Distributed Hash Table Final Report

XUZHENG LU

G34363475

ZETIAN ZHENG

G36558368

XIAOYU SHEN

G33150730

GUANGYUAN SHEN

G48268794

1 INTRODUCTION

Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or workloads between peers. (Wikipedia: peer to peer)

In this kind of distributed system, "peer to peer" means that the function of each node is equal, and each node is connected with others via the internet. In this network, files are transported between nodes. There are many features for peer-to-peer systems, for example, anonymity, hierarchical naming, Lookup, and data storage. The most important feature is the efficient location of data items (Stoica et al., 2003).

A distributed hash table (DHT) is a protocol for structured peer-to-peer (p2p) networks used. Each node in the network follows a specific rule to divide and store the large file index hash table. The DHT entries should be the form: (key, value). The key is the hash value of the file, and the value is the IP address where the file is stored. It is efficient to find and return value/address from the query node.

In this distributed hash table, the joined nodes will automatically form a decentralized structure without the scheduling from the master node. In this way, there will be no system crash when the master node is failed or busy. Also, when a large number of nodes are flooded into the system, there may be a short-term load imbalance. The system will randomly allocate the same amount of storage to each node. Since the hash table can have node exit or sudden node failure, we also need to address the issue of exiting the nodes to ensure query and data stability.

2 RELATED WORK

2.1 CAN DHT

Sylvia Paul Ratnasamy et al. proposed the Content-Addressable Network (CAN DHT) (Ratnasamy et al., 2001). CAN was designed as a scalable, fault-tolerant, and self-organizing distributed hash table. It is similar to a multi-dimensional Cartesian coordinate space, and each node in the network can be identified in the space. Each CAN node maintains a routing table that saves each neighbor's IP address and the coordinate area in the virtual space. The node can route the message to the target area in the coordinate space and organize the entire network.

2.2 Chord DHT

Ion Stoica et al. proposed Chord DHT (Stoica et al., 2001). Chord covers the entire network by stringing the nodes' IDs from small to large to form a ring, and each data is stored at the backward node closest to the data key. Chord uses the similar dichotomy search method, a non-linear search algorithm, and the convergence speed can be very fast. The routing complexity can be reduced to the logarithm of the number of nodes.

2.3 Pastry DHT

Antony Rowstron and Peter Drusche proposed Pastry DHT (Rowstron et al. 2001). Like Chord, Pastry also uses a ring network topology, concatenating each node's hash identifiers into a ring. The difference from Chord is that Pastry introduces the idea of grading, and it does not directly use node hash values to construct a ring but only take the first 'a' bit of the node's hash value and the back 'b' bit as the leaf node of a node on the ring. The advantage of this is that it greatly reduces the ring's space size and can reduce the routing time. Another difference is that Pastry's data is stored on the node closest to the key-value, not just in the backward node.

2.4 Tapestry DHT

Ben Y. Zhao et al. proposed Tapestry DHT (Zhao et al., 2004). In the Tapestry network, each node will assign itself a globally unique node ID, and at the same time, there is a unique root node ID in the entire system. All nodes form a tree topology in the network and data is saved on the node closest to the key value of the data.

3 PROJECT DESIGN

3.1 CAN DHT

The main idea of DHT are as follows:

1. Each file index is represented as a (K, V) pair. K is the keyword, which can be the hash value of the file name (or other information of the file), and V is the IP address of the node that stores the file (or other information of the node).
2. All the file index entries (i.e. (K, V) pairs) form a large hash table. We can find the node address that stores a file as long as we input the K value.
3. The large file hash table will be divided into many local small blocks. The local hash tables are distributed to all nodes in the system according to specific rules so that each node is responsible for maintaining one of the blocks.
4. When a node queries a file, it only needs to route the query message to the corresponding node, where the hash table block maintained the (K, V) pair to be searched.

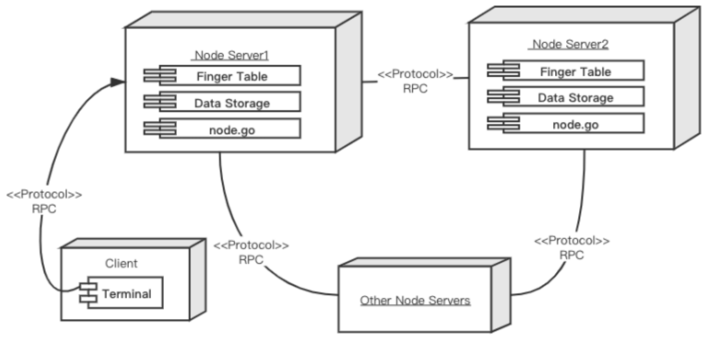


Figure 1 Deployment Diagram

The deployment diagram is shown in this graph. Each node is a node as a container that it holds artifacts: finger table, data storage which is keys and node.go which include the basic function of the node. RPC is a communication protocol between each instance. The user can do some actions which show in use case diagram to control nodes in the chord ring by terminal.

