must know about the existence of a subset of nodes based on communication agreements. Nodes discover other nodes by querying nodes they already know about. Nodes also discover other nodes when they are notified about the existence of such nodes.

Resource discovery [7] is the most basic step in our chord implementation. Every node maintains a finger table of nodes with m entries, where m is the number of bits allowed in the cluster. Finger tables are updated every few seconds by querying other nodes to get the successor node based on the hash identifier of the finger entry. Nodes are also discovered when a new predecessor notifies of its existence to the immediate successor node.

3. Implementation Details

There are three essential applications within the project. One instantiates a node on a Chord, one instantiates a web server necessary for the user interface, and the other instantiates a PUT and GET request to insert and view values in the Chord cluster.

Figure 2: GET and PUT Request Example

3.1. Chord

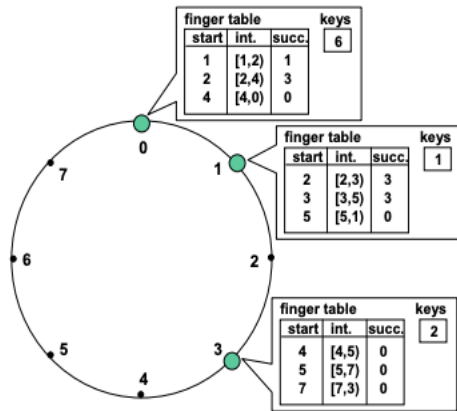


Figure 3.b. from Stoica et al.[2]

The Chord protocol is based on consistent hashing where a key is assigned to the node whose identifier is equal to or first follows the key in the first successor of the key, or the identifier space. Each node stores a finger table of size m , a successor list, and a predecessor [2]. We first call the create function or the join function. When a node is the first node in the Chord cluster, it will call create to establish the Chord cluster.

A node can join the Chord cluster given any known node in the cluster. Our implementation of Chord also has a stabilize function that is called every 6 seconds. The stabilize function is responsible for verifying the node's immediate successor and telling the successor about the node. Our implementation also has a fix fingers function which refreshes finger table entries and is called when a node is joining the Chord cluster.

When a node joins the finger table, the Chord protocol spreads the keys evenly over the nodes which shows load balancing between the nodes. For scalability, the lookup path length and the size of data stored on each node grows as the log of the total number of nodes. A very large scale Chord cluster is feasible without much overhead for key lookup or maintenance. To show availability, the node responsible for a key can be found even if the system is in a state of change.

Following are the brief uses of the most important methods used for our implementation:

- *create()*: called when the first node is created in a cluster.
- *join()*: called when nodes join an existing cluster.
- *findSuccessor()*: used to find the next immediate successor node for a given identifier.

- *closestPrecedingNode()*: used to find the immediate predecessor node for a given identifier.
- *stabilize()*: used to verify whether the successor node for a given node is the actual successor node. If it is not, the actual successor is notified of the given node's existence. Called every 6 seconds asynchronously.
- *notify()*: called by a predecessor node to notify of its existence. The current node will then point to the new predecessor.
- *fixFingers()*: used to fix the finger table entries by finding the current successor for a given identifier. Called every 5 seconds asynchronously.
- *put()*: used to first find the node to which the given key belongs to and then inserts the key-value pair to that node.
- *get()*: used to first find the node to which the given key belongs to and then retrieves the value for the key.
- *updateMap()*: when a predecessor for a node changes, the key-value entries from the new predecessor to the current node are copied to the new predecessor, and deleted from the current node.

A. Creating a Cord Cluster:

The syntax to starting the server without a known node is:

```
npm start -- <host> <port>
```

The syntax to starting the server with a known node is:

```
npm start -- <host> <port> <knownNodeHost> <knownNodePort> <hashId>
```

3.2. ExpressServer

The web server application includes a crawler that navigates through the Chord nodes. In the current implementation, it must be started after at least one node exists or the server will fail. After the first node exists, the web server can be started at any time but the best user experience occurs if it is started after some time so that the fingerTable has time to stabilize.

