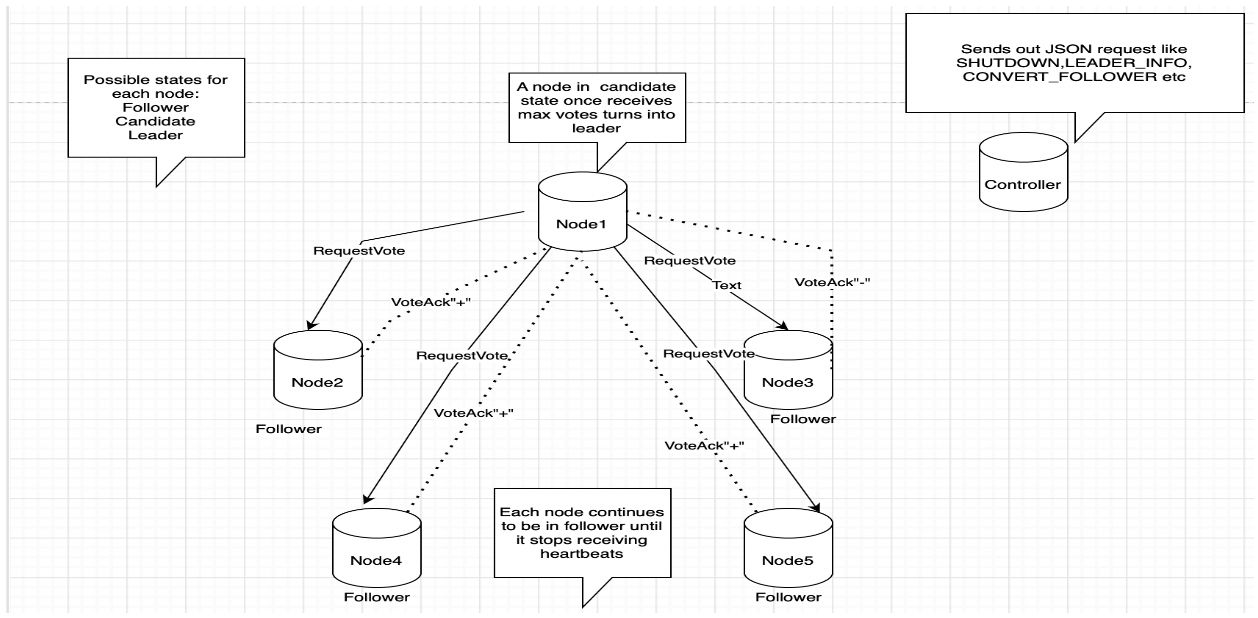
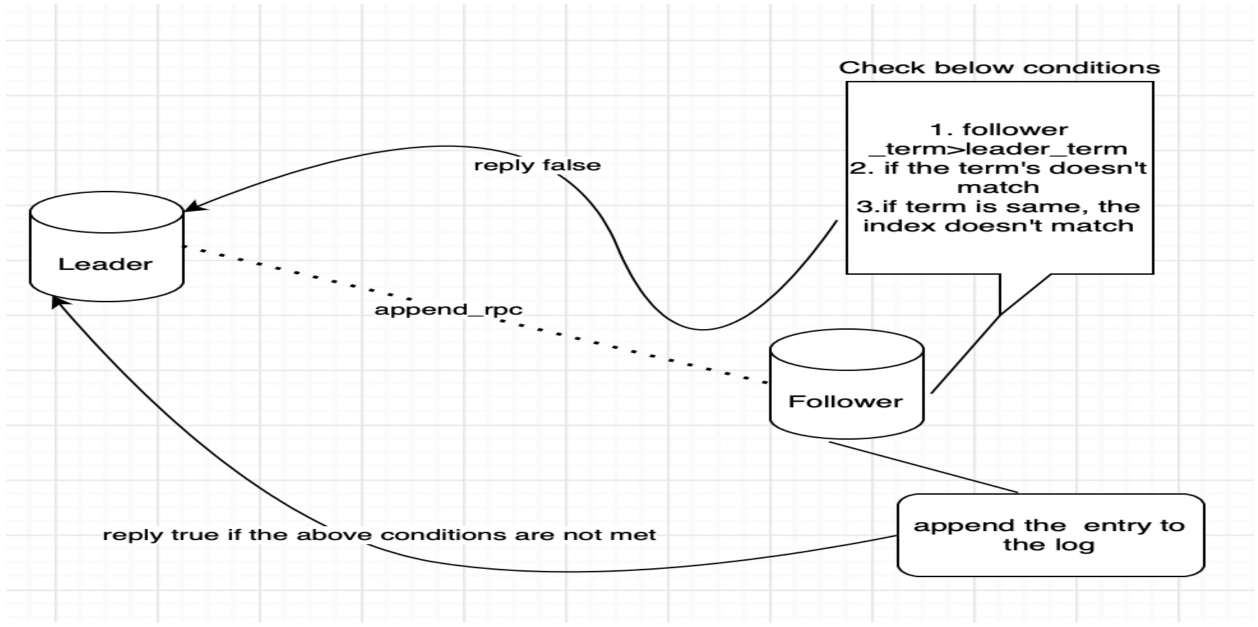
CSE4/586-Distributed Systems
Phase-4 Project Report

