



SINGAPORE UNIVERSITY OF  
TECHNOLOGY AND DESIGN

# **Distributed Systems Project Report**

## **Group 2**

*Adhi Narayan Tharun (1003379)*

*Gerald Lim (1003371)*

*Koh Xian Ming (1003448)*

*Kwa Li Ying (1003833)*

*Nicole Lee (1003591)*

*GitHub link:*

<https://github.com/liying-kwa/50.041-Distributed-Systems-and-Computing-Project>

## **Introduction**

Our project focuses on creating a key-value distributed storage system with features similar to that of Amazon's DynamoDB. Certain features that allow DynamoDB to ensure reliability and efficiency on a large scale that we plan to implement are scalability, high availability, and fault tolerance.

## **Overview**

The SUTD Subject Enrolment System is prone to lags and server failures when there is a surge in the number of transactions, especially at around 9am on the first day of enrolment. Furthermore, some users complain of being unable to enrol into the subject despite the system showing its availability. This may be due to poor scalability or load balancing. As a result, there is a need for a better-designed distributed system to ensure high availability with eventual consistency.

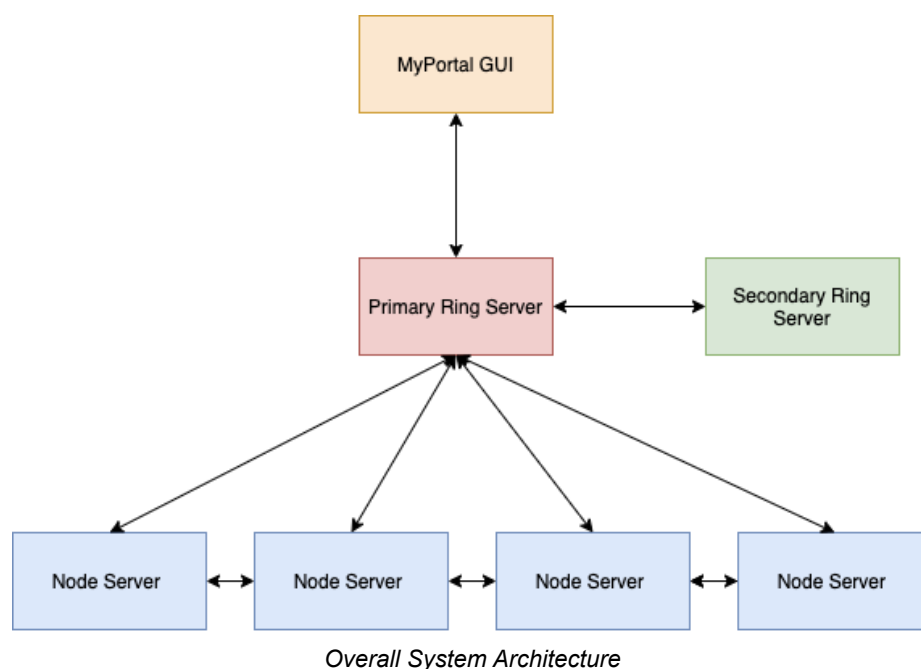
Our distributed database would be able to scale horizontally as the number of transactions increases. This is done through consistent hashing, which would allow us to avoid having to rehash all the keys when scaling the system, thus ensuring smooth load balancing. By implementing replication, we can ensure high availability of the server. Students would be able to add or remove classes even amidst server failures and the server would not just lag during course enrollment. Lastly, through fault tolerance, if a server is down, all read and write operations will be sent to another replica which will hold metadata regarding the intended server recipient. This ensures the desired availability and durability guarantees.

The RingServer maintains information on the ring structure. The client can make GET(CourseID) and PUT(CourseID, Count) requests to the RingServer. This RingServer then contacts the relevant NodeServers which store the data to perform the read or write operation. Location of data is decided through consistent hashing. Permanent failure of the RingServer is addressed by a backup RingServer.

## Overall System Architecture

The overall system architecture comprises 2 core components – front-end and back-end. The back-end should be able to simulate a realistic distributed system and display core features of DynamoDB such as consistent hashing for high scalability and replication for high availability. For databases, we will be leveraging a simple file system that allows for read and write of key-value pairs. The front-end consists of a GUI that serves as an intuitive demonstration for a booking system. It should be capable of simple booking functionalities via GET and PUT requests with the back-end. For simple demonstration purposes, the above implementation will be simulated on a local machine.

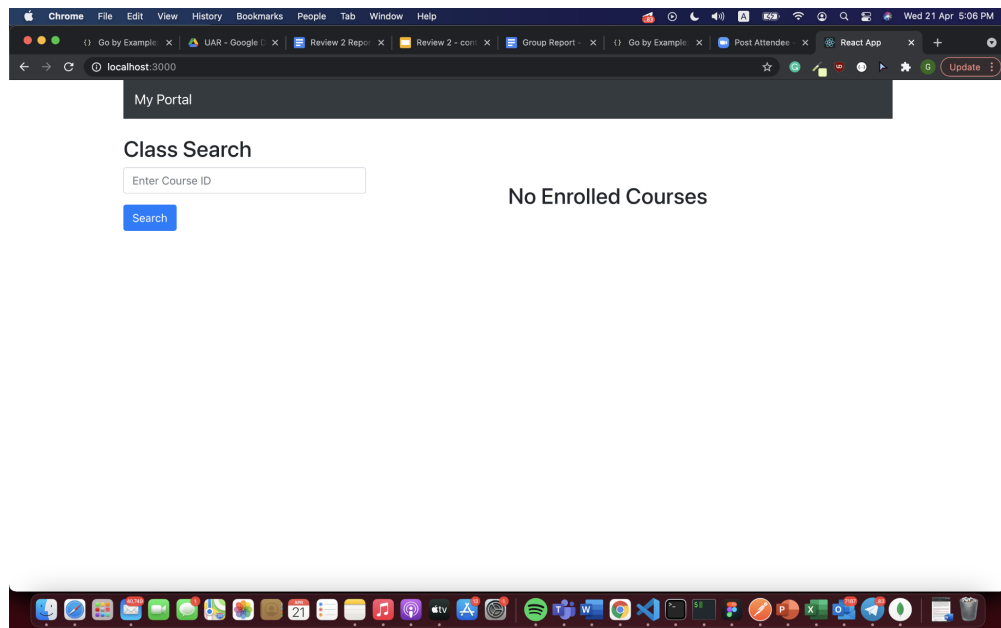
The overall system architecture can be represented by the diagram below. The role of each component will be further elaborated on for the rest of the report.