A. Starting the ExpressServer:

The syntax to starting the ExpressServer is:

```
npm run startExpress <host> <port>
```

3.3. Key-Value Pairs

The Chord protocol supports one operation: given a key, it will determine the node responsible for storing the key's value. Chord does not store keys and values but provides a convenient system that allows us to store key-value pairs in a distributed environment. Our key-value implementation supports PUT and GET requests through the ExpressServer to view, create, and request our key-value pairs.

A. Querying the Finger Tables

The syntax to query the finger table:

```
http://<host>:<port>/fingers?knownHost=<known-host>&knownPort=<known-port>
```

B. Inserting Key-Value Pairs

The syntax to insert a key-value pair:

```
http://localhost:3000/put?knownHost={any-known-host}&knownPort={any-node-with-known-port}&key={key}&val={val}
```

C. Get the Value of a Key

The syntax to get the value of a key:

```
http://localhost:3000/get?knownHost={any-known-host}&knownPort={any-node-with-known-port}&key={key}
```

4. Design Decisions

Our project is implemented using NodeJS as the language, gRPC framework for communication, git for version control, and AWS EC2 for hosting.

4.1. NodeJS

For our project we decided to use NodeJS as the language. NodeJS allows for rapid development and is easily supported by AWS and gRPC. It is also a good choice for building fast and scalable server-side and networking applications and is efficient for real-time applications.

4.2. gRPC

gRPC is a modern open-source high performance Remote Procedure Call (RPC) framework [6]. Our project hosted all of our RPC function calls using the gRPC framework. Using gRPCs allowed us to quickly create and deploy new RPCs since it allowed us to use protocol buffers, or protobufs, to structure our communication. Each

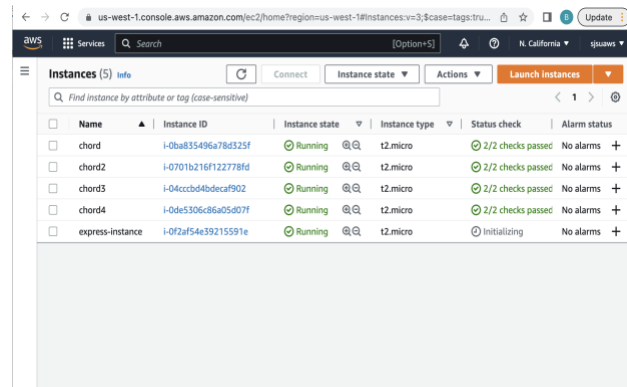
Chord instance played the role of an RPC server and an RPC client.

4.3. Git

To make collaboration and contribution easier for everyone on the team, we used a version control system to keep track of our changes and work. This also helped to distribute the application across the multiple EC2 instances we created. If any change happened locally then it was easy to update our remote nodes to propagate the changes.

4.4. Amazon EC2

To simulate a distributed system, we ran our project in multiple EC2 instances on AWS. The computing resources are powerful and allow us to test the scalability of our system in an environment that is more similar to the real world instead of deploying our project locally on different ports.



Name	Instance ID	Instance state	Instance type	Status check	Alarm status
chord	i-0ba835496a786325f	Running	t2.micro	2/2 checks passed	No alarms
chord2	i-0701b216f122778fd	Running	t2.micro	2/2 checks passed	No alarms
chord3	i-04cccb4bdeca902	Running	t2.micro	2/2 checks passed	No alarms
chord4	i-0de5306c86ad5d07f	Running	t2.micro	2/2 checks passed	No alarms
express-instance	i-0f2af54e39215591e	Initializing	t2.micro		No alarms

Figure 3: EC2 Instances

5. Challenges

Some of the challenges we encountered during this project were with AWS, gRPC, and connection timeouts.