Introduction:

The objective of Phase-4 of the project is to extend earlier implementation of a distributed consensus algorithm called "RAFT" . The specific task in this phase is to implement the log replication part of the algorithm and also to improve the leader election through consistency check. In this report we are including our implementation of the design and a few working screenshots of the completed task.

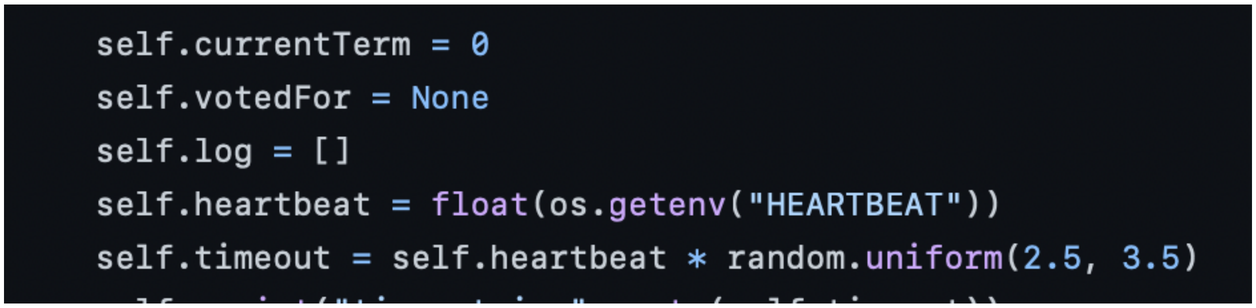
Design Overview:


Possible states for each node: Follower Candidate Leader
A node in candidate state once receives max votes turns into leader
Sends out JSON request like SHUTDOWN,LEADER_INFO, CONVERT_FOLLOWER etc
Controller
Node1
RequestVote
VoteAck"+"
RequestVote
Text
VoteAck"-"
Node2
Follower
RequestVote
RequestVote
Node3
Follower
VoteAck"+"
VoteAck"+"
Node4
Follower
Each node continues to be in follower until it stops receiving heartbeats
Node5
Follower


Check below conditions
1. follower _term>leader_term
2. if the term's doesn't match
3.if term is same, the index doesn't match
reply false
Leader
append_rpc
Follower
append the entry to the log
reply true if the above conditions are not met

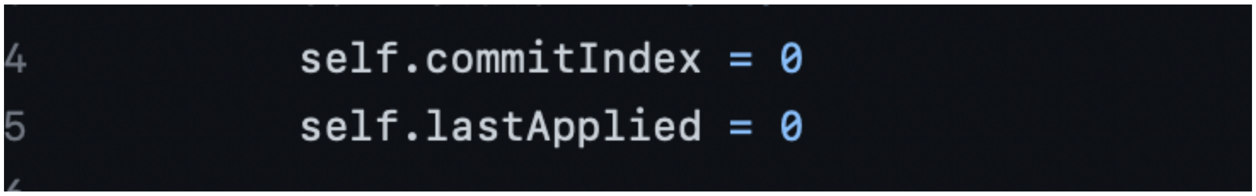
Implementation:

In this phase of the project we will be having five nodes for RAFT implementation with each node containing one docker container. For each node the same node.py code is copied using docker compose file, heart beat is set for 100ms as environment variable in the same file. Coming to the node.py implementation a class named Node is defined in which all the attributes that need to be stored by each node are defined as follows:



```
self.currentTerm = 0
self.votedFor = None
self.log = []
self.heartbeat = float(os.getenv("HEARTBEAT"))
self.timeout = self.heartbeat * random.uniform(2.5, 3.5)
```

Timeout interval is chosen randomly between 2.5&3.5 and multiplied to the heartbeat set in the docker compose file. In addition to the above attributes we have defined the below additional attributes for checking log consistency.