## Front-end

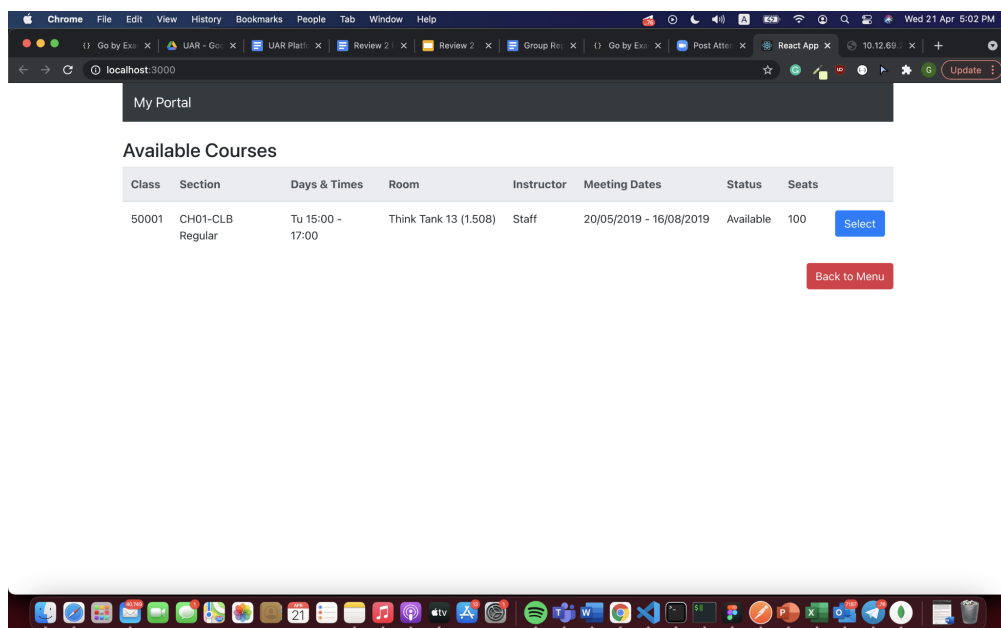
In order to simplify an abstract idea like distributed systems, we created a front-end to demonstrate its usefulness. We have leveraged on React Framework to build a booking system that allows a user to query for available courses and enrol into a selected course. To perform these actions, we have made use of the AXIOS library to perform GET and POST requests from the back-end. With strong capabilities at the back-end, the front-end should not experience lags due to an influx of heavy traffic. The front-end also pings the primary server periodically to check for its availability. Should the primary server fail, it should be able to inform users of a server failure and revert back to normal as soon as the back-end resolves the fault. Below are some screenshots of the front-end. The screenshots below illustrate the process of enrolling in a course.

When the user first enters the webpage, he sees the home screen as shown in the screenshot below.



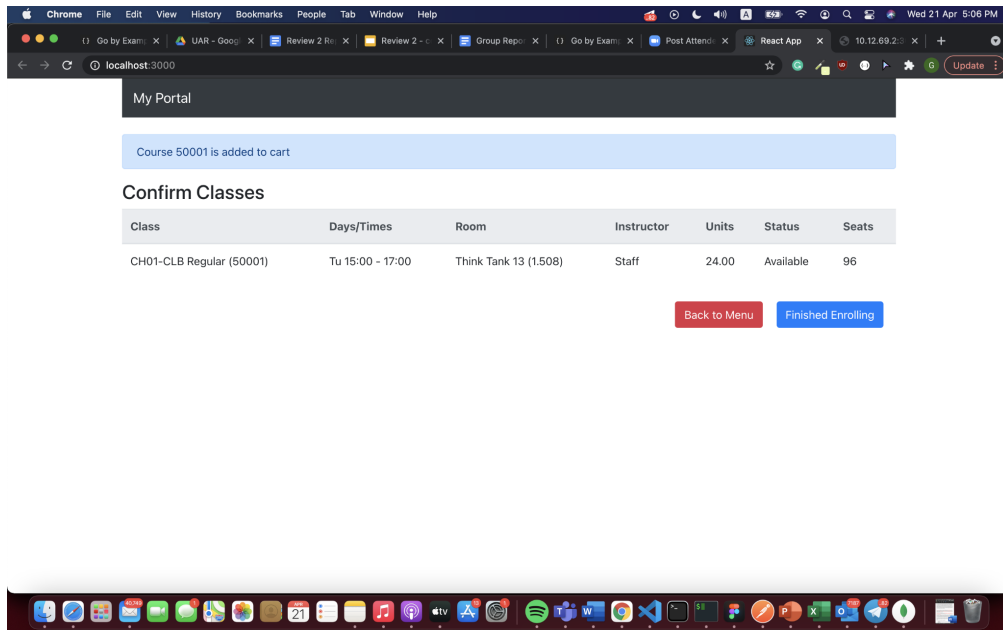
*Home Page of My Portal*

When the user searches for a particular courseID, the GET(CourseID) function is executed on the database, which corresponds to a read operation. The database returns the number of seats (i.e. count) available in the course and the frontend displays this number under the “Seats” column as shown in the screenshot below.



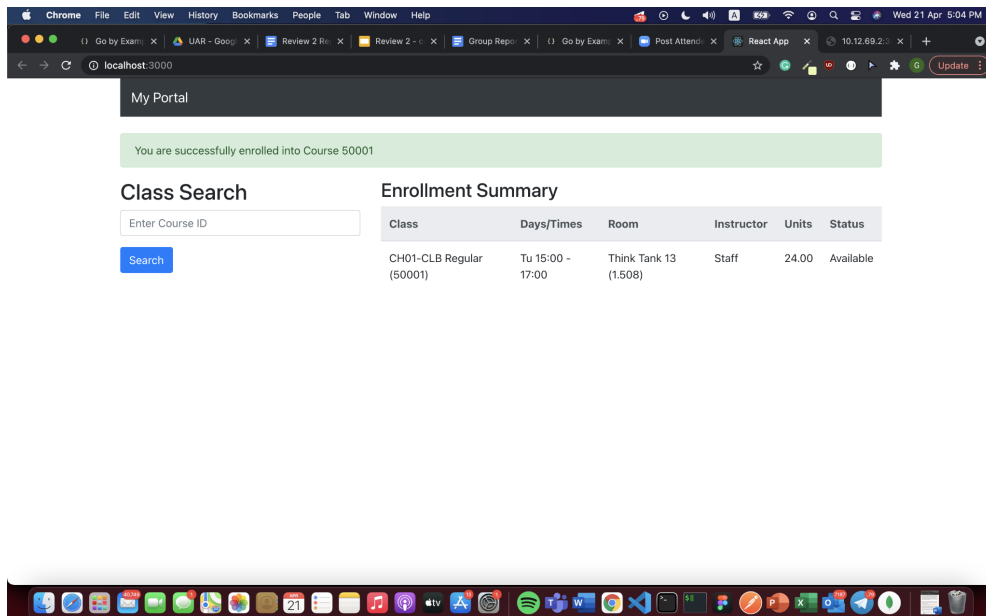
*Select Page of My Portal*

Following this, the course is added to the cart when the user clicks on the “Select” button and he is brought to the “Finish Enrolling” page below. Once he clicks on the “Finish Enrolling” button, the PUT(CourseID, Count-1) function is executed on the database, which corresponds to a write operation.