5.1. AWS

After successfully being able to add nodes and values locally, we tried setting up EC2 instances to see if they would be able to communicate on different servers but we ran into an issue when attempting to add the second node. Connecting two EC2 instances with the join method was throwing an error as it couldn't read the id of the joining node. We solved this problem by changing the security group rules. We allowed all ICMPV4 traffic and this resolved the error.

5.2. gRPC

Getting our client to talk to the server using gRPC was simple enough locally, however, we ran into a lot of issues when we tried to deploy this on AWS. After being able to ping the EC2 instance to a local machine we were still seeing "error": "14 UNAVAILABLE: failed to connect to all addresses" errors when attempting to join an existing cluster. To solve this issue, we changed our gRPC authentication from creating an insecure credential to leaving the credential field undefined.

5.3. Connection Timeouts

Another problem we faced during this implementation was with connection timeouts. The servers were not able to establish connections with each other and the ExpressServer due to timeout issues. We solved this challenge by increasing the timeout to a reasonable threshold.

6. Performance

6.1. Fault Tolerance

While Chord is only useful for locating nodes holding a particular key, we would still need to handle the reliability of the network. In practice, nodes are prone to failure.

When a node goes down, every other node will be notified and every node having the failed node as the finger table entry would update its finger table. Successor and predecessor nodes would inform of respective node updates.

6.2. Scalability

Scalability in Chord is achieved by the logarithmic growth of lookup costs with the number of nodes in the network. Essentially, the cost of lookup grows as the log of number of nodes. This allows even very large systems to be possible. Each node stores information about only a small number of nodes (m). Each node knows more about nodes closely following it than about nodes farther away as defined by the findSuccessor function. In general, a finger table does not contain enough information to determine the successor of an arbitrary key.

6.3. Consistency and Concurrency

Consistency can be described as a property of a distributed system that ensures every node or replica has the same data at a given time. For the Chord algorithm and our project, we have implemented a stabilization protocol that

periodically updates the successor node and finger table. Our stabilization protocol uses these six functions:

- *create()*
- *join()*
- *stabilize()*
- *notify()*
- *fixFingers()*
- *closestPrecedingNode()*

The create function is simply used to create a new Chord cluster. The join function asks n if there are any known Chord nodes and to find the immediate successor of a new node to allow n to join the Chord cluster.

Each time a node runs the stabilize function, it asks its successor for its predecessor and decides whether the predecessor should be node's successor instead. The notify function is called when the node thinks it might be the predecessor.

At this point, all predecessor and successor pointers are now correct, but the finger table still needs to be fixed. Now each node periodically calls the fixFingers function to make sure its finger table is correct. This is how new nodes initialize their finger tables and how existing nodes incorporate new nodes into finger tables. closestPrecedingNode is the final function and is called to check whether the predecessor has failed.

The stabilization protocol guarantees that all the finger table entries involved in the lookup are current, and that after some time all successor pointers will be correct after the last join due to the periodic calls on the stabilize and fixFingers functions.

7. Analysis

Chord specifies how keys are assigned to nodes and how nodes can discover the value of keys not available in the local cache by locating the node holding the key. The worst-case time complexity to search for a key using Chord is $O(\log(n))$.

8. Related Work

8.1. Peer to Peer Systems

Peer-to-peer (P2P) content sharing has been an astonishingly successful P2P application on the Internet. P2P has gained tremendous public attention from Napster, the system supporting music sharing on the Web. It is a new emerging, interesting research technology and a promising product base.

Intel P2P working group gave the definition of P2P as "The sharing of computer resources and services by direct

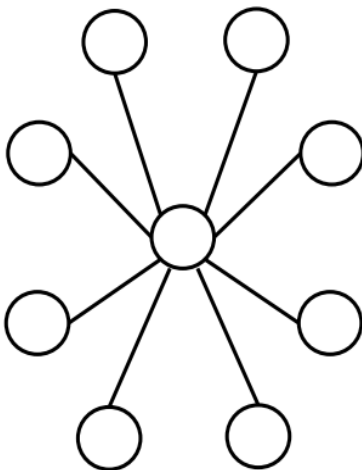
exchange between systems". This thus gives P2P systems two main key characteristics:

- Scalability: there is no algorithmic, or technical limitation of the size of the system, e.g. the complexity of the system should be somewhat constant regardless of the number of nodes in the system.
- Reliability: The malfunction on any given node will not affect the whole system (or maybe even any other nodes).