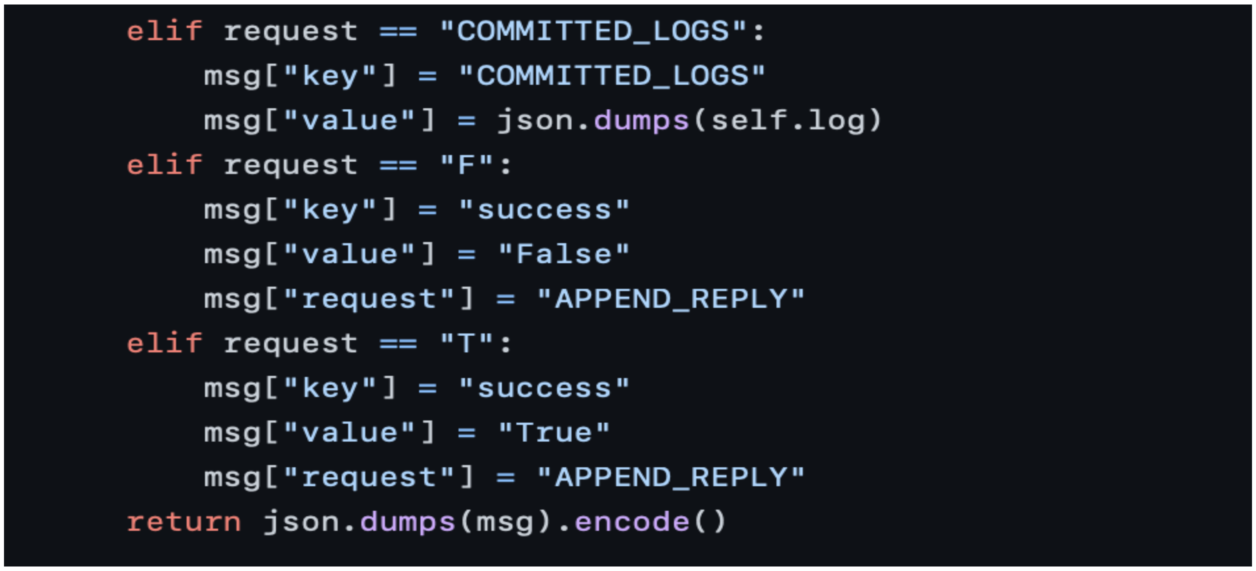


```
        self.commitIndex = 0
        self.lastApplied = 0
```

Descriptions of the key functions defined in phase-3 and modifications done in phase-4:

def generateMessage: This is a common function that will be called depending on the kind of the message that needs to be passed among nodes. The kind of message is set using the request attribute in the message structure. Some of the possible values for request are APPEND_RPC, LEADER_INFO,VOTE_REQUEST,VOTE_ACK etc

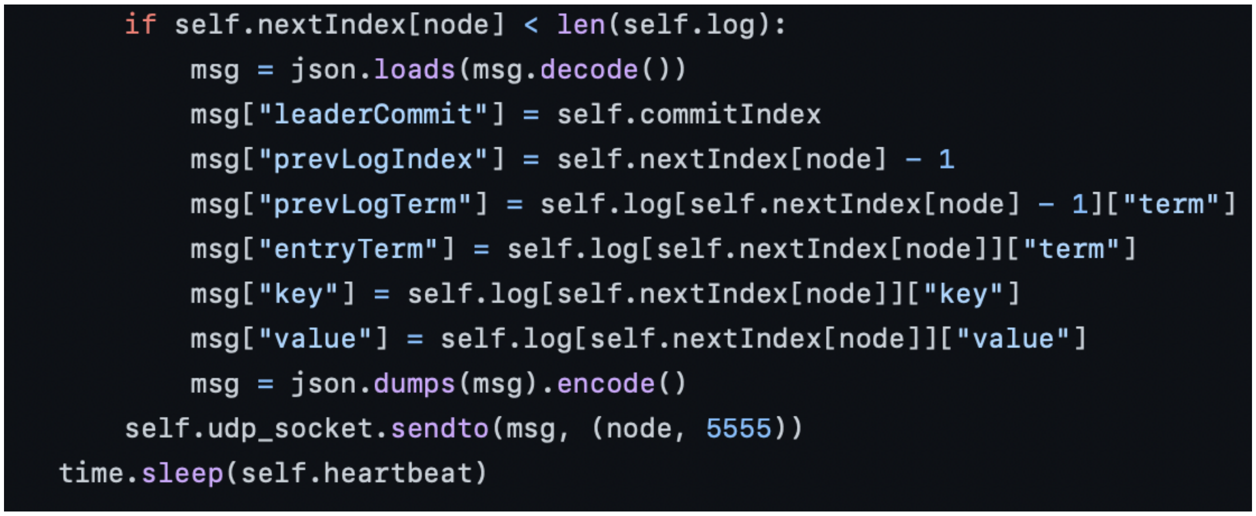
In **phase-4** the generateMessage function is modified to accommodate additional message requests that need to be sent out during append_reply function call and for committed logs.



```
        elif request == "COMMITTED_LOGS":
            msg["key"] = "COMMITTED_LOGS"
            msg["value"] = json.dumps(self.log)
        elif request == "F":
            msg["key"] = "success"
            msg["value"] = "False"
            msg["request"] = "APPEND_REPLY"
        elif request == "T":
            msg["key"] = "success"
            msg["value"] = "True"
            msg["request"] = "APPEND_REPLY"
        return json.dumps(msg).encode()
```

def leaderSender: Whenever a particular node becomes leader this function will be used to send APPEND_RPC requests to other nodes to establish its authority and to send periodic heart beats.