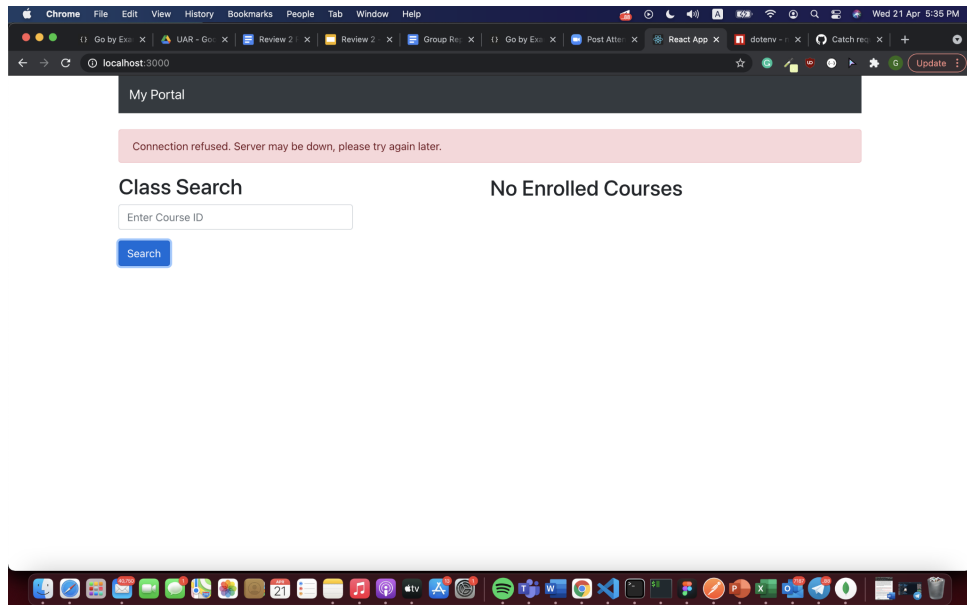
*Finish Enrolling Page of My Portal*

The database then returns a response message with status code that indicates whether the write operation was successful or not. The following screenshot shows a successful enrollment in the course.



*Home Page of My Portal (Successful Enrolment)*

The next screenshot shows an unsuccessful enrollment in the course, which happens if the database refuses connection (if the IP address or port is wrong, or if the database is not up and listening to read/write operations).

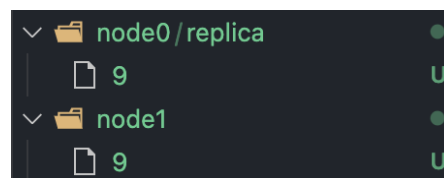


*Home Page of My Portal (Connection refused)*

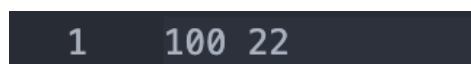
## Back-end

In our distributed system, there are 2 types of servers - RingServer and NodeServer. We model our system structure after a virtual ring where NodeServers reside. A centralised RingServer exists as the point of contact between the front-end and the ring structure. To cater for fault tolerance, there exists a secondary RingServer that acts as a back-up in the case that the primary server experiences failure, the secondary server will resume the duties of the previous RingServer. For higher scalability, we have implemented consistent hashing analogous to DynamoDB that allows for easy adding and removal of NodeServers to and from the ring structure. Lastly, NodeServers are imbued replication features (RF=3 or otherwise configured) that allows for higher availability. Whenever a new node is added into the ring structure, it's content will be replicated to (RF-1) of its successor NodeServers.

Each NodeServer has data storage and they are initialized whenever a NodeServer joins the virtual ring structure. We kept it simple by having text files containing only the courseid and the count (key-value) stored in the NodeServer's folder locally. For each write, the courseid will be hashed and allocated to the nearest NodeServer. For the same hashed value, it will be appended to the same text file and replicated to other NodeServers if required. The database structure for a NodeServer can be represented by the diagram below.



*A snapshot of the database structure*



*A snapshot of a write to a file (Courseid : Count)*

For inter-process communication over the network, we will be leveraging the net/http package. All servers in the distributed system will be registered to a set of handlers with server routes using the *http.HandlerFunc* function that takes in a function as an argument. Functions serving as handlers take a *http.ResponseWriter* and a *http.Request* as arguments. The response writer will be used to fill in HTTP response and request is used to read query params. Lastly, we call *ListenAndServe* with a dedicated port to run the servers in the background. Having all servers communicate via HTTP allows for servers to communicate even at remote locations, hence simulating a realistic distributed network. (*Refer to appendix for HTTP endpoints*)

## Features

The distributed database we have built consists of the following features:

Feature	Rationale	Technique
Scalability	We want our system to be able to scale horizontally as the number of transactions increase. Consistent hashing will allow us to avoid having to rehash all the keys when scaling the system, thus ensuring smooth load balancing.	Consistent hashing
High availability for writes	Students should be able to add or remove classes even amidst server failures. Ability to write to any one server at any point of time. Can compromise some consistency to reduce lag.	Replication
Fault Tolerance (RingServer)	The RingServer acts as a middleman for the frontend and the nodes. It is important to consider fault tolerance for the RingServer because it is a central point of failure.	Primary and Secondary RingServer

Given more time, we would have implemented the following features to enhance our distributed system:

Feature	Rationale	Technique
Consistency & Eventual Consistency	We want the cart to reflect the right modules added in the event the writes to the replica are not completed on time	Sloppy Quorum, Data versioning (vector clocks for conflict resolution on read)
Fault Detection	When one server goes down, need to inform the other servers	Gossip-based protocol
Fault Tolerance (NodeServer)	If a server is down temporarily, all read and write operations will be sent to another replica which will hold metadata regarding the intended server recipient. When the failed server recovers, all data will be transferred from the replica back to the original server. Hence, at any point of time, the number of replicas remains the same. This ensures the desired availability and durability guarantees.	Temporary: Hinted handoff Permanent: Merkle tree

These features and their implementations would be discussed further in the next few sections.



