Phase 4 Report

Distributed Systems, Spring 2022

Imani Muhammad-Graham

Table of Contents

Introduction	....	3
Design Overview	....	3
Application	....	7
Implementation	....	7
Validation	....	11
References	....	13

Introduction

Project

The subject of interest of this topic is communication, featuring a "motivational quote" web application. The app allows users to post a quote that all other users can see. This report corresponds to Phase 4 of the Distributed Systems project.

Phase 4

The motivation behind this Phase was to complete the implementation of the RAFT Consensus Algorithm that was started in Phase 3. Phase 4 incorporates the application designed in Phase 1&2 and the distributed cluster designed in Phase 3. The cluster would be tested and sampled with a controller. This phase entailed:

- Evolving Phase 2's Application to a 5-node connected distributed system
- Incorporating Safe log replication into Phase 3's design

The completion of Phase 2 required a demonstration of the following knowledge/skills:

- Network Design
- RAFT Consensus Algorithm
- Web Development

Design Overview

The system's design consists of multiple nodes working together to communicate with a client. Each node contains a server and database running in separate docker containers. The nodes communicate over a network instantiated by docker, and the server communicates with a client via web socket connections.

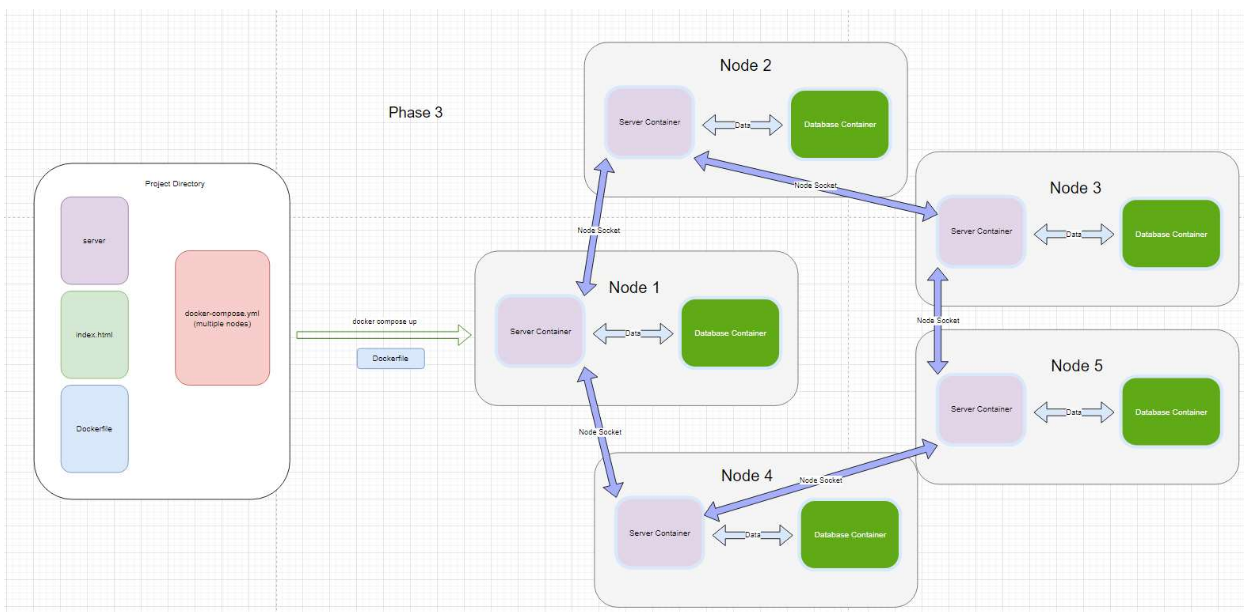


Figure 1: Model of system design (Part 1)