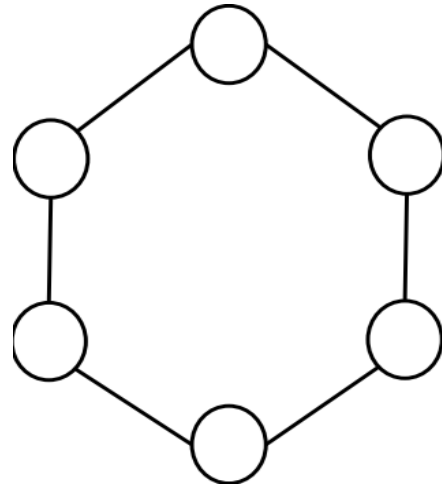
P2P can be categorized into two groups classified by the type of model: pure P2P, and hybrid P2P. Pure P2P models, such as Gnutella, Freenet, do not have a central server. Hybrid P2P models, such as Napster, Groove, Magi, employ a central server to obtain meta-information such as the identity of the peer on which the information is stored or to verify security credentials. In a hybrid model, peers always contact a central server before they directly contact other peers.

According to [3], all peer-to-peer topologies, no matter how different they may be, will have one common feature. All file transfers made between peers are always done directly through a data connection that is made between the peer sharing the file and the peer requesting for it. The control process prior to the file transfer, however, can be implemented in many other ways. As stated by (Nelson, 2001), P2P file sharing networks can be classified into four basic categories which are the centralized, decentralized, hierarchical and ring systems. Although these topologies can exist on their own, it is usually the practice for distributed systems to have a more complex topology by combining several basic systems to create, what is known now as hybrid systems. Four basic systems are:

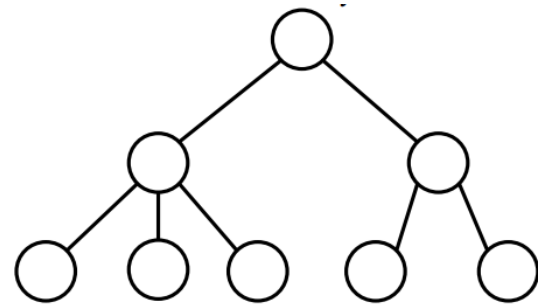
- Centralized Topology



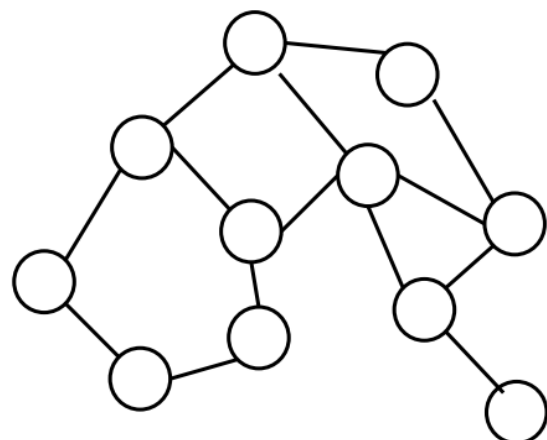
- Ring Topology



- Hierarchical Topology



- Decentralized Topology



Having discussed the basic topologies of peer-to-peer networks, there are more complex real-world systems that generally combine several basic topologies into one system. This is known as a hybrid architecture [4].

8.2. Gnutella

Gnutella is a protocol for distributed search. Although the Gnutella protocol supports a traditional client/centralized server search paradigm, Gnutella's distinction is its peer-to-peer, decentralized model. In this model, every client is a server, and vice versa. These so-called Gnutella servents perform tasks normally associated with both clients and servers. They provide client-side interfaces through which users can issue queries and view search results, while at the same time they also accept queries from other servents, check for matches against their local data set, and respond with applicable results. Due to its distributed nature, a network of servents that implements the Gnutella protocol is highly fault-tolerant, as operation of the network will not be interrupted if a subset of servents goes offline.

