CS-451 DISTRIBUTED SYSTEMS PROF. UMESH BELLUR

DISTRIBUTED KEY-VALUE STORE

VINAY YOGENDRA 140050057,
VISHWESH CHANDRAMOHAN 140050031,
AGRIM GUPTA 140020003

Github Repository of the Project: https://github.com/vishwesh96/Distributed-key-value-Store

0. Framework of the Code Written

Basic Framework of code revolves around our implementation of chord, and RPC calls which enable communication between nodes. The functions designed are demarcated as :-

a. **Chord & Data Replication Functions,** local functions for implementation of basic chord features like Join, Leave, Stabilise, etc and for ensuring data replication
b. **RPC Stubs,** RPC compatible stubs for the relevant functions which need to be called over network
c. **Remote Chord Functions**, network functions which have same signature but an extra 'address' parameter. The functions connect to appropriate node's RPC interface registered, and calls the connected interface's functions remotely
d. **Client Functions,** client functions like Read, Write and Delete.

1. KEYSPACE ALLOCATION AND NODE TOPOLOGY

1.0 DESIGN

For keyspace allocation, we implement the Chord protocol. The node which is the successor of a key will have the value corresponding to the key. This will also be the leader of the nodes which have this data replicated. The replicas are on the next two successor nodes in the chord ring.

1.0.1 Joining of Nodes

When a new node wants to enter the system, it will call join function on an existing node which it knows. From that, it will know its successor node. It will then inform its successor of its presence which will then send it the key-values corresponding to it and make it leader for that key space and also informs its successors so as to maintain the replicas. As a result, it can start taking requests for its part of the keyspace immediately.

1.0.2 Leaving of Nodes

A node going out will inform its successor that it is leaving and exits. It sends the relevant key value pairs to its successors to ensure replicas are maintained. The successor will take over the part of the failing keyspace and will become leader for that. A similar strategy is employed when a node fails. But failed data is obtained from predecessor/successor nodes. It is explained in greater detail in Section 2.2.

1.1 CHORD IMPLEMENTATION

We have used a ring topology for nodes and have tried to implement various functionalities of chord to suit to our needs.

1.1.1 Overview of Chord Implementation

We have a structure Node which represents any Node in the topology. It has two attributes, address and its ID(hash of address).
Another struct LocalNode inherits Node and is the data structure representing the Node which is executing this. LocalNode has attributes relevant to the local process like its predecessor, successor list, key value data, timer etc. Also, LocalNode has a Config which represents the configuration of chord. It should be the same across all nodes.

We use a default configuration which uses the following.
Hash Function : SHA1
Stabilize Time: Varies from 1 second to 2 seconds
HeartBeat Timeout: 500 milliseconds
Number of Successors maintained: 3
Number of replicas: 2 (other than original)

Finger tables are not yet implemented. Hence, finding successor for a key/node is done using O(n) algorithm by going through successors of Node on which request is made.

1.1.2 Create

To start the key value store, we need to create a LocalNode, call Init on it which initializes its data structures and rpc server and then call Create function on it.
This initializes the successor and predecessor of the node and starts stabilize timer.

1.1.3 Join

When a LocalNode wants to connect to a key value store, this is called with address of an existing Node. This finds the successor of the LocalNode by making an RPC call on the known Node and updates the successor information.

After that, the data needs to be redistributed and replicas have to be correctly placed. This is done by calling StabilizeReplicasJoin function remotely on successor.

1.1.4 Leave

When a LocalNode, wishes to leave, it stabilizes replicas using StabilizeReplicasLeave function and then turns off the stabilization timer. It also informs its predecessor to skip a successor using RPC.

1.1.5 Notify

This procedure is used to notify a Node that the requesting Node may be its predecessor. This function checks if the hash of informer actually lies between existing predecessor and self hash. Accordingly it updates its predecessor.

1.1.6 Ping

This is used to check immediately if a node is alive. Used to check predecessor.

1.1.7 Schedule

This schedules a Stabilize() call after random time between min and max from default configuration. This is called after Create() or Join(). After every stabilize call, this is again called.

1.1.8 Stabilize

This function mainly stabilizes the predecessor, successor pointers and data when the topology changes.
It uses heartbeats to check if any successor failed. Updates successor pointer and replicated data accordingly.
Then, it checks for any node that has joined between current node and successor i.e checks for a new successor and then notifies the successor that this is its predecessor.
In case of changes successors list is updated and liveness of predecessor is also checked.