In earlier append_rpc's were just functioning as heartbeat messages but in **phase-4** additional attributes like prevLogIndex, prevLogTerm,commitIndex etc are needed to be included in the message.



```
        if self.nextIndex[node] < len(self.log):
            msg = json.loads(msg.decode())
            msg["leaderCommit"] = self.commitIndex
            msg["prevLogIndex"] = self.nextIndex[node] - 1
            msg["prevLogTerm"] = self.log[self.nextIndex[node] - 1]["term"]
            msg["entryTerm"] = self.log[self.nextIndex[node]]["term"]
            msg["key"] = self.log[self.nextIndex[node]]["key"]
            msg["value"] = self.log[self.nextIndex[node]]["value"]
            msg = json.dumps(msg).encode()
        self.udp_socket.sendto(msg, (node, 5555))
    time.sleep(self.heartbeat)
```

def candiateSender: This function will be used by a node that has transitioned to candidate state from follower state to request votes from other nodes.

def sendVote: This function as the name suggests is for sending a response to a REQUEST_VOTE message from the candidate node.

def convertLeader,convertFollwer,convertCandiate: These three functions are for changing between states of the node i.e follower, candidate and leader.

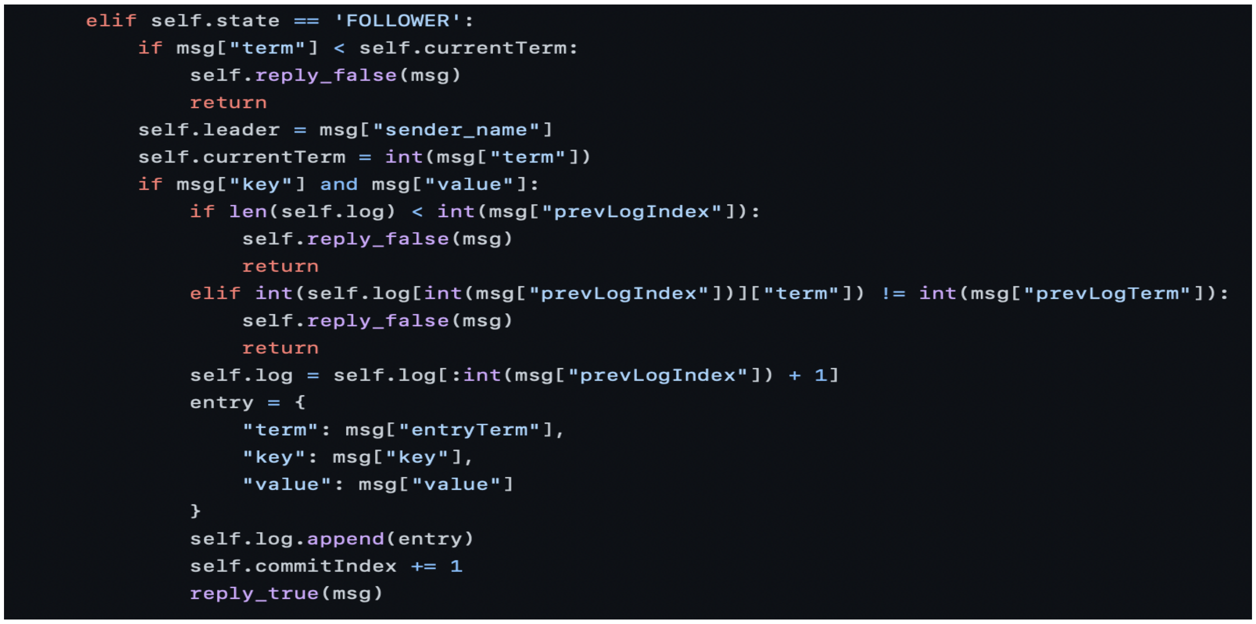
In the converLeader function of **phase-4** a slight modification is made to accommodate the change that a node once it becomes leader should initialize nextIndex value for the other nodes.

def threadlesslisterner: This function acts as a listener thread waiting for the message to be captured from the other node and invokes corresponding function based on the request received. This function executes only after checking if the self.alive attribute is true or not. If the attribute is false then the node stops listening. This attribute comes handy while implementing node shut down.