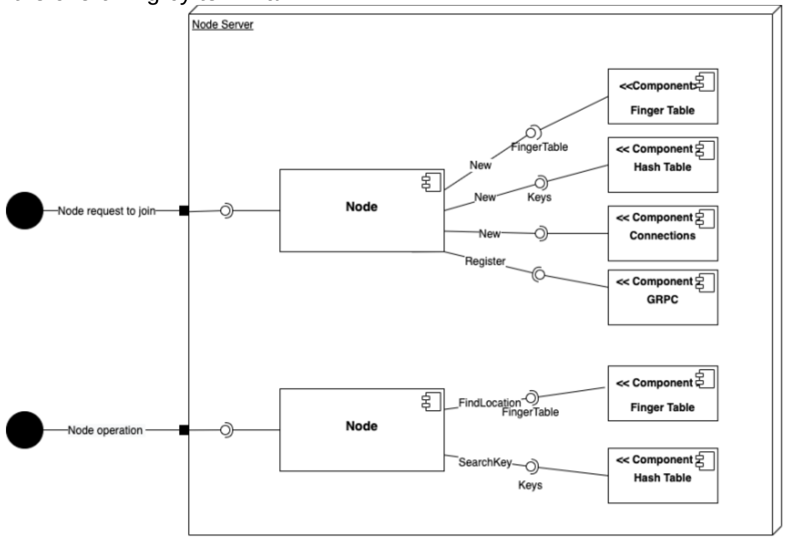


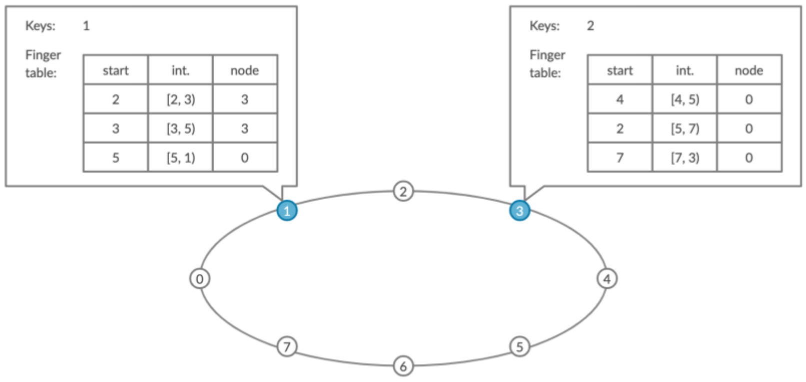
Figure 2 Component Diagram

This is a component diagram. When the node starts and joins the chord, the node sets the preset parameters locally and then enters the node through a port call, then creates a new finger table, a new local hash table, a connection and a GRPC in the node. when the node performs an operation, the node is called through the port and then first finds the precedence and successor based on the finger table node, and then find the corresponding hash value based on the node location.

3.2 Architecture

We will use the architecture similar to Chord DHT proposed by Ion Stoica et al. in 2003 (Stoica et al., 2003). Chord covers the entire network by stringing the nodes' IDs from small to large to form a ring, and each data is stored at the backward node closest to the data key. Chord uses the similar dichotomy search method, a non-linear search algorithm, and the convergence speed can be very fast. The routing complexity can be reduced to the logarithm of the number of nodes.

The structure of a basic Chord-based system is shown as follows:



start	int.	node
2	[2, 3)	3
3	[3, 5)	3
5	[5, 1)	0

start	int.	node
4	[4, 5)	0
2	[5, 7)	0
7	[7, 3)	0

Figure 3 Chord-based System Structure

1. **NID (Node Identifier):** an m-bits number representing a physical machine. m must be large enough to make the probability of sharing the same NID be negligible.
2. **KID (Key Identifier):** an m-bits number representing a key (or resource). Key is hash-bound to a resource so that the KID can be obtained by the Key through the hash operation. m must be large enough to make the probability of sharing the same NID be negligible.
3. **Constant hash function:** Compared with the general hash function, the joining and leaving of nodes have the least impact on the entire system using the constant hash function. SHA-1 is used in Chord for constant hash calculation.
4. **Chord Ring:** NID and KID are allocated on a ring with the size of 2^m, for key distribution, node distribution, and key location. The key is allocated to the node with NID>=KID. This node becomes the successor node of k, the first node in the clockwise direction from k on the ring, denoted as successor(k). The nodes are placed on the ring from large to small in the clockwise direction.

3.3 Core Functions Design

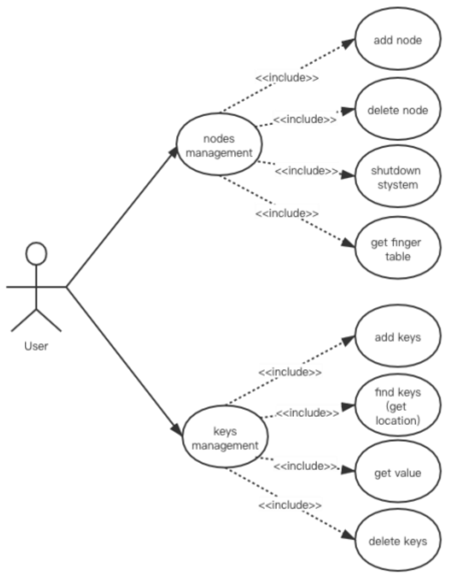


Figure 4 Use Case Diagram

The function design is shown in this graph. In our system, there are three major parts: node management, keys management and node stabilization, the core functions are: find keys (get key location), add new node and node stabilization.