1.2 Supporting RPC Functions to chord

We have many RPC supported functions which are described briefly below:

FindSuccessor : Finds successor of a key given as input by successively checking through successors i.e using O(n) strategy instead of using finger tables

GetPredecessor : Get predecessor of a node

SkipSuccessor: Skips a successor and modifies successors list as successor is about to leave.

StabilizeReplicasJoin: This stabilizes replicas on surrounding nodes when a node joins. This calls SendReplicasSuccessorJoin to remotely add data to a successor.

SendReplicasSuccessorJoin/Leave: These are helper functions to manipulate replicated data when a node joins or leaves

Heartbeat: This is the function which responds to heart beats.

Read/Write/Delete Key: These functions are called by Node to read/write/delete key values onto nodes.

GetRemoteData: Gets data of requested replica on a node

1.3 Data Representation and Replication Strategy

A node will store the data corresponding to Keyspaces of the node itself, the node's predecessor and the node's predecessor's predecessor. This is done via **three maps**, data[0], data[1] and data[2]. So, basically whatever data is stored in a node, is replicated in the 2 successors of the node, and the node acts as **leader of the three replicas** of the information of the node stored in the system. We take care that the replicas remain consistent, i.e. when a node leaves/enters or a client writes/deletes a node, the replication property remains invariant.

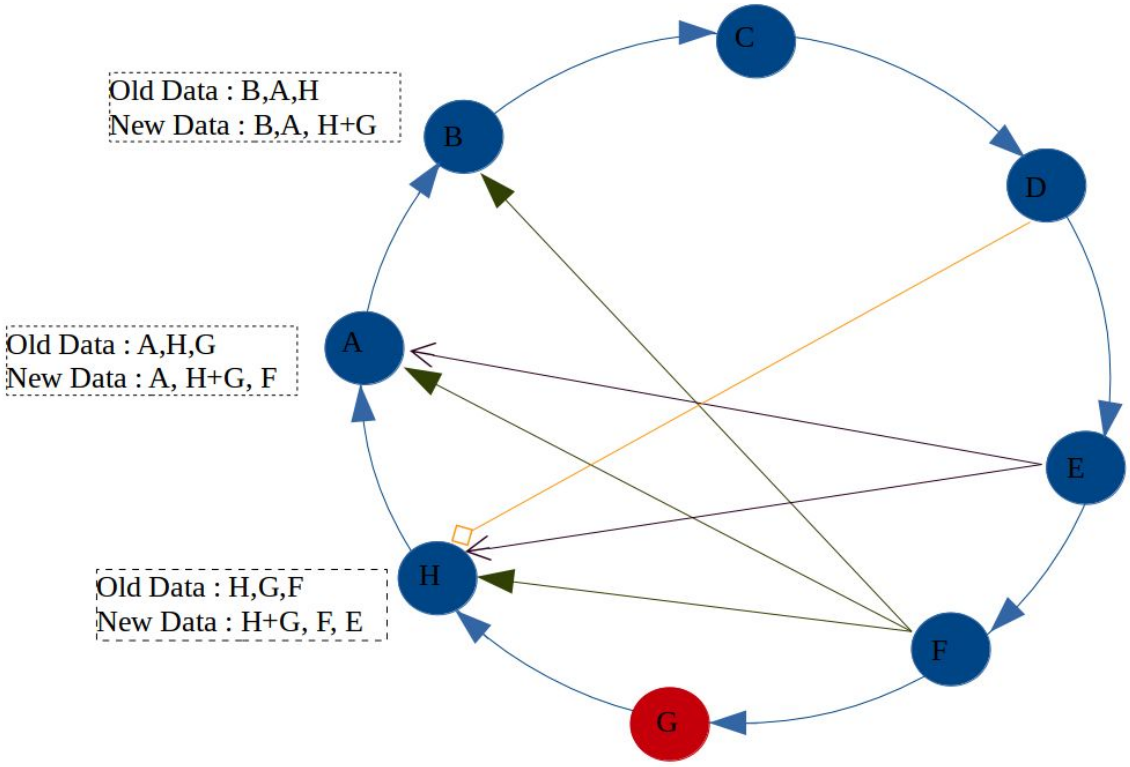
2. FAILURE DETECTION

The Leader of the node calls stabilize function periodically, every 1 to 2 s. The function stabilize also incorporates failure detection through heartbeat channel.