# High Scalability

## Consistent hashing for Reads and Writes

Consistent hashing reduces the need to redistribute most of the data when a server is added or removed. It is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed hash table by assigning them a position on an abstract circle, or hash ring. This allows servers and objects to scale without affecting the overall system.

For consistent hashing to work, the output range of a hash function is treated as a fixed circular space or ring, which means the largest hash value wraps around to the smallest hash value.

Each object key will belong to the server whose hash value is closest, in a counterclockwise direction (or clockwise, depending on the conventions used). In other words, to find out which server to ask for a given key, we need to locate the key on the circle and move in the ascending angle until we find a server. To do so, the hash of the key is calculated and mapped to the key space of the hash ring and the nearest node will be responsible for coordinating the reading and writing for this object.

Essentially, each node is responsible for several regions in the ring between itself and its predecessor node on the ring. The advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected. Removing a node results in its object keys being randomly reassigned to the rest of the servers, leaving all other keys untouched whereas if a node is added, only  $k/N$  keys need to be remapped when  $k$  is the number of keys and  $N$  is the number of servers (more specifically, the maximum of the initial and final number of servers).

For this project, the key represents the CourseID of a subject while the value represents the Count (number of remaining slots) of the associated CourseID.

### Hash Function

1. The key of the object is converted into bytes and then hashed using the cryptographic MD5 function
2. Convert the MD5 string from hex to integer and then perform a modulo on the resulting integer by the upper bound of the ring's keyspace.

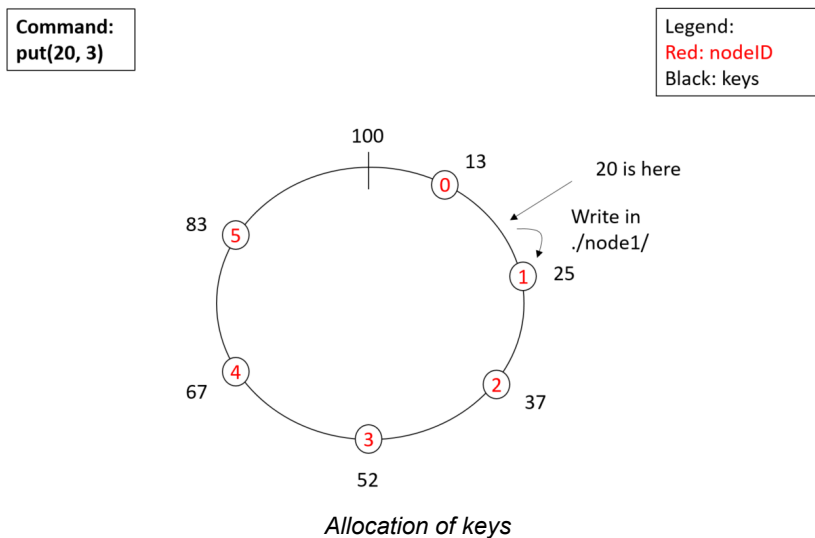
### Maintaining the Ring structure

1. When a NodeServer is first instantiated, it will be given the URL of the RingServer. The RingServer will then randomly assign a position on the ring for this NodeServer.
2. If the position has already been taken, the Ring will attempt to randomly assign a position in the ring that has not been assigned before.

## Allocation Of Keys

To allocate the key, the RingServer must be partition aware and maintain a consistent view on the ring structure. This is done via `AllocateKey()` function which the RingServer can execute

1. We first generate a hash using the Hash Function described above.
2. Next, perform a clockwise search on the node starting from 0 all the way to the upper bound of the keyspace and search for the first node located on the ring whose hash is greater than the hash of the key.
3. If the hash of the key exceeds all of the existing nodes' hash in the ring, assign it to the node with the lowest hash value in the ring.
4. Once the key is assigned, the RingServer handling the request will relay it to the assigned node.



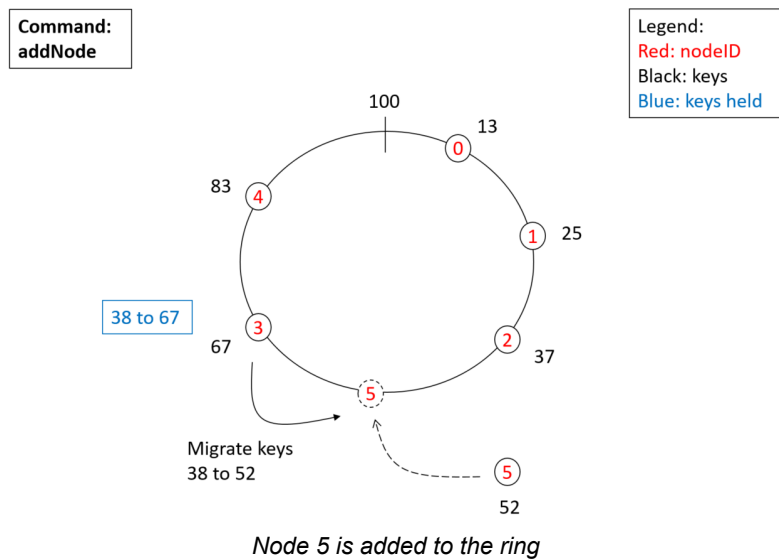
## Data Migration (when a node is added)

### Storage Data Structure

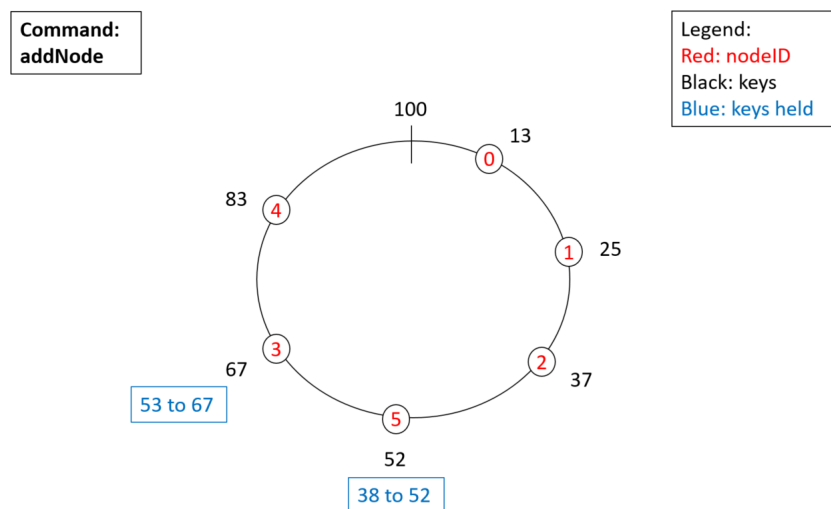
Each node contains files named according to the hash values that it is assigned. Each file would contain the keys corresponding to the hash together with its value. Each line in the file would represent a record of key-value pairs (e.g. CourseID : Count for this project).