1. Find Keys (Key Location)
    a. The key location is the core function of the Chord system. When node N is looking for a key with KID=id, it will ask whether the key is on the next node. If the KID is between the NID of the node and the NID of the next node, then the key is allocated to the next node. Otherwise, the same query will be asked on the next node, recursively. Finally, the location of the key will be found.
    b. The query time complexity is O(N), where N is the number of nodes in the ring. To accelerate the searching process, Chord uses finger tables to maintain up to m entries on each node, where m is the number of bits in the hash key.
2. Add New Node

When there is a new node added into the system, we must maintain three invariants:

a. The successors of every node should point to their successors correctly.
b. Every key should be correctly stored in the successor(k).
c. The finger tables on every node should be updated correctly.
d. Therefore, there are some tasks that need to be done: the first is Initialize node n with its predecessor and the finger table. And then the new node notifies other nodes to update their predecessors and finger tables. Last, the new node takes over its responsible keys from its successor.

3. Node Stabilization:

   To ensure the reliability of our system, all nodes need to automatically update their status, therefore, this stabilization function is running periodically in the background. In our design there are some functions to maintain node stabilization:

   a. Stabilize(): In the case p recently joined the system, n will ask its successor for its predecessor p and decides whether p should be n's successor instead
   b. Inform(): which notifies node n's successor of its existence, and change its predecessor to n
   c. findNextFinger(): which updates finger tables
   d. checkPreNode(): Periodically checks if its' predecessor is alive

4 DISTRIBUTED SYSTEMS CHALLENGES

There are several challenges in implementing a distributed hash table:

4.1 Fault Tolerance

We have considered fault-tolerant solutions during the project implementation. We create a corresponding channel for each node when it starts, and when the node exits normally or unexpectedly, it will broadcast the message of node exit to other nodes through the channel, and the other nodes will remove the existing node from the finger table in the next data operation. And each node will also modify its own predecessor and successor nodes according to the nodes before and after.

But we didn't consider the nodes starting at the same time when we started them. So, we need to wait for one node to finish starting before we can start the next node when it starts. But in actual operation, a large, distributed hash table cannot wait for other nodes to finish starting.

4.2 Scalability

Our design has a good scalability. When a new node is added, the node is still constantly checking the node before and after, updating the routing table if it is a new node, and keeping it unchanged if it is the original node. Besides, other nodes need to be notified of the addition of new nodes. Last, the new node will take over responsible data from its successor.

4.3 Heterogeneity

In our project, we used Protocol Buffers which is a technology developed by Google as our data serialized method. This technology can help to transform the data structure to a storing or transmission form and the Protocol Buffer can be used in any platform with any language: C++, C#, Dart, Go, Java, and Python. So, in our design, the data storing, and transmission is compatible with different machines (Wiki: Protocol buffer).

4.4 Openness

Our project follows a standard rule to offer services. GRPC is a communication protocol to ensure connections between nodes. GRPC uses Protocol Buffers to encode the data, in this way, this protocol is stricter. Another feature of GRPC is authentication it provides two types of credentials; channel credentials and call credentials, therefore it ensures openness of our project (Wiki: GRPC).

4.5 Security

Our project is based on the DHT system, it has more ability against hostile attacks from outside than a centralized system. Because our data storage is distributed, each node only maintains partial data, therefore when the system is being attacked, it is difficult to get the whole data from the system.

4.6 Concurrency

In our project, all node operations add, delete, and query support concurrent operations, which are processed sequentially according to the order of requests, and we add control locks to each operation to prevent read and write inconsistencies. We also create channels on each node so that also our programs can be executed concurrently and the order in which the messages arrive determines the order of execution.

4.7 Quality of Service

Based on our local testing of the program, we found that all data and operations were delivered accurately, although the project would experience some timeouts. Based on the local quality of service, our project can perform all the functions of the same type of hash table.

4.8 Transparency

Our project has transparency, when any user gets the node code, they can do any operation on the hash table as long as they join the network. And there is no master node in the project, when the first node exits, our project can still run.

5 EXPERIMENTS AND RESULTS

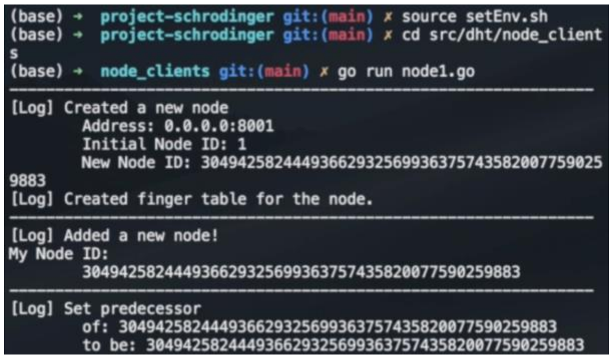
The node instance will be running locally, and to identify different nodes, we assigned a different port number to each node.

5.1 Baseline DHT System

Before testing the functionality, we built a baseline DHT system, which consists of only three nodes, following steps shown below:

5.1.1 Step1: Set the environment for each node

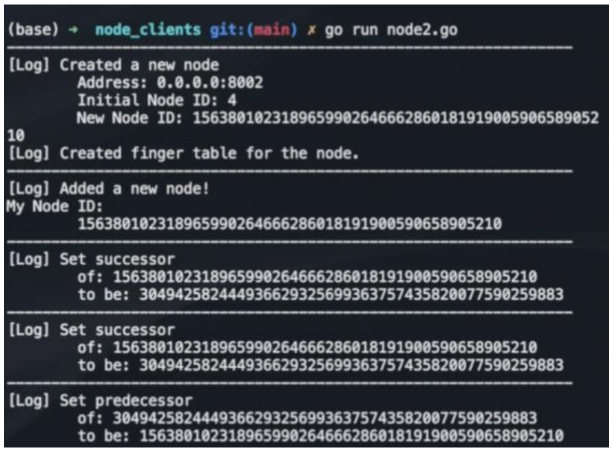
5.1.2 Step2: Initialize the first node



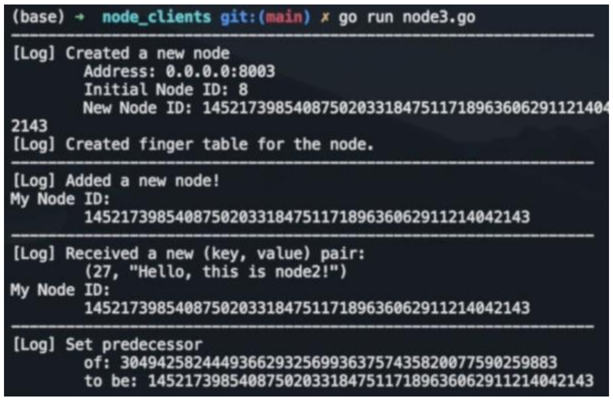
```
(base) → project-schrodinger git:(main) ✗ source setEnv.sh
(base) → project-schrodinger git:(main) ✗ cd src/dht/node_client
s
(base) → node_clients git:(main) ✗ go run node1.go
--------------------------------------------------------------------------
[Log] Created a new node
        Address: 0.0.0.0:8001
        Initial Node ID: 1
        New Node ID: 3049425824449366293256993637574358200775902 5
9883
[Log] Created finger table for the node.
--------------------------------------------------------------------------
[Log] Added a new node!
My Node ID:
        30494258244493662932569936375743582007759025 9883
--------------------------------------------------------------------------
[Log] Set predecessor
        of: 304942582444936629325699363757435820077590259883
        to be: 304942582444936629325699363757435820077590259883
```

As shown above, the node got a new ID, which is the hash of the initial ID that we assigned. And it set itself to be the predecessor.

5.1.3 *Step3: Add new nodes to the node that already in the DHT system*



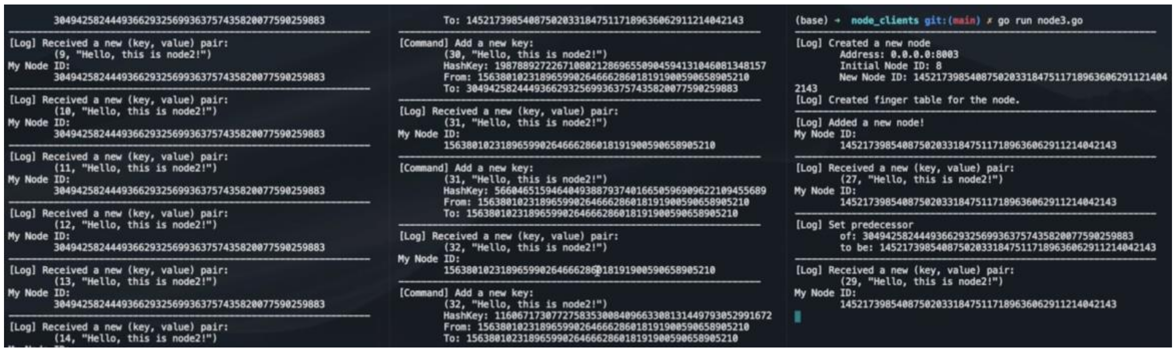
```
(base) → node_clients git:(main) ✗ go run node2.go
-----------------------------------------------------------------
[Log] Created a new node
        Address: 0.0.0.0:8002
        Initial Node ID: 4
        New Node ID: 1563801023189659902646662860181919005906589052
10
[Log] Created finger table for the node.
-----------------------------------------------------------------
[Log] Added a new node!
My Node ID:
        156380102318965990264666286018191900590658905210
-----------------------------------------------------------------
[Log] Set successor
        of: 156380102318965990264666286018191900590658905210
        to be: 304942582444936629325699363757435820077590259883
-----------------------------------------------------------------
[Log] Set successor
        of: 156380102318965990264666286018191900590658905210
        to be: 304942582444936629325699363757435820077590259883
-----------------------------------------------------------------
[Log] Set predecessor
        of: 304942582444936629325699363757435820077590259883
        to be: 156380102318965990264666286018191900590658905210
```



```
(base) → node_clients git:(main) ✗ go run node3.go
-----------------------------------------------------------------
[Log] Created a new node
        Address: 0.0.0.0:8003
        Initial Node ID: 8
        New Node ID: 145217398540875020331847511718963606291121404
2143
[Log] Created finger table for the node.
-----------------------------------------------------------------
[Log] Added a new node!
My Node ID:
        1452173985408750203318475117189636062911214042143
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (27, "Hello, this is node2!")
My Node ID:
        1452173985408750203318475117189636062911214042143
-----------------------------------------------------------------
[Log] Set predecessor
        of: 304942582444936629325699363757435820077590259883
        to be: 1452173985408750203318475117189636062911214042143
```

As shown above, when node 2 and 3 are initialized, the connections were updated on all the nodes.