2.1 HEARTBEAT IMPLEMENTATION

The leader pings the successors periodically, waits for 500ms for them to respond. If it doesn't hear a reply in the time window, it assumes that the node is dead and initiates the recovery procedure.

2.2 Recovery Procedure



Here, Node G has failed unexpectedly, without informing others of the same. Node F,D and E will detect the failure through heartbeat channel. The Node F (i.e. the predecessor of the failed node) will initiate the data altering process in successors of the failed node as shown in the figure. The rest 2 nodes (D and E) will only update their successor tables appropriately. In the data altercation process, the data stored in E is sent to H by F, and the node H adds the two keyspaces of itself and that of node G (the failed node), and updates the predecessor information to F and E appropriately, so that data of node G is not lost, whilst maintaining the predecessor and successor relationships as well. Nodes A and B also update the data accordingly.

3. CLIENT FUNCTIONALITY, LOAD BALANCING AND CONSISTENCY

3.1 CLIENT FUNCTIONS

We have basically provided three functions to the client. Those are :-

a. **Client_remoteRead**
b. **Client_remoteWrite**
c. **Client_remoteDelete**

These function takes the address of the known node to client, and key against which operations were desired and establishes a connection with that node. Now, the known node finds the leader of the keyspace the client requested, and contacts the leader to perform appropriate action. In the Read function, the requested return string (the string stored against the key), is passed via reference to the function, and is filled in appropriately by the nodes to return it to the client. In the Write function, the string to be written against the key requested is also taken as input.

3.2 LOAD BALANCING IN READ IMPLEMENTATION

In the read operation, when the control is passed by the known node to the keyspace leader node, it implements load balancing, by allocating task to nodes where replica of the data exist, if necessary, to reduce workload of leader. This is done by maintaining a last_read parameter, which is 0, if the last read operation was done on the leader, 1, if it was done on successor of leader, and 2 if the last read operation was done on successor of successor of leader. Leader redirects the read function to its (last_read+1)%3, i.e. if last read was 2, the operation will be done on the leader itself. Thus, the same node doesn't end up taking all the requests.

3.3 BLOCKING NATURE OF THE FUNCTIONS

All the RPC calls concerned with write and delete are kept to be synchronous, thus, while execution of write and delete, the channels between

a. the client and the known node
b. Known node and Leader of keyspace client requests info from
c. Leader of keyspace and it's successors where the changes in data has to be replicated.

are blocked, due to the nature of synchronous call.

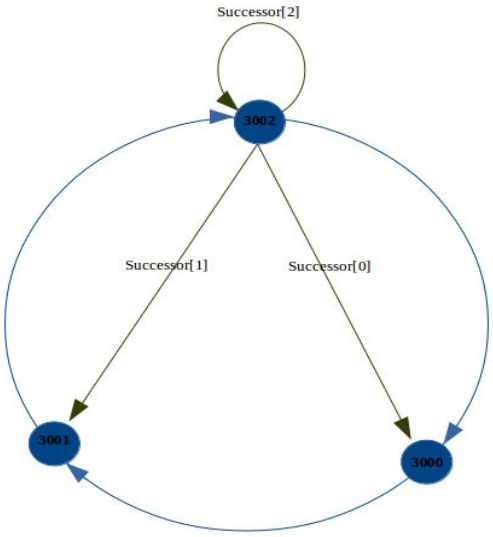
Thus, here if another client requests data to be written/read on the keyspace for which the known node is leader of/ the same keyspace which previous client requested, it will have to wait for the previous client to finish the work. In case of read, the **channels b and c** have **Async Client RPC** calls, whereas channel a has Sync RPC, which enable requests coming from some other client for that same keyspace/ the keyspace of which the known node is leader of to go through simultaneously.

4. TEST RESULTS

4.1 Join and Leave of Nodes including Node Failure

First we run chord-create.go to create a LocalNode and call Create() on it. Its Address is 127.0.0.1:3000(Node 0). Now, we run chord-join.go twice with arguments 1 & 2 to Join two nodes with Addresses 127.0.0.1:3001(Node 1) and 127.0.0.1:3002(Node 2).