In **phase-4** additional implementation that is required for handling STORE and RETRIEVE messages has been added.

def append_rpc: This function is used for sending append RPC requests. Depending upon the state of the node specific functionality is defined. If the state of the node is leader it does nothing as the leader is the one that sends out append_rpc. If the state of the node is candidate and the candidate receives an append_rpc message it means that leader election is completed and some other node has become the leader, so the candidate converts back to follower state.

In **phase-4** of the project modifications are made to implement the below functionality if the state of the node is follower. The follower node has to reply back with a message whether it has accepted the append_rpc or not. The node will reply false if the follower current term is greater than the term indicated in the append_rpc message or if the index of the follower's log and prevLogIndex are not matching which means follower's logs are more updated.



```
        elif self.state == 'FOLLOWER':
            if msg["term"] < self.currentTerm:
                self.reply_false(msg)
                return
            self.leader = msg["sender_name"]
            self.currentTerm = int(msg["term"])
            if msg["key"] and msg["value"]:
                if len(self.log) < int(msg["prevLogIndex"]):
                    self.reply_false(msg)
                    return
                elif int(self.log[int(msg["prevLogIndex"])]["term"]) != int(msg["prevLogTerm"]):
                    self.reply_false(msg)
                    return
                self.log = self.log[:int(msg["prevLogIndex"]) + 1]
                entry = {
                    "term": msg["entryTerm"],
                    "key": msg["key"],
                    "value": msg["value"]
                }
                self.log.append(entry)
                self.commitIndex += 1
                reply_true(msg)
```

If the conditions for false criteria are not met then the follower node should accept the log entry and append the entry to its own log and reply with a true message.

def vote_ack: This function defines how a node must respond once it receives an acknowledgement for vote request .If the node has received a positive response from the other node then the vote count is increased by 1. Then overall vote count is checked and if it is a majority then the convertLeader function is called and the state is changed to leader.

def timeout: This function just calls another function named convertCandiate which changes the state of the node to Candidate if the current is follower node and it times out before receiving a heart beat from the leader.

Additional functions defined during phase-4 implementation:

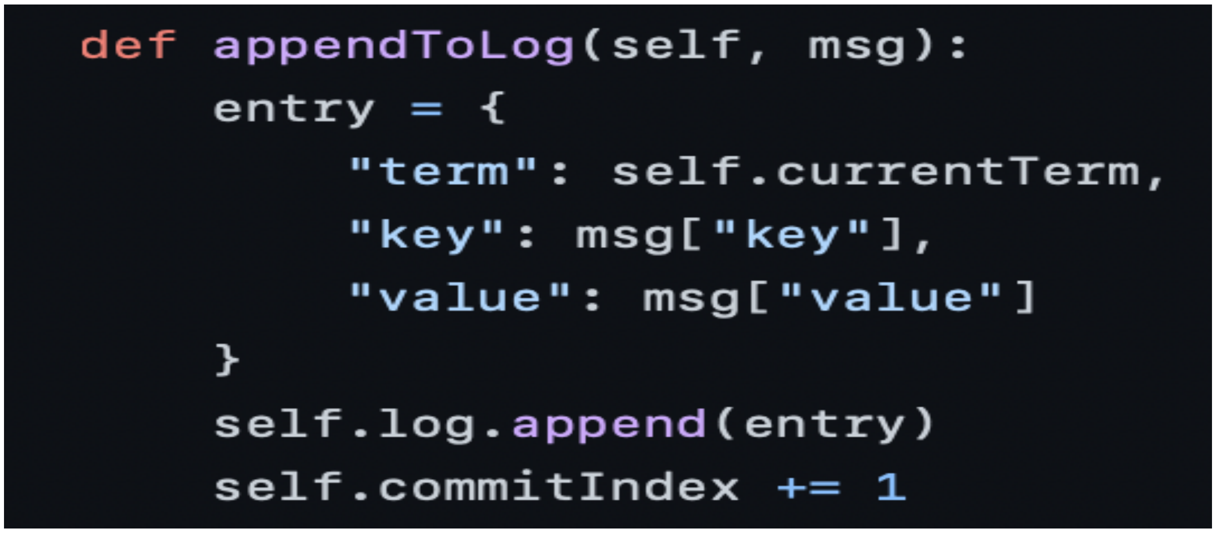
def reply_false, def reply_true: When a follower node receives an append_rpc message from the leader it can either accept the log entry or reject the log entry. Corresponding reply messages true if log entry has been accepted else false if rejected will be handled by these functions.