### Splitting of Keys

When a node is added, it is assigned a random position in the ring (that is not already taken). It will take on some of the keys from the successor node:



Upon adding a new node at position 52, the new node manages keys that hash from 38 to 52. They were previously managed by Node 3. As such, some existing (key,value) data will have to migrate from the successor's node to this new node. Data with keys that fall within the hash range of 38 and 52 will be migrated by write requests from the successor's node to this new node:

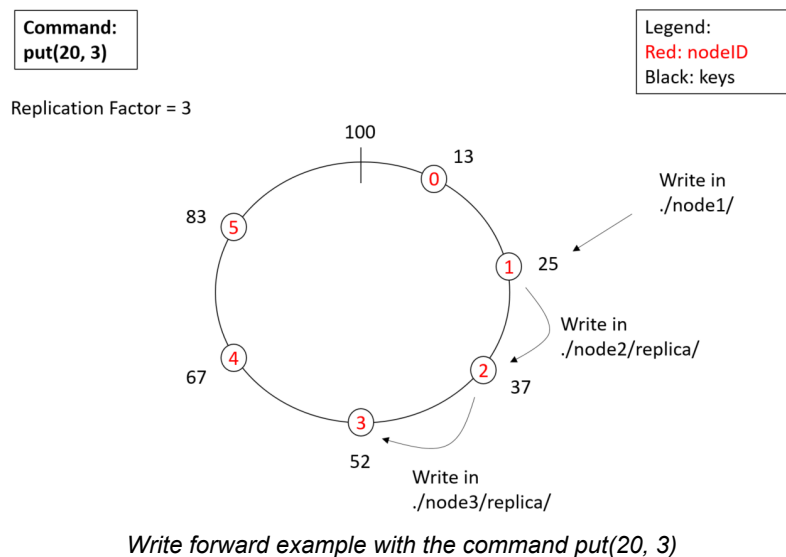


# High Availability by Replication

## Write Forward

Write requests will occur as per normal – the RingServer will read from / write to the node with the closest counterclockwise hash from the hash of the key. We will refer to this (closest) node as the coordinator node. To replicate the data based on the Replication Factor (RF) defined as a constant in the code, the coordinator node will forward data to its next (RF-1) nodes, also known as its successors, to be written to the `./node<ID>/replica/` directory.

In the example below, the client executes a write command `PUT(20, 3)`. The corresponding coordinator node is node1 with a hash of 25 and the RingServer writes this data to the `./node1/` directory. Following this, node1 replicates the data by forwarding this write request to node2 and node3, to store the data in the `./node2/replica` directory and `./node3/replica` directory respectively.



For proper replication, each node will keep a copy of who are its (RF-1) successors, which are the clockwise neighbouring nodes, and (RF-1) predecessors, which are the counterclockwise neighbouring nodes.

**Command:**  
**addNode**

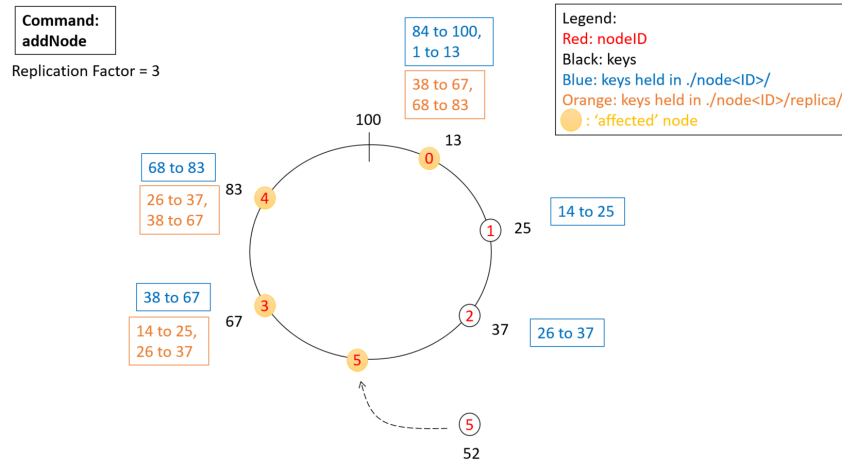
Replication Factor = 3

**Legend:**  
 Red: nodeID  
 Black: keys  
 Blue: keys held in ./node<ID>/  
 Orange: keys held in ./node<ID>/replica/  
 Green : new node's predecessor  
 Blue : new node's successor

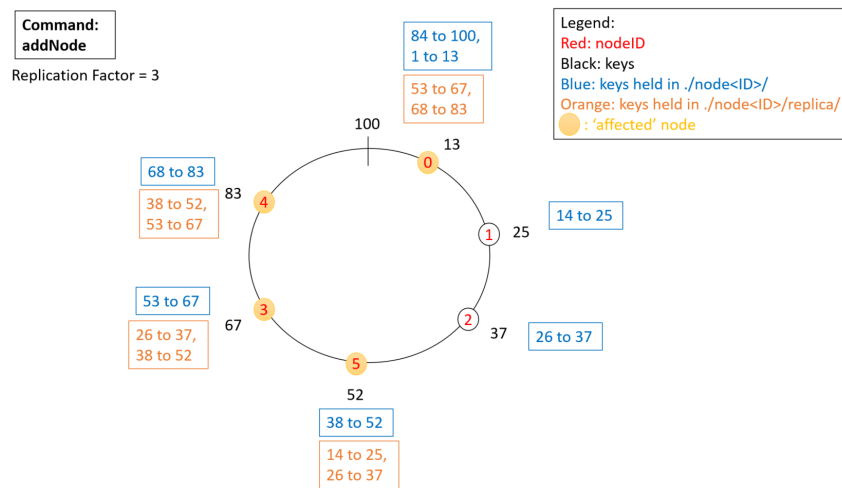
## Reload Replica

- The new node itself
- The new node's successor
- The new node's successor's successors