From the Successor/Predecessor updated statements, we can see that we have the following topology. (right)



For 127.0.0.1:3002

Successor[0] = 127.0.0.1:3000

Successor[1] = 127.0.0.1:3001

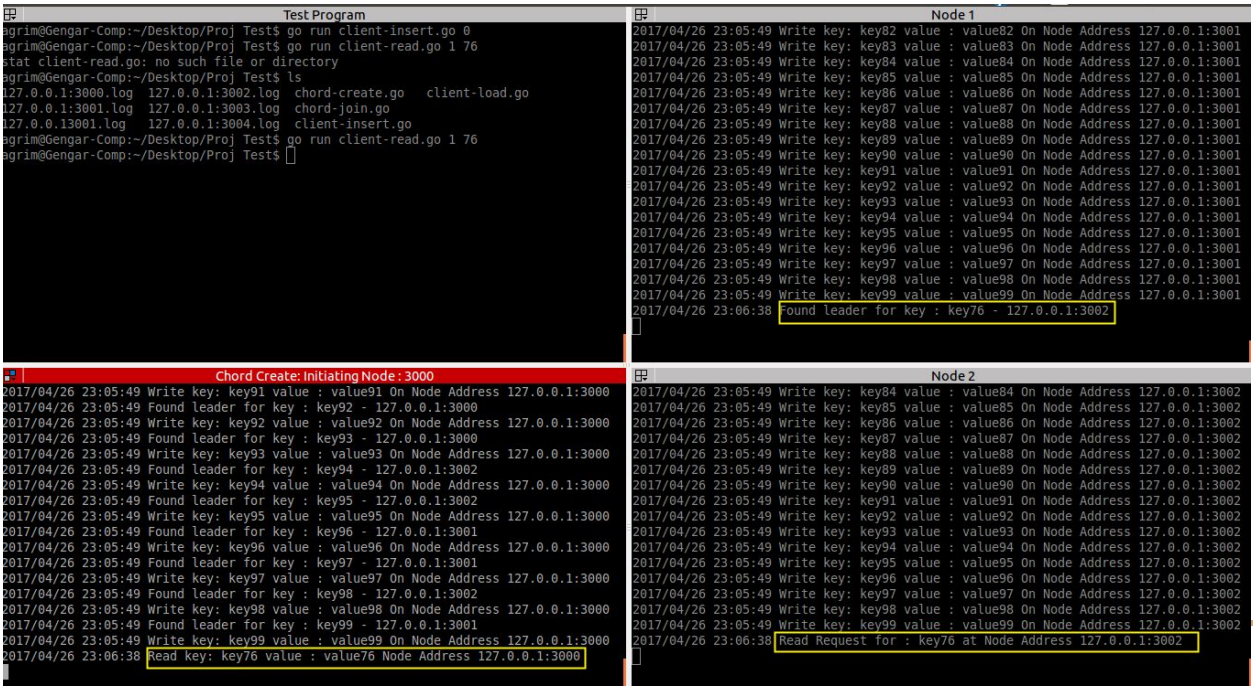
Successor[2] = 127.0.0.1:3002

Predecessor = 127.0.0.1:3001

Similarly, other nodes also update successors and predecessor.

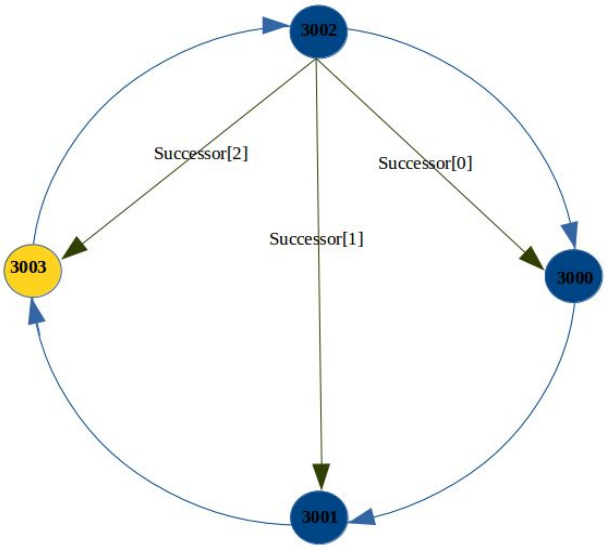
Now, we run a client program client-insert.go with argument 0 which contacts inserts 100 key value pairs into Node 0. Node 0 passes each key to its leader which replicates the data. In this case, as there are only three nodes, all nodes have all data.

Now, we run client-read.go with arguments [1 76] which contacts node 1 for key “key76” which passes to leader node 2. In the screenshot **below**, we observe that Node 2 gets the request while Node 1 was contacted.