def append_reply: This function defines what a leader node should do depending on whether the follower node has accepted or rejected the log entry. If the follower node accepts the log entry then leader should increment nextIndex by 1 for that particular node else decrement it by 1

def store: This function defines how a node should respond on receiving a client request/command which in this case is to STORE. In a cluster only the leader should add the client's request as entry to its own log. If any other node apart from the leader receives the request then it should return the leader info for the cluster.

def retrieve: Similar to the def store function only leader should be able to retrieve the log info and return it. If any other node receives the RETRIEVE request it should return the leader info for the cluster.

def appendToLog: This function as the name suggests adds a log entry to the self.log attribute which is a list data type defined for every node. Also after adding the entry to the log it increments the commitIndex value by 1.



```python
def appendToLog(self, msg):
    entry = {
        "term": self.currentTerm,
        "key": msg["key"],
        "value": msg["value"]
    }
    self.log.append(entry)
    self.commitIndex += 1
```

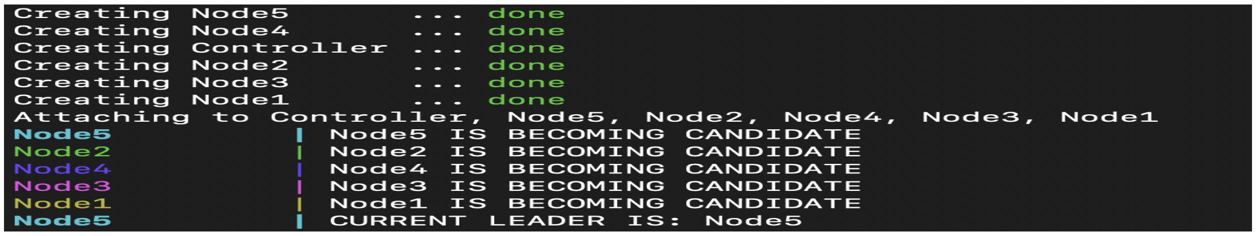
Distinction between basic log replication in phase-2 and phase-4:
The basic difference between log replication in both phases is that, in phase-2 there was no consensus involved once a client request is submitted it will be passed on to the other nodes and gets executed in all of them.

But in this phase once a client request is received only the leader should respond to the client with the output and also the client request is not immediately executed and a log entry is maintained on all the nodes.. Firstly, once the leader receives a client request it appends the request to its own log and the leader tries to replicate the log entry to other follower nodes through append_rpc call. Once it has been replicated to the majority of the server then the leader node executes the request and returns the result to the user. This is way safe log replication and consensus is achieved.

How to Run: While in the directory containing the docker-compose.yml file, simply run "docker-compose build" followed by "docker-compose up".

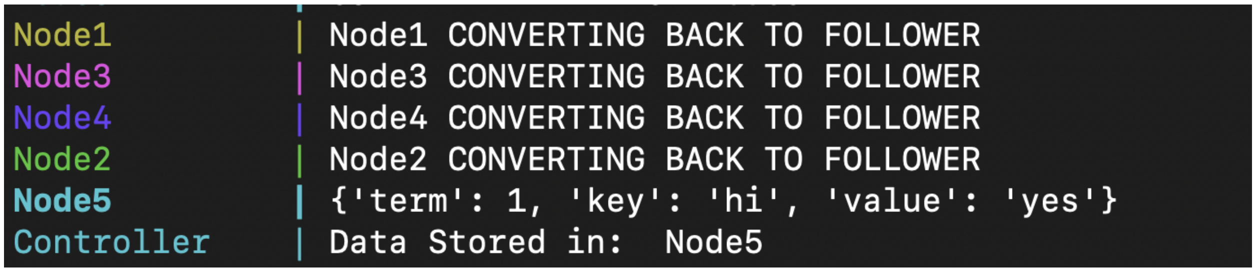
Validation:

Pic-1: Nodes, once initialized, begin an election and decide which node to make the leader and in this case Node 5 has become the leader.