12

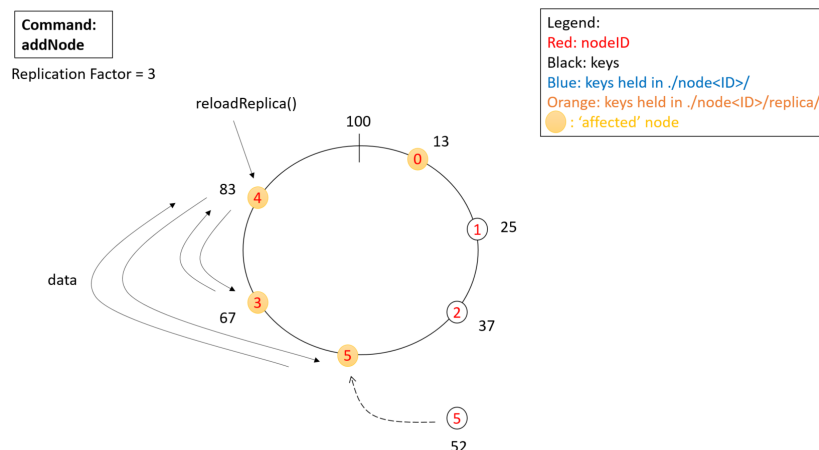


*Example of Reload Replica – Before new node is added*



*Example of Reload Replica – After new node is added*

To refresh the data under the `./node<ID>/replica/`, the RingServer will notify the affected nodes and each affected node will reload its replica by requesting data from their predecessors to be stored in its `./node<ID>/replica/` directory. This action is illustrated in the diagram below:



*Example of Reload Replica – An 'affected' node calls the reload-replica function*

## Fault Tolerance (RingServer)

Our database is made to handle permanent failure of the RingServer. The RingServer fails when machines that contact it do not receive a response from it before timeout.

With fault tolerance, we aim to achieve a high level of availability and performance despite failure of the primary RingServer.

### Impact of permanent RingServer failure

As the RingServer coordinates writing and reading of nodes, once the RingServer is down, the front-end would not be able to write and read values to the database. Users would also not be able to add or remove nodes to the ring structure. The RingServer stores data of the ring structure. In the case of a failure, we would have a backup server that takes over the ring structure, as well as all the functionalities of the RingServer.

### Secondary RingServer

When the system is set up, a secondary RingServer would be set up alongside the primary RingServer. This secondary RingServer periodically pings the primary RingServer to check if it is alive. The primary server would then respond each time with the current ring structure. The secondary RingServer then stores the most updated ring structure, in case the primary RingServer fails.

The primary RingServer fails when either:

1. The secondary RingServer does not receive response from it before timeout, or
2. Frontend server tries to write to the database but does not receive response

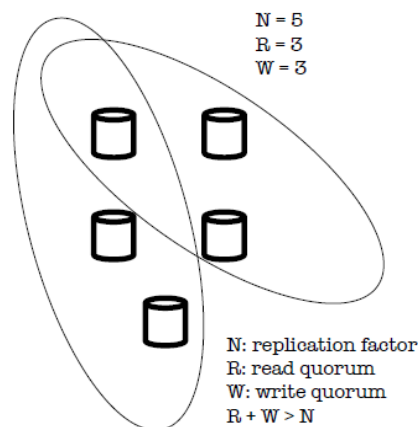
When the primary RingServer fails, the secondary RingServer will take over as the RingServer. It broadcasts to all the NodeServers to inform them of its IP address and port number and that it is now the new RingServer, so that all nodes will know to contact it instead of the failed RingServer. Once it has received an acknowledgement from all nodes, it will officially become the RingServer and it will start listening to the front-end and all nodes to take over all functionality of the primary RingServer.

## Correctness/Consistency

Our database is consistent as it always gets the most updated value from the coordinator node that contains the data.

### Further Improvements

Our database can be more efficient if we implement the sloppy quorum technique through the use of logical clocks to speed things up. Whenever a request is received by the NodeServer, it should attach a logical timestamp to the request before sending them to the respective node. On reconciliation, we will always retrieve the data entry with the latest timestamp if there are conflicting reads.



*Illustration of Sloppy Quorum*

With sloppy quorum, the Ringserver can forward read/write requests to all NodeServers. The read operation is completed when the Ringserver receives  $R$  or more replies. The write operation is completed when the Ringserver receives  $W$  or more replies. Sloppy quorum ensures that at least one of the read replies will contain the most updated value. The Ringserver can just compare the timestamps of the replies to get the most updated value.



# Experiments/Evaluations/Tests

## Add/Remove Nodes

When we register a NodeServer, the NodeServer is informed of its nodeID, hash, predecessors and successors by the RingServer:

```
$ go run nodeserver.go
NodeServer> register 5002
{0 192.168.56.1 5002 4 map[] map[]}
Successfully registered. Response: {"Id":0,"Ip":"192.168.56.1","Port":"5002","Hash":"4","Predecessors":{},"
Successors":{}}
2021/04/24 23:20:54 [NodeServer] Started and Listening at 192.168.56.1:5002.
```

*A NodeServer's registration from its own point of view*

## Write Operation

Upon writing to the RingServer, it will initiate a write request to the respective NodeServer. If the file for the hash does not exist in the node, it will be created. Otherwise, the file will be updated with a new line if it is a new key or replaced if the key is already existing in the file.

```
NodeServer> 2021/04/24 23:21:06 [NodeServer] Received Write Request
File does not exist
Creating file....
Successfully wrote to node!
2021/04/24 23:21:10 [NodeServer] Received Write Request
File does not exist
Creating file....
Successfully wrote to node!
2021/04/24 23:21:17 [NodeServer] Received Write Request
File already exists, proceeding to update file...
Successfully wrote to node!
```

*A write operation from the corresponding coordinator NodeServer's point of view*