Now we join another node using chord-join.go with argument 4. Its address is 127.0.0.1:3003(Node 3).

The topology now is (as shown on right):



For 127.0.0.1:3002, the new node joining

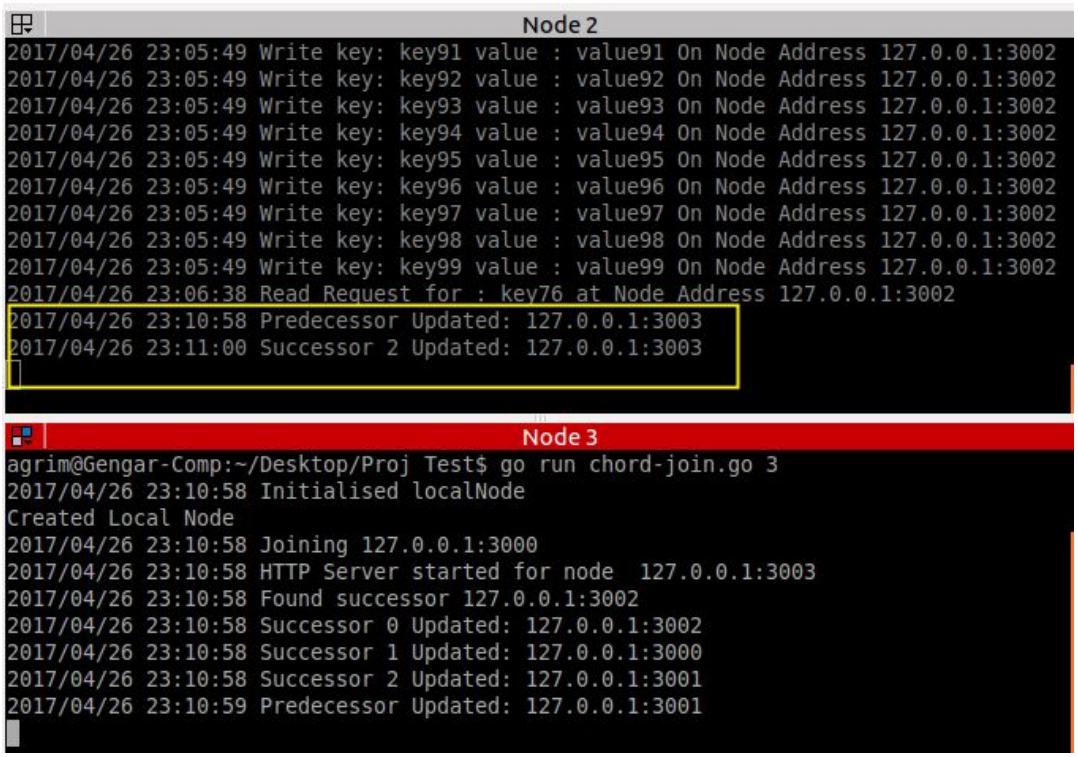
Successor[0] = 127.0.0.1:3000

Successor[1] = 127.0.0.1:3001

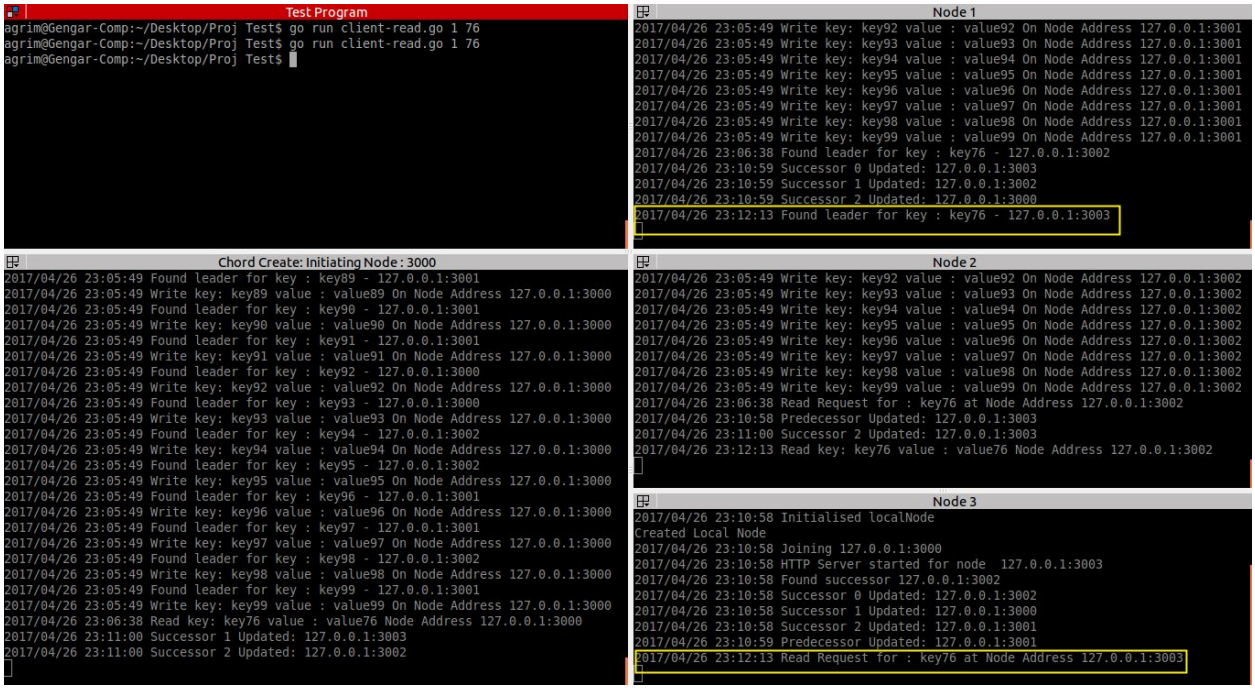
Successor[2] = 127.0.0.1:3003

Predecessor = 127.0.0.1:3003

Similarly for other Nodes. The update is shown in **below** screenshot.