5.1.4 *Step4: Add keys to the system*



```
        304942582444936629325699363757435820077590259883
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (9, "Hello, this is node2!")
My Node ID:
        304942582444936629325699363757435820077590259883
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (10, "Hello, this is node2!")
My Node ID:
        304942582444936629325699363757435820077590259883
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (11, "Hello, this is node2!")
My Node ID:
        304942582444936629325699363757435820077590259883
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (12, "Hello, this is node2!")
My Node ID:
        304942582444936629325699363757435820077590259883
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (13, "Hello, this is node2!")
My Node ID:
        304942582444936629325699363757435820077590259883
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (14, "Hello, this is node2!")
```

```
        To: 1452173985408750203318475117189636062911214042143
-----------------------------------------------------------------
[Command] Add a new key:
        (30, "Hello, this is node2!")
        HashKey: 198788927226710802128696550904594131046081348157
        From: 156380102318965990264666286018191900590658905210
        To: 304942582444936629325699363757435820077590259883
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (31, "Hello, this is node2!")
My Node ID:
        156380102318965990264666286018191900590658905210
-----------------------------------------------------------------
[Command] Add a new key:
        (31, "Hello, this is node2!")
        HashKey: 566046515946404938879374016650596909622109455689
        From: 156380102318965990264666286018191900590658905210
        To: 156380102318965990264666286018191900590658905210
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (32, "Hello, this is node2!")
My Node ID:
        156380102318965990264666286018191900590658905210
-----------------------------------------------------------------
[Command] Add a new key:
        (32, "Hello, this is node2!")
        HashKey: 1160671730772758353008409663308131449793052991672
        From: 156380102318965990264666286018191900590658905210
        To: 156380102318965990264666286018191900590658905210
```

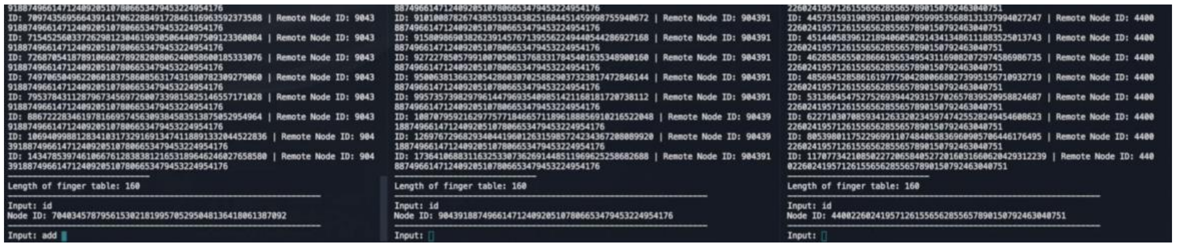
```
(base) → node_clients git:(main) ✗ go run node3.go
-----------------------------------------------------------------
[Log] Created a new node
        Address: 0.0.0.0:8003
        Initial Node ID: 8
        New Node ID: 145217398540875020331847511718963606291121404
2143
[Log] Created finger table for the node.
-----------------------------------------------------------------
[Log] Added a new node!
My Node ID:
        1452173985408750203318475117189636062911214042143
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (27, "Hello, this is node2!")
My Node ID:
        1452173985408750203318475117189636062911214042143
-----------------------------------------------------------------
[Log] Set predecessor
        of: 304942582444936629325699363757435820077590259883
        to be: 1452173985408750203318475117189636062911214042143
-----------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (29, "Hello, this is node2!")
My Node ID:
        1452173985408750203318475117189636062911214042143
```

We let the second node automatically add a key-value pair to the system every second. And we can find out which node finally received the key by checking the received log. As shown above, all of the three nodes received some keys.

5.2 Functionality Testing

We implemented three interactive nodes, in which we can execute some commands manually, such as add the key, get the value of the key, find the location of a key, delete a key, or print out the finger table.