When we write key-value pairs to the RingServer, it forwards the write request to the respective node:

```
$ go run ringserver.go p
2021/04/24 23:20:48 [RingServer] Started and Listening at 192.168.56.1:5001 for Nodes.
2021/04/24 23:20:48 [RingServer] Started and Listening at 192.168.56.1:3001 for Frontend.
RingServer> 2021/04/24 23:20:54 [RingServer] Receiving Registration Request from a Node
Adding Node #0 with 192.168.56.1:5002 to the ring...
ring
{
  "MaxID": -1,
  "RingNodeDataMap": {
    "4": {
      "Id": 0,
      "Ip": "192.168.56.1",
      "Port": "5002",
      "Hash": "4",
      "Predecessors": {},
      "Successors": {}
    }
  }
}

RingServer> write 50001 100
map[4:{0 192.168.56.1 5002 4 map[] map[]}]
this is the hash below:
1
Successfully wrote to node. Response: 200 OK -- Successfully wrote to node!

RingServer> write 50002 200
map[4:{0 192.168.56.1 5002 4 map[] map[]}]
this is the hash below:
7
Successfully wrote to node. Response: 200 OK -- Successfully wrote to node!

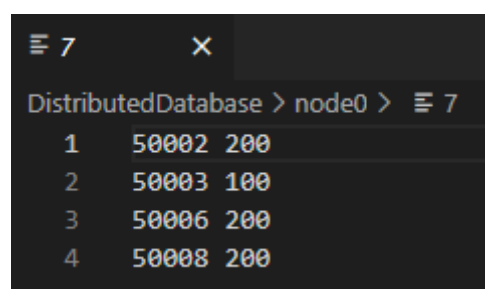
RingServer> write 50003 100
map[4:{0 192.168.56.1 5002 4 map[] map[]}]
this is the hash below:
7
Successfully wrote to node. Response: 200 OK -- Successfully wrote to node!

RingServer>
```

*A write operation from the RingServer's point of view*

As there is only one NodeServer currently, all write requests will be forwarded to Node 0.

Keys 50002, 50003, 50006 and 50008 hashes to 7 and will be written to the same file (file name 7) in Node 0 with each entry on a newline:



```
DistributedDatabase > node0 > 7
1 50002 200
2 50003 100
3 50006 200
4 50008 200
```

*File contents after write requests*

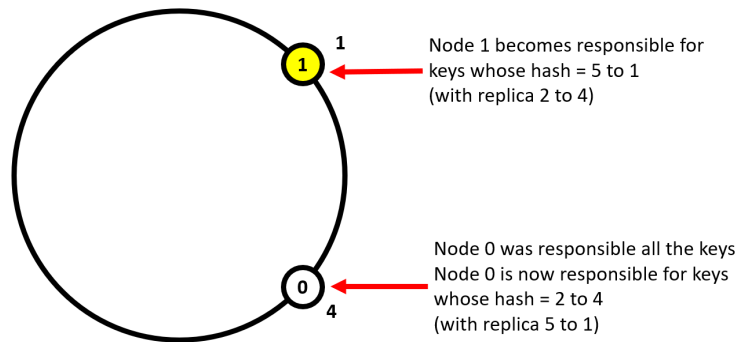
## Migration of Data (When a NodeServer is added)

When we add a new NodeServer, the data should split according to the hash. In this example, Node 1 is added to the ring at a hash position of 1. Hence, Node 0 should migrate the data of keys with hash in the range of 5,6,7,8,9,1 to Node 1 such that Node 0 will contain data of keys with hashes of 2,3,4 itself:

```
ring
{
  "MaxID": -1,
  "RingNodeDataMap": {
    "1": {
      "Id": 1,
      "Ip": "192.168.56.1",
      "Port": "5003",
      "Hash": "1",
      "Predecessors": {
        "4": {
          "Id": 0,
          "Ip": "192.168.56.1",
          "Port": "5002",
          "Hash": "4"
        }
      },
      "Successors": {
        "4": {
          "Id": 0,
          "Ip": "192.168.56.1",
          "Port": "5002",
          "Hash": "4"
        }
      }
    },
    "4": {
      "Id": 0,
      "Ip": "192.168.56.1",
      "Port": "5002",
      "Hash": "4",
      "Predecessors": {
        "1": {
          "Id": 1,
          "Ip": "192.168.56.1",
          "Port": "5003",
          "Hash": "1"
        }
      },
      "Successors": {
        "1": {
          "Id": 1,
          "Ip": "192.168.56.1",
          "Port": "5003",
          "Hash": "1"
        }
      }
    }
  }
}
```

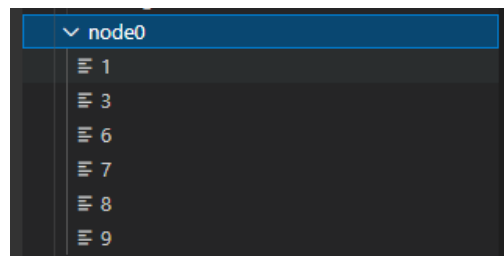
RingServer> █

*NodeServers' hashes after being added to the ring*

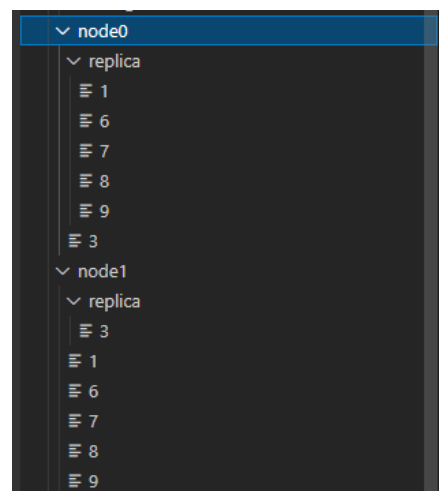


*Illustration of how data migration should happen after Node 1 is added*

We observe that the migration is indeed correct with each node having the correct replicated data as well:



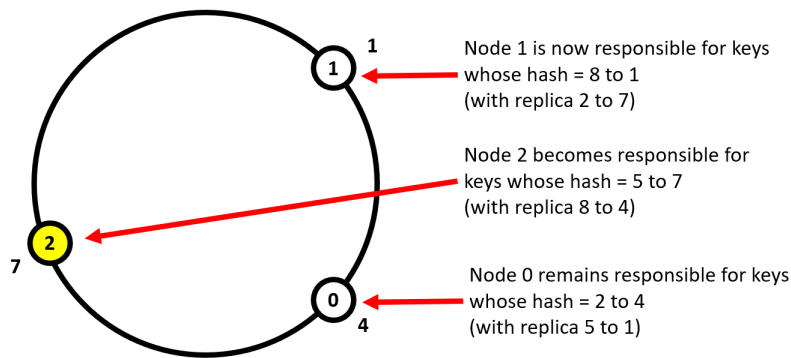
*Node0' folder contents before adding Node1*



*Nodeservers' folder contents before adding Node1*

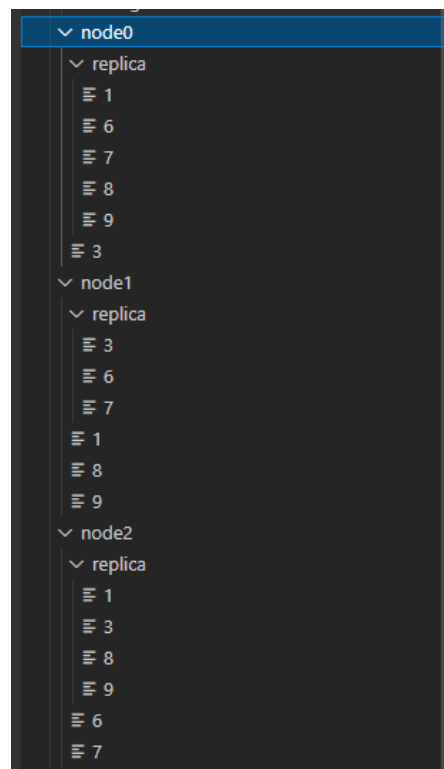
### **Reload-Replica (When a NodeServer is added)**

Continuing from the test setup above, we added another new NodeServer and observed the updated ring structure. Nodes 0, 1 and 2 have hashes of 4, 1 and 7 respectively.