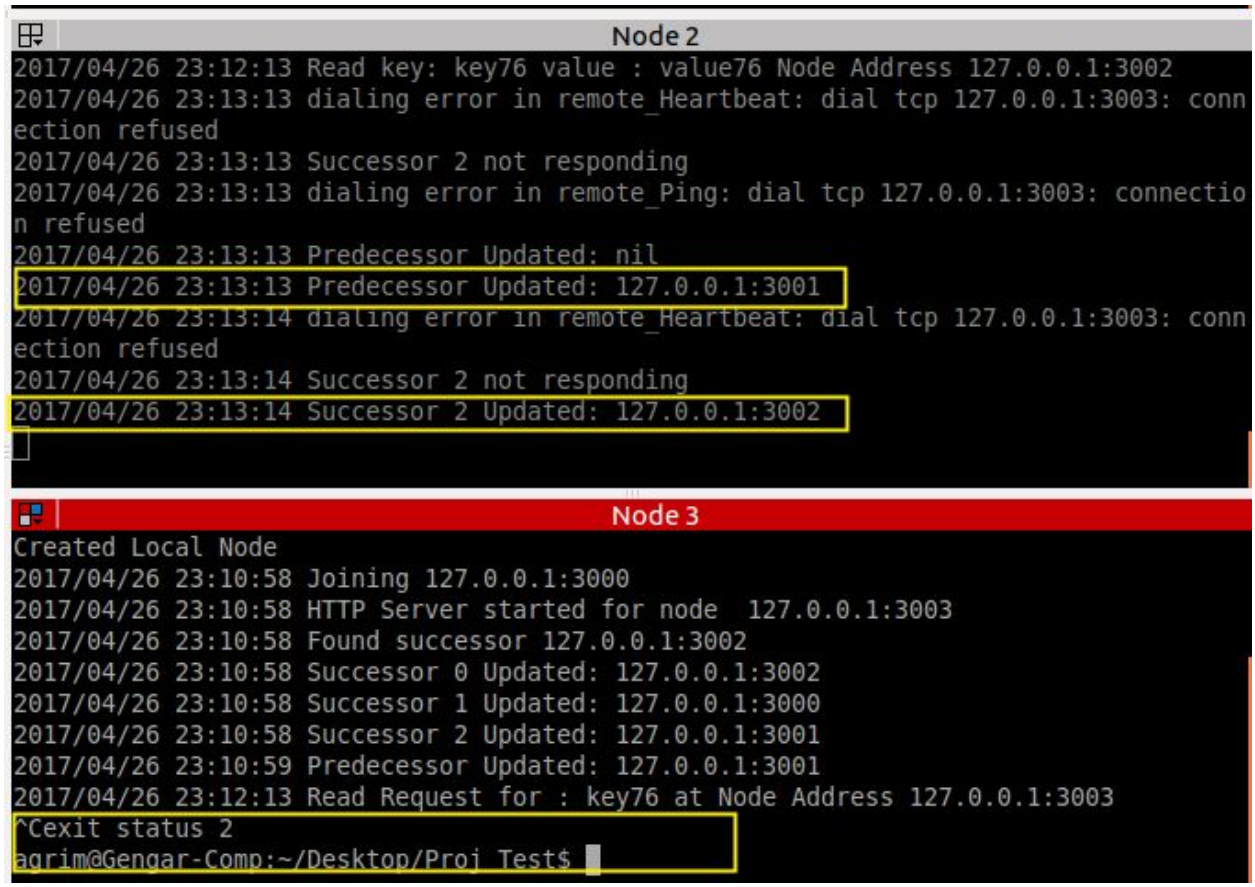


If we run client-read.go with arguments [1 76] again, node 1 passes it to Node 3 which is the leader for keyspace containing hash of “key76”. This shows that the keyspace is being split and assigned to the newly joined Node. In following screenshot, we see Node 3 receives the request as it is leader.



Now, if we kill Node 3, which simulates Node failure, the system detects that Node 3 is not responding and returns to the previous topology by updating Successor[2] and Predecessor for Node 2.