5.2.1 Print out Finger Table and Node ID



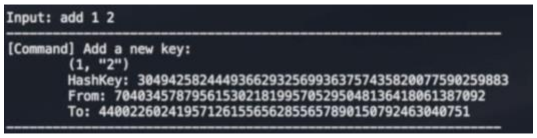
```
918874966147124092051078066534794532224954176
ID: 7097435695664391417062288491728461169635923735 88 | Remote Node ID: 9043
918874966147124092051078066534794532224954176
ID: 715452560337262981230461993050644097509123360084 | Remote Node ID: 9043
918874966147124092051078066534794532224954176
ID: 726870541878910660278928280806240058600185333076 | Remote Node ID: 9043
918874966147124092051078066534794532224954176
ID: 749706504962206018375860856317431980782309279060 | Remote Node ID: 9043
918874966147124092051078066534794532224954176
ID: 795378431128796734569726007339815825146557171028 | Remote Node ID: 9043
918874966147124092051078066534794532224954176
ID: 886722283461978166957456309384583513875052954964 | Remote Node ID: 9043
918874966147124092051078066534794532224954176
ID: 1069409988128341031732916913474118891332044522836 | Remote Node ID: 904
3918874966147124092051078066534794532224954176
ID: 1434785397461066761283838121653189646246027658580 | Remote Node ID: 904
3918874966147124092051078066534794532224954176
--------------------------------
Length of finger table: 160
----------------------------------------------------------------------
Input: id
Node ID: 704034578795615302181995705295048136418061387092
----------------------------------------------------------------------
Input: add
```

```
887496614712409205107806653479453224954176
ID: 910100878267438551933438251684451459998755940672 | Remote Node ID: 904391
887496614712409205107806653479453224954176
ID: 915809869038262391457671395562249440544286927168 | Remote Node ID: 904391
887496614712409205107806653479453224954176
ID: 927227850579910070506137683317845401635348900160 | Remote Node ID: 904391
887496614712409205107806653479453224954176
ID: 950063813663205428603070258829037323817472846144 | Remote Node ID: 904391
887496614712409205107806653479453224954176
ID: 995735739829796144796935409851421168181720738112 | Remote Node ID: 904391
887496614712409205107806653479453224954176
ID: 1087079592162977577184665711896188856910216522048 | Remote Node ID: 90439
1887496614712409205107806653479453224954176
ID: 1269767296829340441960126315985724234367208089920 | Remote Node ID: 90439
1887496614712409205107806653479453224954176
ID: 173641068831163253307362691448511969625258682688 | Remote Node ID: 904391
887496614712409205107806653479453224954176
--------------------------------
Length of finger table: 160
----------------------------------------------------------------------
Input: id
Node ID: 904391887496614712409205107806653479453224954176
----------------------------------------------------------------------
Input:
```

```
2260241957126155656285565789015079246304075 1
ID: 445731593190395101080795999535688131337994027247 | Remote Node ID: 4400
226024195712615565628556578901507924630407 51
ID: 451440583961218940605029143413486111883525013743 | Remote Node ID: 4400
226024195712615565628556578901507924630407 51
ID: 462858565502866619653495431169082072974586986735 | Remote Node ID: 4400
226024195712615565628556578901507924630407 51
ID: 485694528586161977750428006680273995156710932719 | Remote Node ID: 4400
226024195712615565628556578901507924630407 51
ID: 531366454752752693944293157702657839520958824687 | Remote Node ID: 4400
226024195712615565628556578901507924630407 51
ID: 622710307085934126332023459747425528249454608623 | Remote Node ID: 4400
226024195712615565628556578901507924630407 51
ID: 805398011752296991107484063836960905706446176495 | Remote Node ID: 4400
226024195712615565628556578901507924630407 51
ID: 1170773421085022720658405272016031660620429312239 | Remote Node ID: 440
0226024195712615565628556578901507924630407 51
--------------------------------
Length of finger table: 160
----------------------------------------------------------------------
Input: id
Node ID: 440022602419571261556562855657890150792463040751
----------------------------------------------------------------------
Input:
```

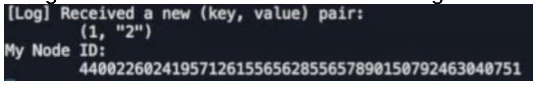
We initialized three interactive nodes, and then printed out the finger table and the Node ID on each node. As shown above, the finger tables are updated correctly.

5.2.2 Add keys



```
Input: add 1 2
----------------------------------------------------------------------
[Command] Add a new key:
        (1, "2")
        HashKey: 304942582444936629325699363757435820077590259883
        From: 704034578795615302181995705295048136418061387092
        To: 440022602419571261556562855657890150792463040751
----------------------------------------------------------------------
```

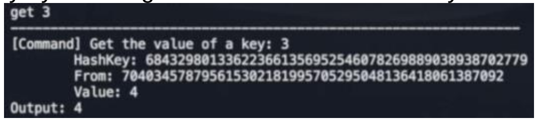
After a key is added, it will be assigned and stored in a node on the ring.



```
[Log] Received a new (key, value) pair:
        (1, "2")
My Node ID:
        440022602419571261556562855657890150792463040751
```

5.2.3 Get the value of a given key

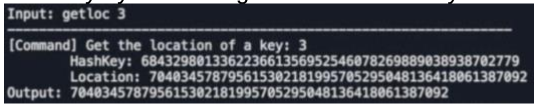
We can get the value of the key by checking the node that stored the key.



```
get 3
----------------------------------------------------------------------
[Command] Get the value of a key: 3
        HashKey: 684329801336223661356952546078269889038938702779
        From: 704034578795615302181995705295048136418061387092
        Value: 4
Output: 4
```

5.2.4 Get the location of a given key

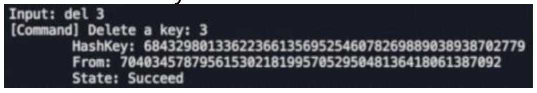
We can also get the location of the key by calculating the hash of the key.



```
Input: getloc 3
----------------------------------------------------------------------
[Command] Get the location of a key: 3
        HashKey: 684329801336223661356952546078269889038938702779
        Location: 704034578795615302181995705295048136418061387092
Output: 704034578795615302181995705295048136418061387092
```

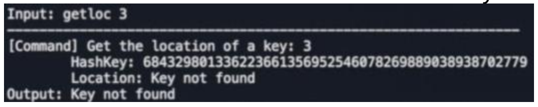
5.2.5 Delete a key

The node that stored the key will delete the key.



```
Input: del 3
[Command] Delete a key: 3
        HashKey: 684329801336223661356952546078269889038938702779
        From: 704034578795615302181995705295048136418061387092
        State: Succeed
```

After the key was deleted, we could not find the value nor location of the key.