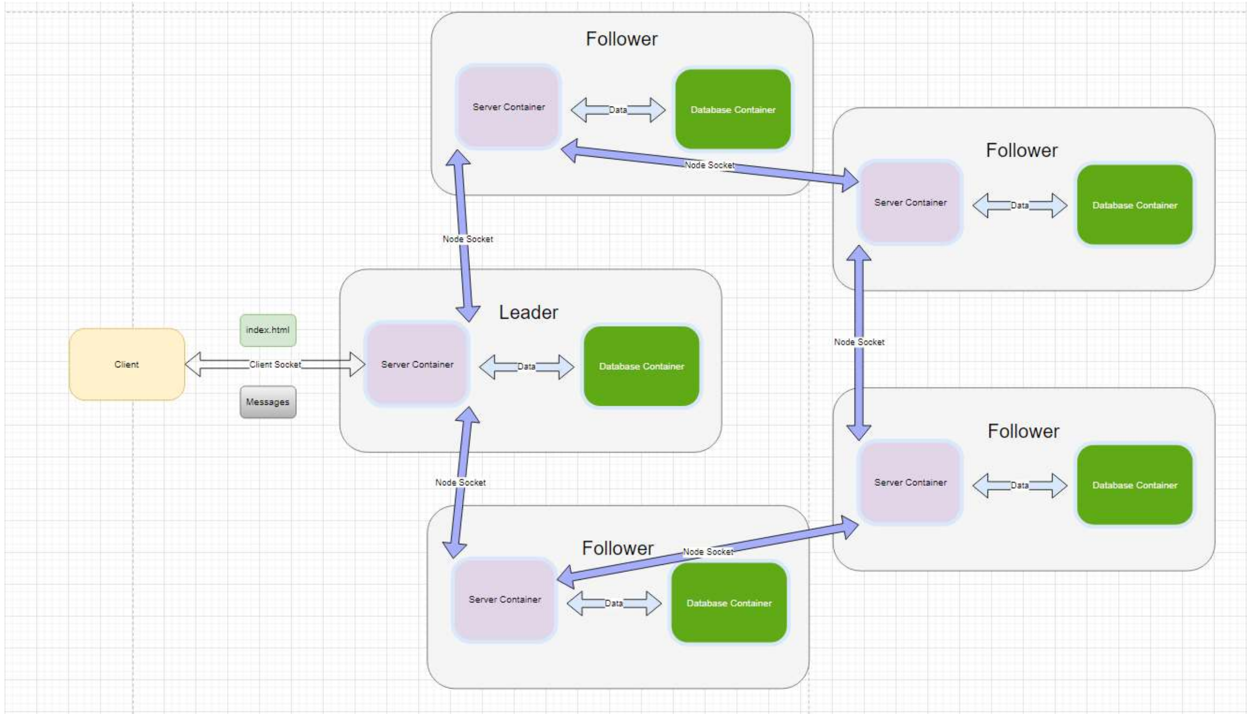


Figure 1: Model of system design (Part 2)