This is shown in **below** screenshot



If we run client-read.go with arguments [1 76] again, node 1 passes it to Node 2 which is the original leader for keyspace containing hash of 76. This shows that the key space was merged when the Node failed.

Similarly, on calling Leave on a Node, we observe same results.

4.2 Load Balancing

As above, we create Nodes 0, 1 and 2 and call client-insert.go

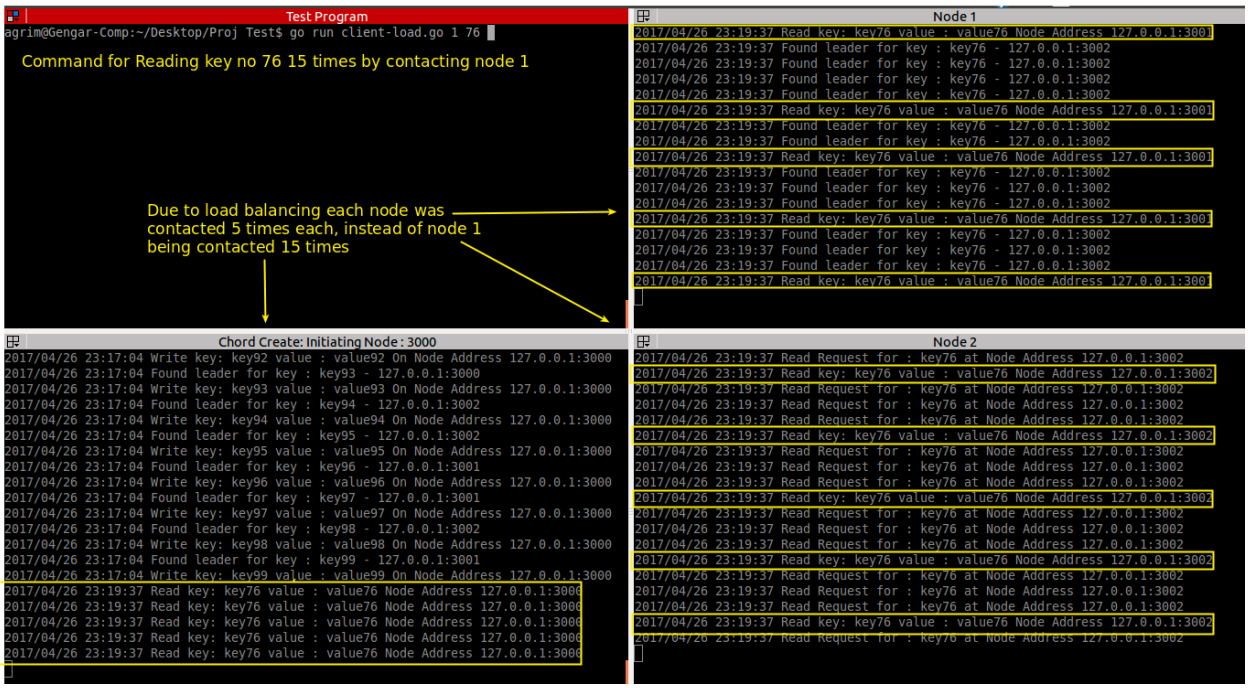
Now, we run client-load.go with arguments [1 76]. This contacts Node 1 to read value of key “key76” for fifteen times.