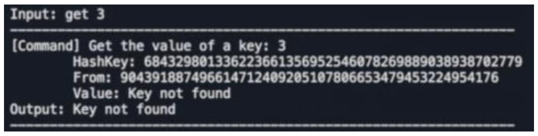


```
Input: getloc 3
----------------------------------------------------------------------
[Command] Get the location of a key: 3
        HashKey: 684329801336223661356952546078269889038938702779
        Location: Key not found
Output: Key not found
```

5.3 Fault Tolerance Testing

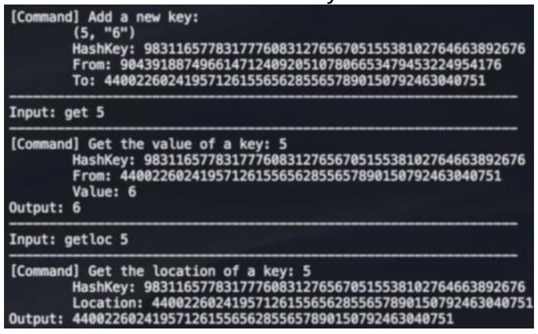
To test the fault tolerance of the system, we could simply shut down a node and to test whether the system still works and reconnect the node again and test the performance.

5.3.1 Step1: Shut down a node



```
Input: get 3
--------------------------------------------------------------------------
[Command] Get the value of a key: 3
        HashKey: 68432980133622366135695254607826988903893870277 9
        From: 904391887496614712409205107806653479453224954176
        Value: Key not found
Output: Key not found
--------------------------------------------------------------------------
```

After a node is shut down, we cannot find the value of the key that is stored on the node.

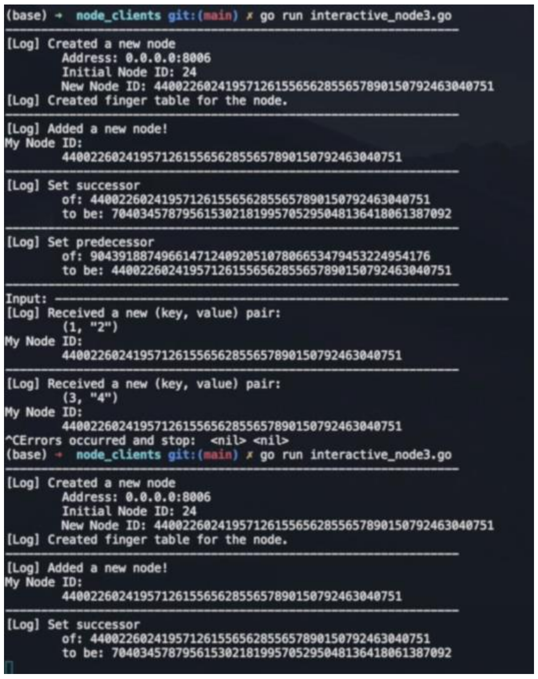


```
[Command] Add a new key:
        (5, "6")
        HashKey: 983116577831777608312765670515538102764663892676
        From: 904391887496614712409205107806653479453224954176
        To: 440022602419571261556562855657890150792463040751
--------------------------------------------------------------------------
Input: get 5
--------------------------------------------------------------------------
[Command] Get the value of a key: 5
        HashKey: 983116577831777608312765670515538102764663892676
        From: 440022602419571261556562855657890150792463040751
        Value: 6
Output: 6
--------------------------------------------------------------------------
Input: getloc 5
--------------------------------------------------------------------------
[Command] Get the location of a key: 5
        HashKey: 983116577831777608312765670515538102764663892676
        Location: 440022602419571261556562855657890150792463040751
Output: 440022602419571261556562855657890150792463040751
```

Nonetheless, the rest part of the system could still work as normal. As shown above, we can add and find a key on the node that is still working.

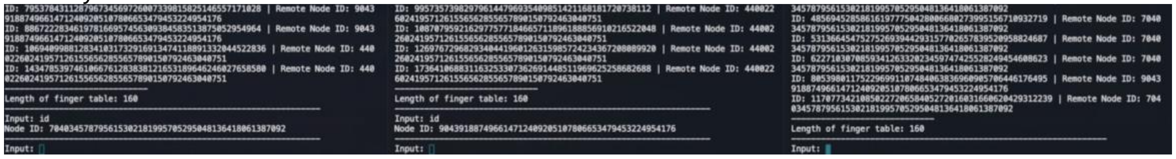
5.3.2 Step2: Reconnect a node

Then we tried to reconnect the node to the system. As shown below, after being added, the node could work as normal.



```
(base) →  node_clients git:(main) ✗ go run interactive_node3.go
--------------------------------------------------------------------------
[Log] Created a new node
        Address: 0.0.0.0:8006
        Initial Node ID: 24
        New Node ID: 440022602419571261556562855657890150792463040751
[Log] Created finger table for the node.
--------------------------------------------------------------------------
[Log] Added a new node!
My Node ID:
        440022602419571261556562855657890150792463040751
--------------------------------------------------------------------------
[Log] Set successor
        of: 440022602419571261556562855657890150792463040751
        to be: 704034578795615302181995705295048136418061387092
--------------------------------------------------------------------------
[Log] Set predecessor
        of: 904391887496614712409205107806653479453224954176
        to be: 440022602419571261556562855657890150792463040751
--------------------------------------------------------------------------
Input: --------------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (1, "2")
My Node ID:
        440022602419571261556562855657890150792463040751
--------------------------------------------------------------------------
[Log] Received a new (key, value) pair:
        (3, "4")
My Node ID:
        440022602419571261556562855657890150792463040751
^CErrors occurred and stop:  <nil> <nil>
(base) →  node_clients git:(main) ✗ go run interactive_node3.go
--------------------------------------------------------------------------
[Log] Created a new node
        Address: 0.0.0.0:8006
        Initial Node ID: 24
        New Node ID: 440022602419571261556562855657890150792463040751
[Log] Created finger table for the node.
--------------------------------------------------------------------------
[Log] Added a new node!
My Node ID:
        440022602419571261556562855657890150792463040751
--------------------------------------------------------------------------
[Log] Set successor
        of: 440022602419571261556562855657890150792463040751
        to be: 704034578795615302181995705295048136418061387092
```