The Gnutella protocol defines the way in which servents communicate over the network. It consists of a set of descriptors used for communicating data between servents and a set of rules governing the inter-servent exchange of descriptors. Currently, the following descriptors are defined:

Descriptor	Description
Ping	Used to actively discover hosts on the network. A servent receiving a Ping descriptor is expected to respond with one or more Pong descriptors.
Pong	The response to a Ping. Includes the address of a connected Gnutella servent and information regarding the amount of data it is making available to the network.
Query	The primary mechanism for searching the distributed network. A servent receiving a Query descriptor will respond with a QueryHit if a match is found against its local data set.
QueryHit	The response to a Query. This descriptor provides the recipient with enough information to acquire the data matching the corresponding Query.
Push	A mechanism that allows a firewalled servent to contribute file-based data to the network.

Figure 4: Gnutella descriptors.

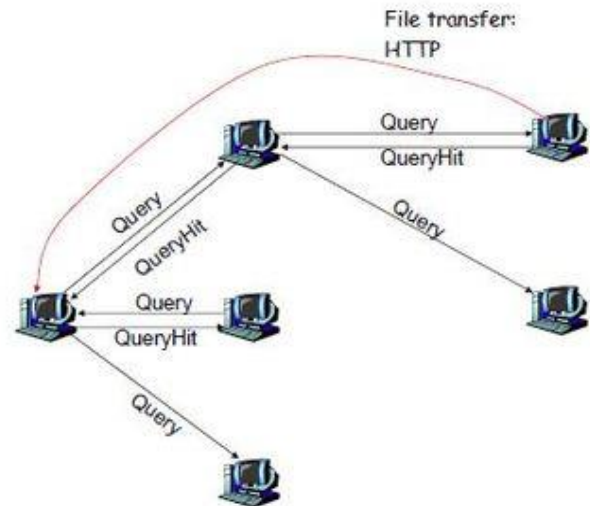


Figure 5: Gnutella Protocol.

8.3. Napster

Napster is a file-sharing P2P application that allows people to search for and share MP3 music files through the vast Internet. It was single handedly written by a teenager named Shawn Fanning (Jeff, 2000). Not only did he develop the application, but he also pioneered the design of a protocol that would allow peer computers to communicate directly with each other. This paved a way for more efficient and complex P2P protocols by other organizations and groups.

The architecture of Napster is based on the Centralized Model of P2P file-sharing. It has a Server-Client structure where there is a central server system which directs traffic between individual registered users. The central servers maintain directories of the shared files stored on the respective PCs of registered users of the network. These directories are updated every time a user logs on or off the Napster server network. Clients connect automatically to an internally designated "metaserver" that acts as a common connection arbiter. This metaserver assigns at random an available, lightly loaded server from one of the clusters. Servers appeared to be clustered about five to a geographical site and Internet feed, and able to handle up to 15,000 users each. The client then registers with the assigned server, providing identity and shared file information for the server's local database. In turn, the client receives information about connected users and available files from the server.

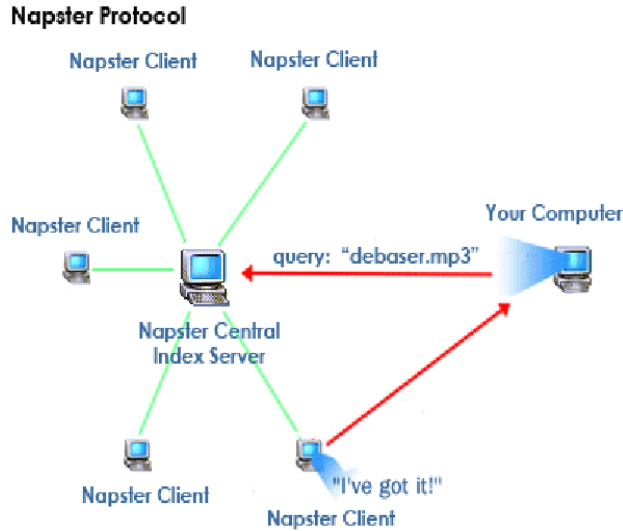


Figure 6: Napster Protocol.