Here the data is replicated on all Nodes as there are only three nodes.

From the logs, we observe that each of the Nodes served the Read request 5 Times.

This shows that Read requests are uniformly balanced across the replicas.

This is shown in the following screenshot below.

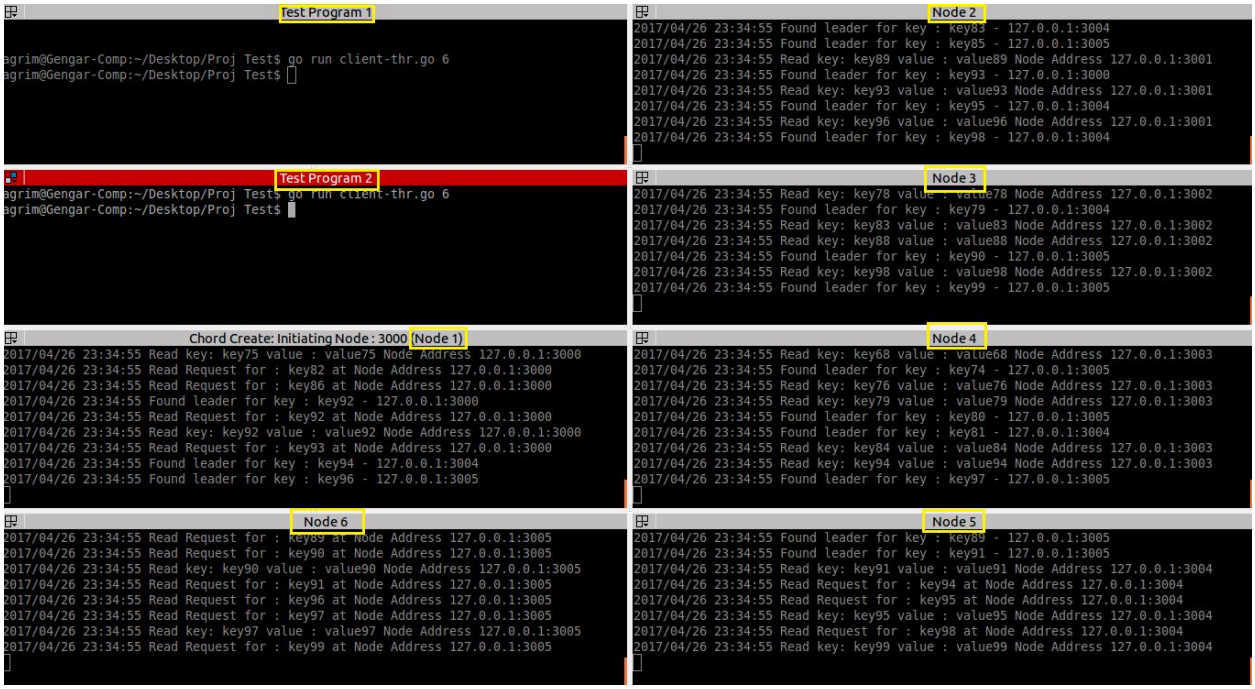


4.3 Scalability

We start Node0 and call create. Then, we insert data using client-insert.go. Now, we join 5 more nodes. Then, we run client-thr.go twice with argument 6 which sends read requests for keys from 0 to 99 by choosing a random nodes.

We observe that the system works smoothly.

The following screenshot shows the working.



5. FUTURE WORK

5.1 CHORD FINGER TABLE

We didn’t get time to maintain finger tables for quicker lookup, but since anyways, we wouldn't be able to test this feature effectively, due to limited resources, we had kept it for the last

5.2 TRANSACTIONS

Incorporation of transaction support and Two Phase commit between the coordinator and the nodes where the data is stored is not yet incorporated.

5.3 ROBUST FAILURE HANDLING

Right now our design deals with failure of nodes anywhere in system if they are isolated in time sufficient enough for system to stabilize. Also, we handle the case where at most one replica among the three fails. We are yet to handle the case where two replicas fail at once.

6. REFERENCES

1) Used https://github.com/armon/go-chord as a reference for implementing Chord

   Implementation is based on Chord_DHT.pdf provided on Piazza.

2) Golang documentation https://golang.org/doc/

3) RPC reference : Tutorial on RPC in Golang by Parth Desai