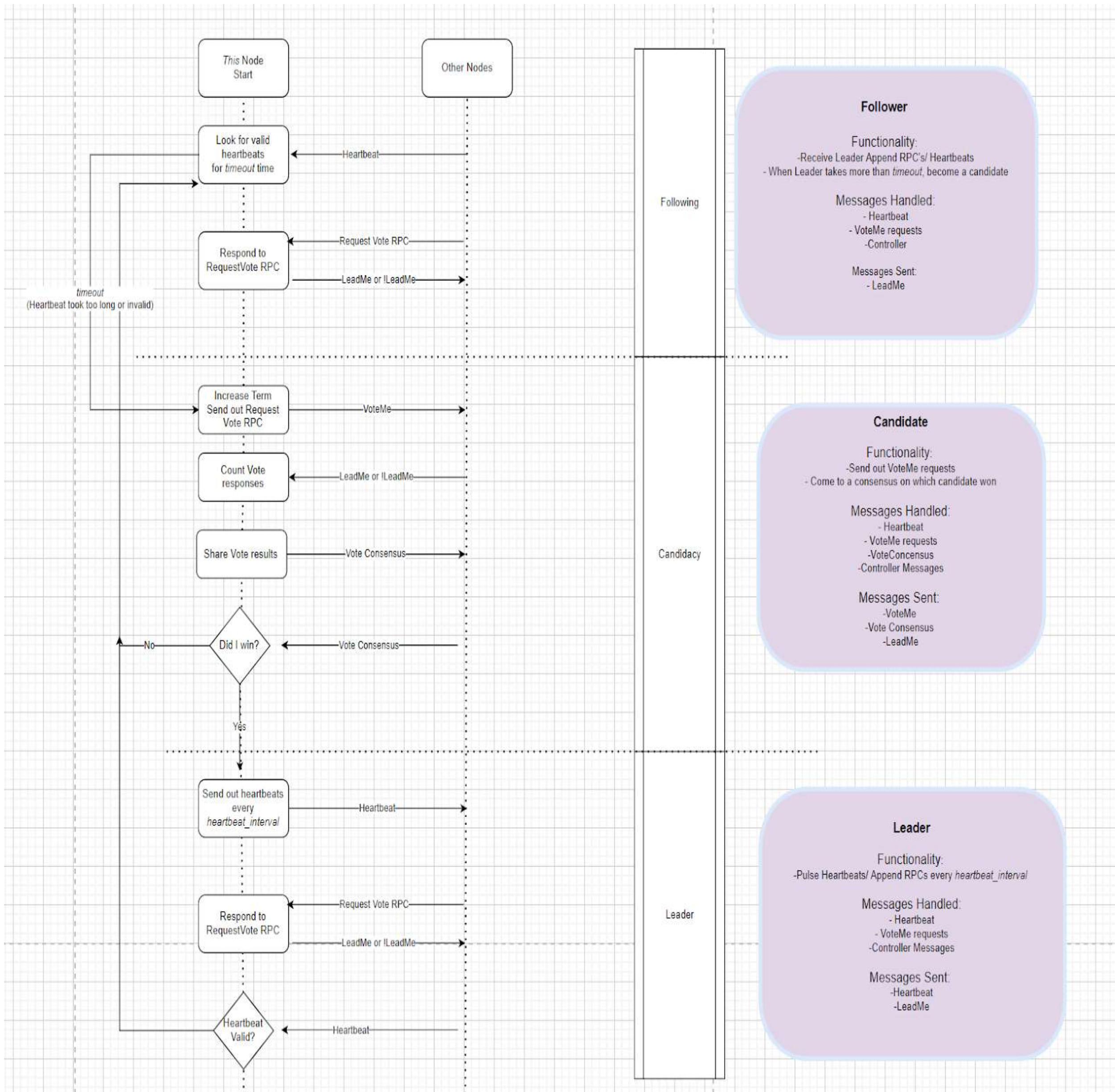


Figure 2: Functionality Overview (part 1)