Although formally organized around a user directory design, the Napster implementation is very data centric. The primary directory of users connected to a particular server is only used indirectly, to create file lists of content reported as shared by each node.

The Napster Protocol Due to the fact that Napster is not an open-source application, it was only possible to build up a similar application in revealing the Napster protocol by reverse-engineering.

Napster works with a central server which maintains an index of all the mp3 files of the peers. To get a file you have to send a query to this server which sends you the port and IP address of a client sharing the requested file. With the Napster application it is now possible to establish a direct connection with the host and to download a file.

9. Testing

We developed our protocol locally for easy modification and testing. Final testing was performed on Amazon EC2 using multiple t2.micro instances. We were able to make sure our finger tables were updated correctly with the ExpressServer. For example, if we have one Chord cluster with 3 nodes with IDs 1, 2, and 7 then we would expect our table to look like:

4001 (1):

0 - 2
1 - 7
2 - 7

4002 (2):

0 - 7
1 - 7
2 - 7

4007 (7):

0 - 1
1 - 1
2 - 7

In the first table we have the first node with hash ID 1 starting on port 4001. At finger 0 it points to node 2, at finger 1 it points to node 7, and finger 2 points to node 7 as well. Similarly, we see where the nodes are stored at hash ID 2 and hash ID 7. After running the ExpressServer we see a similar output in our EC2 testing:

← → ↻ Not Secure | 13.56.159.193:3000/fingers?knownHost=54.67.125.220&knownPort=4001

Details for Node with Id 1

Finger Id	Node Id	Host	Port
0	2	3.101.81.216	4002
1	7	54.183.180.138	4007
2	7	54.183.180.138	4007

Details for Node with Id 2

Finger Id	Node Id	Host	Port
0	7	54.183.180.138	4007
1	7	54.183.180.138	4007
2	7	54.183.180.138	4007

Details for Node with Id 7

Finger Id	Node Id	Host	Port
0	1	54.67.125.220	4001
1	1	54.67.125.220	4001
2	7	54.183.180.138	4007

Figure 7: ExpressServer showing the finger table with hash ID's 1, 2 and 7.

10. Demonstration

Below is an example log output after creating a new cluster:

```
Server started at 54.67.125.220:4001
Node 1:54.67.125.220:4001 => A new cluster is being created
Node 1:54.67.125.220:4001 => Fixing finger 0
Node 1:54.67.125.220:4001 => Finding successor for id 2 - Current successor 1
Node 1:54.67.125.220:4001 => Finding successor => No finger or successor found - successor is itself
Node 1:54.67.125.220:4001 => Finger 0 set to 1:54.67.125.220:4001
```

Figure 8: Log output after first node starts Chord cluster.

Another log output screenshot of our second node joining the cluster:

```
Server started at 3.101.81.216:4002
Node 2:3.101.81.216:4002 => Node joining the cluster
Node 2:3.101.81.216:4002 => Found successor 1:54.67.125.220:4001
Node 2:3.101.81.216:4002 => Node's successor and first finger: 1:54.67.125.220:4001
```

Figure 9: Log output after second node joins existing Chord cluster.