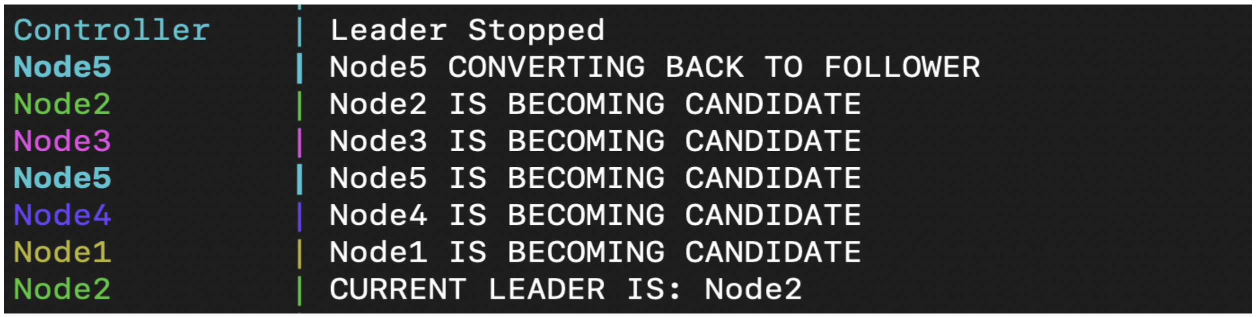
```
Creating Node5         ... done
Creating Node4         ... done
Creating Controller    ... done
Creating Node2         ... done
Creating Node3         ... done
Creating Node1         ... done
Attaching to Controller, Node5, Node2, Node4, Node3, Node1
Node5            | Node5 IS BECOMING CANDIDATE
Node2            | Node2 IS BECOMING CANDIDATE
Node4            | Node4 IS BECOMING CANDIDATE
Node3            | Node3 IS BECOMING CANDIDATE
Node1            | Node1 IS BECOMING CANDIDATE
Node5            | CURRENT LEADER IS: Node5
```

Pic-2: Once the leader is elected other nodes go back to follower state and the controller issues a store request with key=hi and value=yes and we can see that log entry being displayed by current leader node 5.



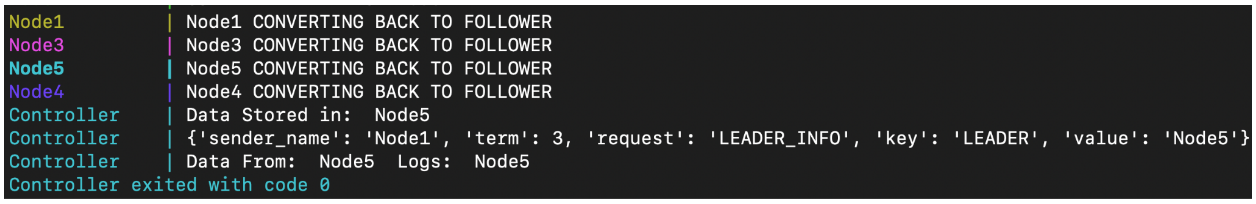
```
Node1           | Node1 CONVERTING BACK TO FOLLOWER
Node3           | Node3 CONVERTING BACK TO FOLLOWER
Node4           | Node4 CONVERTING BACK TO FOLLOWER
Node2           | Node2 CONVERTING BACK TO FOLLOWER
Node5           | {'term': 1, 'key': 'hi', 'value': 'yes'}
Controller      | Data Stored in:  Node5
```

Pic3: Leader node is forced to stop. In this case Node 5 is stopped. It forces a new election and new Node 2 is elected as leader.



```
Controller      | Leader Stopped
Node5           | Node5 CONVERTING BACK TO FOLLOWER
Node2           | Node2 IS BECOMING CANDIDATE
Node3           | Node3 IS BECOMING CANDIDATE
Node5           | Node5 IS BECOMING CANDIDATE
Node4           | Node4 IS BECOMING CANDIDATE
Node1           | Node1 IS BECOMING CANDIDATE
Node2           | CURRENT LEADER IS: Node2
```

Pic4: Now retrieve request is issued by the controller and node5 is again restarted which becomes the leader again and displays the logs.



```
Node1           | Node1 CONVERTING BACK TO FOLLOWER
Node3           | Node3 CONVERTING BACK TO FOLLOWER
Node5           | Node5 CONVERTING BACK TO FOLLOWER
Node4           | Node4 CONVERTING BACK TO FOLLOWER
Controller      | Data Stored in:  Node5
Controller      | {'sender_name': 'Node1', 'term': 3, 'request': 'LEADER_INFO', 'key': 'LEADER', 'value': 'Node5'}
Controller      | Data From:  Node5  Logs:  Node5
Controller exited with code 0
```

References:

-Flask documentation: https://flask.palletsprojects.com/en/2.0.x/

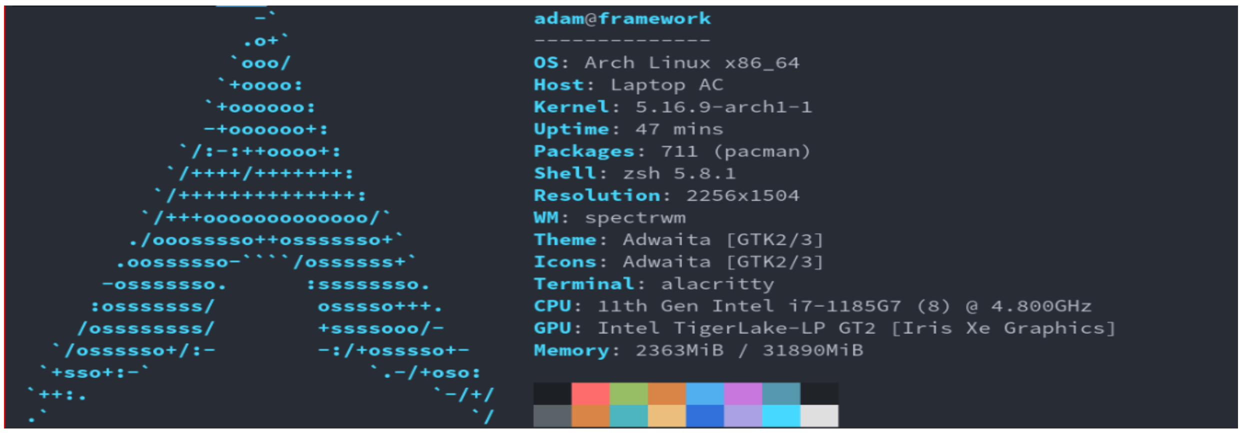
-Docker: https://docs.docker.com/get-started/

-RAFT: https://raft.github.io/

-And material provided on UBLearns

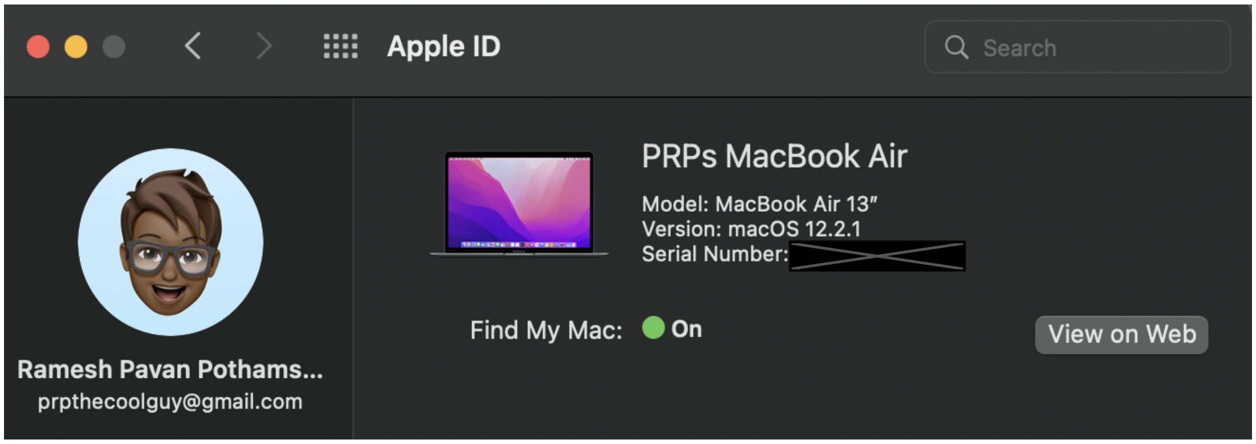
Proof Of Laptop Ownership:

Adam:



```
                   -`                    adam@framework
                  .o+`                   --------------
                 `ooo/                   OS: Arch Linux x86_64
                `+oooo:                  Host: Laptop AC
               `+oooooo:                 Kernel: 5.16.9-arch1-1
               -+oooooo+:                Uptime: 47 mins
             `/:-:++oooo+:               Packages: 711 (pacman)
            `/++++/+++++++:              Shell: zsh 5.8.1
           `/++++++++++++++:             Resolution: 2256x1504
          `/+++ooooooooooooo/`           WM: spectrwm
         ./ooosssso++osssssso+`          Theme: Adwaita [GTK2/3]
        .oossssso-````/ossssss+`         Icons: Adwaita [GTK2/3]
       -osssssso.      :ssssssso.        Terminal: alacritty
      :osssssss/        osssso+++.       CPU: 11th Gen Intel i7-1185G7 (8) @ 4.800GHz
     /ossssssss/        +ssssooo/-       GPU: Intel TigerLake-LP GT2 [Iris Xe Graphics]
   `/ossssso+/:-        -:/+osssso+-     Memory: 2363MiB / 31890MiB
  `+sso+:-`                 `.-/+oso:
 `++:.                           `-/+/
 .`                                 `/
```

Ramesh Pavan:



PRPs MacBook Air

Model: MacBook Air 13"
Version: macOS 12.2.1
Serial Number: [redacted]

Find My Mac: ● On

Ramesh Pavan Pothams...
prpthecoolguy@gmail.com