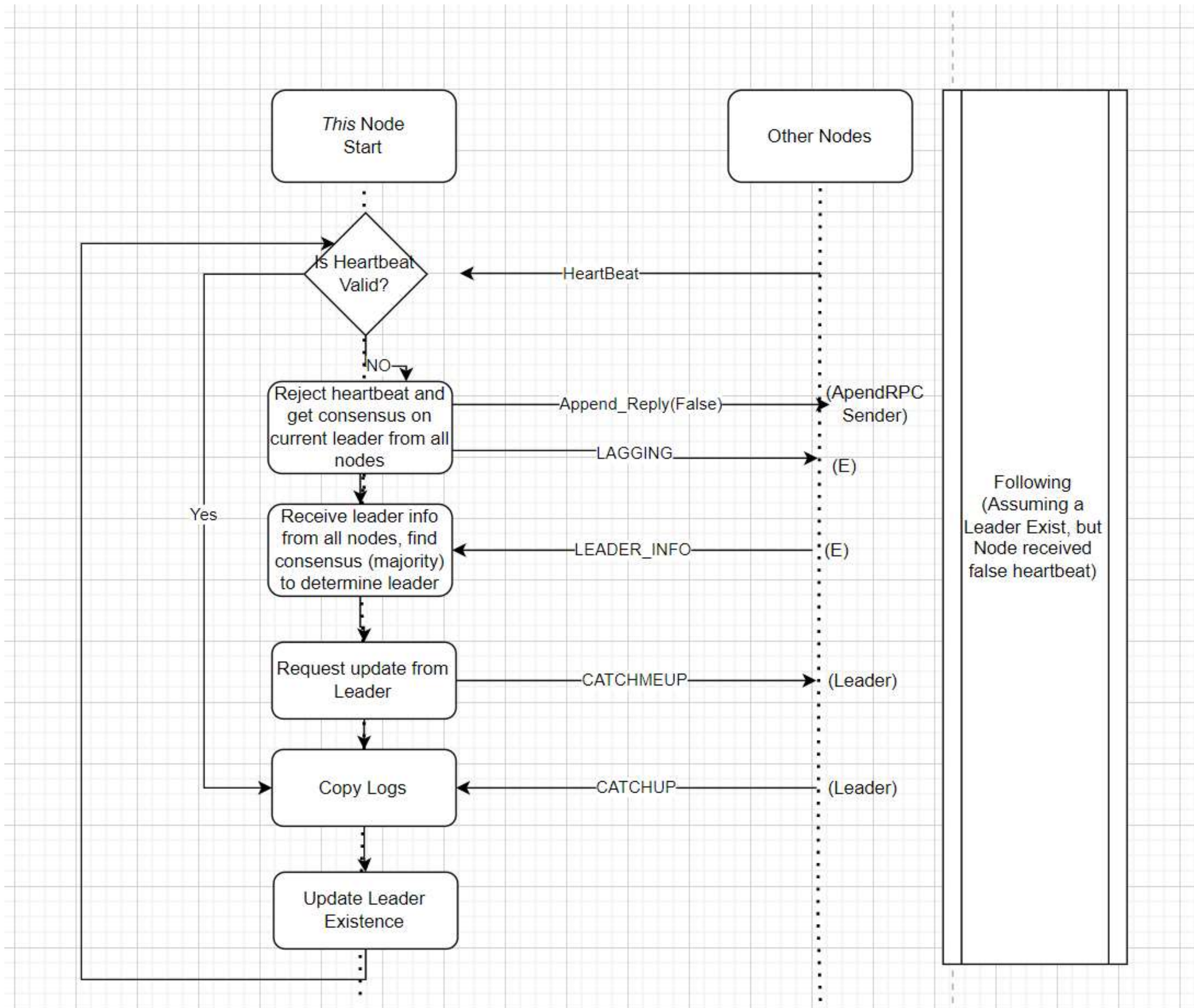


Figure 2: Functionality Overview (part 2)

This message architecture for Lagging nodes was chosen over ‘leader’s netindex’-via project description- because it allows for the Lagging node to get a consensus of who the leader is before trusting the leader that sent an AppendRPC. In the case that the Lagging nodes’ own logs are corrupted, the logs won’t be accepted from the leader, so CATCHUP contains all of the leaders’ logs instead of one log at a time.

Application

The Application

This phase involved adding safe log replication to the consensus algorithm designed in Phase 3, as well as incorporating Phase 1's web application. The result is a simple RAFT-based cloud hosted web application. The following tools were used to develop the application:

- Websockets
- Flask.io
- Javascript
- Python

Docker

To deploy the app in its own container, 'docker-compose.yml' and 'Dockerfile' files were used. The docker-compose file sets up the networks, containers and the libraries and dependencies needed to run The Application, while the Dockerfile builds the container image.