Lastly, we show the log output after our third node joins the cluster:

```
Server started at 54.183.180.138:4007
Node 7:54.183.180.138:4007 => Node joining the cluster
Node 7:54.183.180.138:4007 => Found successor 1:54.67.125.220:4001
Node 7:54.183.180.138:4007 => Node's successor and first finger: 1:54.67.125.220:4001
Node 7:54.183.180.138:4007 => Fixing finger 1
Node 7:54.183.180.138:4007 => Finding successor for id 1 - Current successor 1
Node 7:54.183.180.138:4007 => Finding successor => Finding closestPrecedingNode
Node 7:54.183.180.138:4007 => Finding successor => closestPrecedingNode is 7:54.183.180.138:4007
Node 7:54.183.180.138:4007 => Finger 1 set to 1:54.67.125.220:4001
```

Figure 10: Log output after third node joins existing Chord cluster.

Inserting a value into Chord cluster:

```
← → ↻ ⚠ Not Secure | 13.56.159.193:3000/put?knownHost=54.67.125.220&knownPort=4001&key=3&val=num3

Inserted at below node
Node Id  Host  Port
7       54.183.180.138 4007
```

Figure 11: ExpressServer showing a value inserted with PUT request.

Viewing a specific value from known node:

```
← → ↻ ⚠ Not Secure | 13.56.159.193:3000/get?knownHost=54.67.125.220&knownPort=4001&key=3

Returned Value: num3
Retrieved from below node
Node Id  Host  Port
7       54.183.180.138 4007
```

Figure 12: ExpressServer showing a value and node from GET request.

11. Future Work

We have implemented all the essential features of the chord p2p network. It is possible to improve this implementation further by adding more features to the existing framework. Some possible improvements for the future include implementing replication.

11.1. Replication

The correctness of the Chord protocol relies on the fact that each node knows its successor. However, this invariant can be compromised if nodes fail. An incorrect successor will lead to incorrect lookups. To increase robustness, each Chord node maintains a successor list of size r , containing the node's first r successors. If a node's immediate successor does not respond, the node can substitute the second entry in its successor list. All r successors would have to simultaneously fail in order to disrupt the Chord cluster, an event that can be made very improbable with modest values of r . Assuming each node fails independently with probability p , the probability that all r successors fail simultaneously is only p^r . Increasing r makes the system more robust.

To make sure there is no data loss, we can also ensure that every update will be replicated to at least 1 other node. Data replication also ensures high availability as the same data would be available at two different nodes. When a node goes down, all its data will be replicated to another node based on the position of the node in the hash cluster.

12. Conclusion

Many distributed peer-to-peer applications need to determine the node that stores a data item. The Chord protocol solves this challenging problem in a decentralized manner. It offers a powerful primitive; given a key, it determines the node responsible for storing the key's value and does so efficiently. In the steady state, in an n -node network, each node maintains routing information for only $O(\log(n))$ other nodes and resolves all lookups via $O(\log(n))$ messages to other nodes.

Attractive features of Chord include its simplicity, provable correctness, and provable performance even in the face of concurrent node arrivals and departures. It continues to function correctly, albeit at degraded performance, when a node's information is only partially correct. Our theoretical analysis and simulation results confirm that Chord scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and answers most lookups correctly even during recovery.

References

- [1] H. Zhang, Y. Wen, H. Xie, and N. Yu, *Distributed hash table: Theory, platforms and applications*. New York, NY: Springer, 2014.
- [2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord," Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '01, 2001.
- [3] Peter B., Tim W., Bart D. and Piet D. (2002), "A Comparison of Peer-to-Peer Architectures", Broadband Communication Networks Group (IBCN), Department of Information Technology (INTEC), Ghent University, Belgium, 1-2.
- [4] B. Yang, H. Garcia-Moline (February 2002), Designing a Super-Peer Network, Stanford University.
- [5] Uwizeyimana, F. (2014, June 16). *Concurrency control in distributed database systems*. ACM Computing Surveys (CSUR).
- [6] Kasun Indrasiri and Danesh Kuruppu, *gRPC: Up and Running*. O'Reilly Media, 2020.
- [7] M. Harchol-Balter, T. Leighton, and D. Lewin, "Resource discovery in distributed networks," *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing - PODC '99*, 1999, doi: 10.1145/301308.301362.