*Illustration of how data migration should happen after Node 2 is added*

From the Replication Section, we know that a NodeServer should have its (RF-1) predecessors' data stored under its `./node<ID>/replica/` folder. In this case, Node 0 should have files with hashes 5 to 1 (mod 10), Node 1 should have files with hashes 2 to 7 (mod 10), and Node 2 should have files with hashes 8 to 4 (mod 10). From the nodes' folder contents, we know that the transfer of data to be stored as replica (i.e. reload-replica) is indeed correct with each node having the correct replicated data as well:



*Nodeservers' folder contents before adding Node 2*

## Fault Tolerance (RingServer)

To test the fault tolerance for the RingServer, we purposely crash the primary NodeServer and observe the output:

ad replica. Response: 200 OK — Successfully replicated data to affectedNode! Successfully told affectedNode to reload replica. Response: 200 OK — Successfully replicated data to affectedNode! Successfully told affectedNode to reload replica. Response: 200 OK — Successfully replicated data to affectedNode! ^Csignal: interrupt (base) nicole@Nicolas-MacBook-Pro DistributedDatabase %	Informing nodes of new primary server Successfully wrote to node. Response: Counter = 2, length of ringnode = 2 Secondary ring server received replies from all nodes 2021/04/25 16:58:36 [RingServer] Started and Listening at 10.12.221.55:5002 for Nodes. 2021/04/25 16:58:36 [RingServer] Started and Listening at 10.12.221.55:3002 for Frontend.	Successfully replicated data to affectedNode! Successfully wrote to node! Successfully requested for transfer of data (to replicate) from predecessor. Response: 200 OK — Successfully replicated data to affectedNode! [Node 0] Received notice that primary ring server is down, sending acknowledgement. [Node 0] Changing ring server port to 5002	Successfully requested for transfer of data (to replicate) from predecessor. Response: 200 OK — Successfully replicated data to affectedNode! Successfully wrote to node. Response: 200 OK — Successfully wrote to node! NodeServer> [Node 1] Received notice that primary ring server is down, sending acknowledgement. [Node 1] Changing ring server port to 5002
--	--	---	---

### Secondary RingServer takes over as the primary RingServer

To ensure that the system continues to be functional, we test the write operation using the new primary RingServer (previous Secondary RingServer) that just took over:

ad replica. Response: 200 OK — Successfully replicated data to affectedNode! Successfully told affectedNode to reload replica. Response: 200 OK — Successfully replicated data to affectedNode! Successfully told affectedNode to reload replica. Response: 200 OK — Successfully replicated data to affectedNode! ^Csignal: interrupt (base) nicole@Nicolas-MacBook-Pro DistributedDatabase %	ed and Listening at 10.12.221.55:3002 for Frontend. write 7 8 map[4:{1 10.12.221.55 2 4 map[8:{0 10.12.221.55 1 8}] map[8:{0 10.12.221.55 1 8}]} 8:{0 10.12.221.55 1 8 map[4:{1 10.12.221.55 2 4}] map[4:{1 10.12.221.55 2 4}]]] this is the hash below: 0 Successfully wrote to node. Response: 200 OK — Successfully wrote to node! RingServer>	Successfully wrote to node! Successfully requested for transfer of data (to replicate) from predecessor. Response: 200 OK — Successfully replicated data to affectedNode! [Node 0] Received notice that primary ring server is down, sending acknowledgement. [Node 0] Changing ring server port to 50022021/04/25 16:59:25 [NodeServer] Received Write Request File does not exist Creating file.... Successfully wrote to node!	that primary ring server is down, sending acknowledgement. [Node 1] Changing ring server port to 50022021/04/25 16:59:25 [NodeServer] Received Write Request 10.12.221.55:1 File does not exist Creating file.... Writing message to NodeServer at 10.12.221.55:1 Successfully wrote to node! Successfully wrote to node. Response: 200 OK — Successfully wrote to node!
--	---	--	--

### Write operation succeeds after secondary RingServer took over as primary RingServer

## Concluding Remark/Reflection

Inspired by DynamoDB, we prototyped a highly available and scalable data store, used for storing key-value records. We have provided the desired levels of availability and performance by handling failures in the RingServer. Our database is also incrementally scalable and allows service owners to scale up and down based on their required demands. We also allow service owners to configure their replication factor to meet their desired performance, durability and consistency. We have demonstrated that decentralised techniques can be combined to provide a single highly-available system and shown that an storage system that follows eventual consistency can be a building block for highly available applications.

While the project was really hard, it was also very fulfilling when each feature was implemented successfully. This project gave us good insights to how distributed systems are implemented and the considerations that should be taken in such systems.

## Appendix

S/N	Server Type	Endpoint	HTTP Verb	Payload	Caller
1	RingServer	/read-from-node	GET	-	Client
2	RingServer	/write-to-node	POST	{message: Message}	Client
3	RingServer	/add-node	POST	{nodeData: NodeData}	NodeServer
4	RingServer	/remove-node	POST	{nodeData: NodeData}	NodeServer
5	RingServer	/send-res	GET	-	NodeServer
6	NodeServer	/read	GET	-	RingServer
7	NodeServer	/write	POST	{message: Message}	RingServer
8	NodeServer	/update-predecessors	POST	{predecessors: SimpleNodeData}	RingServer
9	NodeServer	/update-successors	POST	{successors: SimpleNodeData}	RingServer
10	NodeServer	/migrate-data	POST	{newNodeData: NodeData}	RingServer
11	NodeServer	/reload-replica	POST	{affectedNode: SimpleNodeData}	RingServer
12	NodeServer	/replicate	POST	{thisNodeData: NodeData}	NodeServer
13	NodeServer	/update-nodes	POST	{message: PortNo}	RingServer