Implementation

The Application

Safe log replication amongst the cluster and the web application were the two things implemented in Phase 4. For the Leader functionality, hosting of the web application and broadcasting AppendRPC's with Log information was added. Naturally, each nodes' message handler was upgraded to receive, judge, and append logs from the leader. Each state sent, and processed messages differently based on their roles, as seen in the figures below. Every node, no matter the state, had a listener running at all times. There is also a handler for messages from a Controller which is intended to change the state of the system with different commands including: SHUTDOWN, FOLLOW, TRYLEAD, STATUS, STORE, and RETRIEVE.

```
#node recieving command from controller
elif dm['sender_name'] == "Controller":
    #UPDATE
    if Request == 'STATUS':···
    #turn node into a follower
    elif Request == 'FOLLOW':···
    #turn node into a candidate
    elif Request == 'TRYLEAD':···
    #have node play dead
    elif Request == 'PLAYDEAD':···
    #Add something to the log
    elif Request == 'STORE':···
    #Retrieve Logs
    elif Request == 'RETRIEVE':···
#node recieving message from other node
elif node_info['state'] != 'd':
    #follower recieving a candidate's vote request
    if Request== "VOTEME":···
    #candidate recieving a follower's vote respopnse
    elif Request == "LEADME":···
    #candidate recieving a follower's vote respopnse
    elif Request == "!LEADME":···
    #follower recieving a leader's heartbeat
    elif Request == "HEARTBEAT":···
    #Node recieving message from Lagging Node
    elif Request == "LAGGING":···
    #From node to Lagging Node
    elif Request == "LEADER_INFO" and dm['recipient'] == name:···
    #Lagging node to Leader
    elif Request == "CATCHMEUP":···
    #Leader trying to CATCHUP Lagging Node
    elif Request == "CATCHUP":···
    #Candidates checking for new leader
    elif Request == "VOTECONCENSUS":···
    #Append_Reply from followers to leader
    elif Request == "Append_Reply":···
    #Bad Message
    else:···
#node is dead
else:···
```

Figure 3: Message Handler: The conditionals seen above in message_handler() express the different types of messages handled from the controller or other nodes

```
        elif Request == "HEARTBEAT":#verify heartbeat
            #If node is lagging with respects to logs
            if dm['prev_log_index'] > len(node_info['log']):#Send false success, broadcast Lagging ⋯
            #Valid Heartbeat, caught up with logs
            elif dm['term'] >= node_info['term'] and node_info['log'][dm['prev_log_index']]['term'] <= dm['prev_log_term']:#copy log⋯
            #invalid heartbeat
            else:#send failure RPC    ⋯
        #Node recieving message from Lagging Node
        elif Request == "LAGGING":#send LEADER_INFO to lagging node⋯
        #From node to Lagging Node
        elif Request == "LEADER_INFO" and dm['recipient'] == name:#Get concensus from nodes on leader⋯
        #Lagging node to Leader
        elif Request == "CATCHMEUP": #Leader sending CATCHUP to lagger⋯
        #Leader trying to CATCHUP Lagging Node
        elif Request == "CATCHUP":  #Lagger recieving CATCHUP from leader⋯
        #Candidates checking for new leader
```

Figure 4: Heartbeat handling

This follows figure 2’s description of heartbeat sending and handling

Docker

The dockerfiles below built the image for the containers

Phase_4 > RAFTBABY > Node > Dockerfile > FROM

```
FROM python:3.7-alpine3.14

RUN pip install flask
RUN pip install flask-socketio
# Stop the cache
ADD https://www.google.com /time.now

COPY . .

EXPOSE 5555

ENTRYPOINT ["python", "-u", "node.py"]
```

Phase_4 > RAFTBABY > Controller > Dockerfile > FROM

```
FROM python:3.7-alpine3.14

COPY . .

EXPOSE 5555

ENTRYPOINT ["python", "-u", "CNTRL.py"]
```

Figure 5: Dockerfile

Installs dependencies and runs server and controller containers. Google.com/time.now was to force new builds every time the container was run.

The docker-compose file below sets contexts and builds the server and controller containers for each node.