The finger tables and Node IDs are shown as follows, and we could find out that the nodes were reconnected correctly.



```
ID: 79537843112879673456972600733981582514655717 1028 | Remote Node ID: 9043
918874966147124092051078066534794532249541 76
ID: 88672228346197816695745630938458351387505 2954964 | Remote Node ID: 9043
918874966147124092051078066534794532249541 76
ID: 10694099881283410317329169134741188913320 44522836 | Remote Node ID: 440
022602419571261556562855657890150792463040751
ID: 14347853974610667612838381216531896462460 27658580 | Remote Node ID: 440
022602419571261556562855657890150792463040751
--------------------------------
Length of finger table: 160
------------------------------------------------------------------------
Input: id
Node ID: 704034578795615302181995705295048136418061387092
------------------------------------------------------------------------
Input:
```

```
ID: 995735739829796144796935409851421168181720738112 | Remote Node ID: 440022
602419571261556562855657890150792463040751
ID: 1087079592162977577184665711896188856910216522048 | Remote Node ID: 44002
2602419571261556562855657890150792463040751
ID: 1269767296829340441960126315985724234367208089920 | Remote Node ID: 44002
2602419571261556562855657890150792463040751
ID: 173641068831163253307362691448511969625258682688 | Remote Node ID: 440022
602419571261556562855657890150792463040751
--------------------------------
Length of finger table: 160
------------------------------------------------------------------------
Input: id
Node ID: 904391887496614712409205107806653479453224954176
------------------------------------------------------------------------
Input:
```

```
345787956153021819957052950481364180613870 92
ID: 485694528586161977750428006680273995156710932719 | Remote Node ID: 7040
345787956153021819957052950481364180613870 92
ID: 531366454752752693944293157702657839520958824687 | Remote Node ID: 7040
345787956153021819957052950481364180613870 92
ID: 622710307085934126332023459747425528249454608623 | Remote Node ID: 7040
345787956153021819957052950481364180613870 92
ID: 805398011752296991107484063836960905706446176495 | Remote Node ID: 9043
918874966147124092051078066534794532249541 76
ID: 1170773421085022720658405272016031660620429312239 | Remote Node ID: 704
0345787956153021819957052950481364180613870 92
--------------------------------
Length of finger table: 160
------------------------------------------------------------------------
Input:
```

6 PERFORMANCE AND EVALUATION

6.1 Performance

6.1.1 Node fault tolerance

Since our program is based on the local port, we take the performance of the local node fault tolerance as the result. According to our tests, when we start 100 nodes in macOS and shut down 10% randomly, our nodes still work fine.

6.1.2 Delay

Network latency is an important indicator of network performance. Usually, the low latency is 1~30ms, medium latency between 30~50ms, high latency more than 50ms. We built a case with a 10-node distributed hash table, our average latency is 43ms.

6.2 Discussion

6.2.1 Advantages

a. Single point of failure

The design of our DHT firstly solved the Single point of failure for the centralized system like Napster. In our design, we don't have a central node to occur SPOF problem. And we use GRPC and channels to ensure the connections between nodes.

b. Broadcast Storm

The Broadcast Storm is the accumulation of broadcast and multicast traffic on a structured p2p system. Our DHT system avoids this problem by efficiently searching and storing keys in the chord ring.

c. Files store

Compared to the traditional system, our product only maintains partial files on each node, and these files are well balanced in the whole system.

6.2.2 Disadvantages

a. Lost data

Even though we realized that the system has fault tolerance. However, there is still an issue that some data may be lost. Because we did not implement the data backup function. So in future work, we will implement two functions: log function and snapshotting to backup and restore data. It should be like AOP and snapshotting functions in Redis.

b. Robustness of the system

There are still some minor problems with the stability of our system. For example, the node's waiting time exceeds the maximum idle time, which causes the node to reconnect so that certain operations are not performed. However, these situations are not common, and we have been debugging to try to solve these problems.

6.3 Future Work

In the future, we can bring the initialization fault tolerance to the system so that we don’t need to wait for nodes to initialize. Also, the RPC deadline exceeded errors should be solved. Finally, our testing and evaluation is performed locally, and in the future, we can test the system on clusters.

REFERENCES

[1] Stoica, Ion, et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications." IEEE/ACM Transactions on networking 11.1 (2003): 17-32.

[2] Ratnasamy, Sylvia, et al. "A scalable content-addressable network." Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. 2001.

[3] Zhao, Ben Y., et al. "Tapestry: A resilient global-scale overlay for service deployment." IEEE Journal on selected areas in communications 22.1 (2004): 41-53.

[4] Rowstron, Antony, and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems."IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer, Berlin, Heidelberg, 2001.

[5] Medrano-Chávez, M. (2015). A performance comparison of Chord and Kademlia DHTs in high churn scenarios. Peer-to-Peer Networking and Applications, 8(5), 807–821. https://doi.org/10.1007/s12083-014-0294-y

[6] Wiki: GRPC. https://www.wikiwand.com/en/GRPC

[7] Wiki: Protocol Buffers. https://www.wikiwand.com/en/Protocol_Buffers