Phase_4 > RAFTBABY > docker-compose.yml

```yaml

version: "3.7"
services:
    node1: ···
    node2: ···
    node3: ···
    node4: ···
    node5:
      container_name: Node5
      build: Node/.
      environment:
        - Port=5555
        - app_name=Node5
        - group=224.1.1.1
        - tor1=5
        - tor2=10
        - heartrate=.25
        - client_host=220.1.1.1
        - client_port=5000

    controller:
      container_name: Controller
      build: Controller/.
      environment:
        - Port=5555
        - app_name=Controller
        - group=224.1.1.1
      stdin_open: true
      depends_on:
        node1:
          condition:  service_started
        node2: ···
        node3: ···
        node4: ···
        node5: ···
```

Figure 6: Docker-compose
Sets contexts, images, builds containers, defines port usage. The 5 nodes, and Controller were defined as such.
The command to run the application is:
docker compose up —build

Validation

Cluster up and running

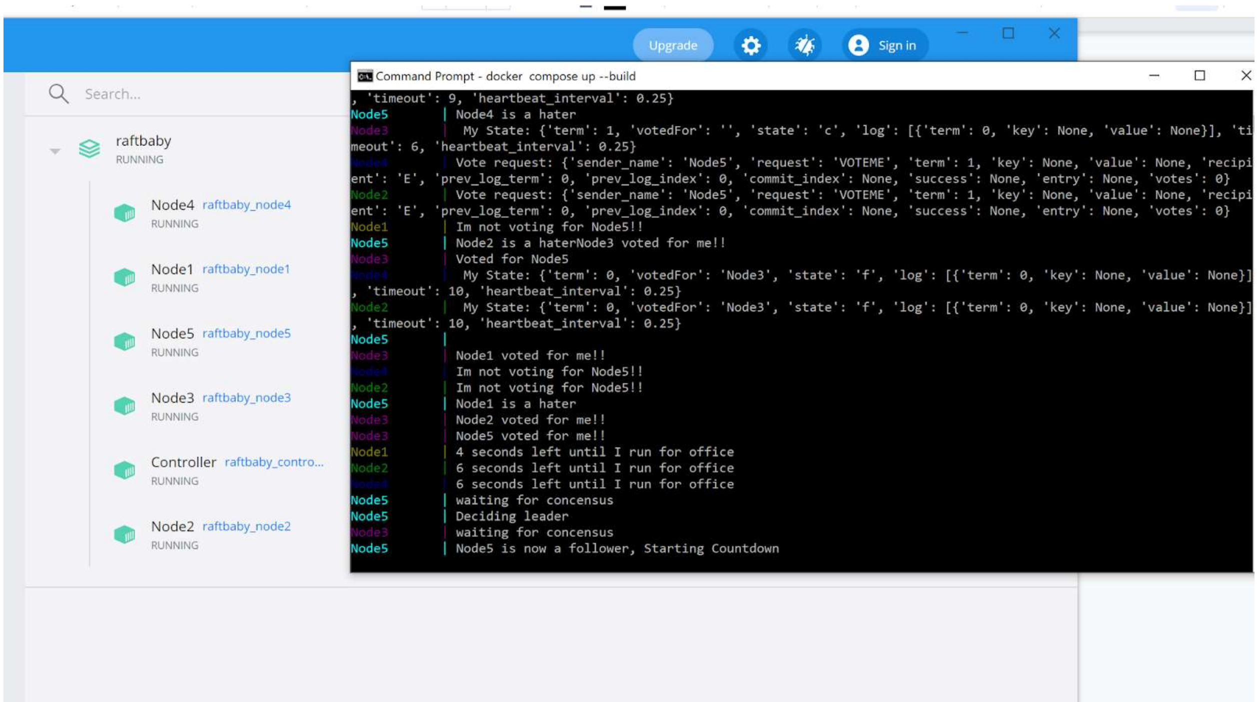


Figure 7: System running

Docker compose up –build was ran and created node containers. The nodes each started as a follower and were given a random timeout value, went into candidacy and are voting on the next leader.

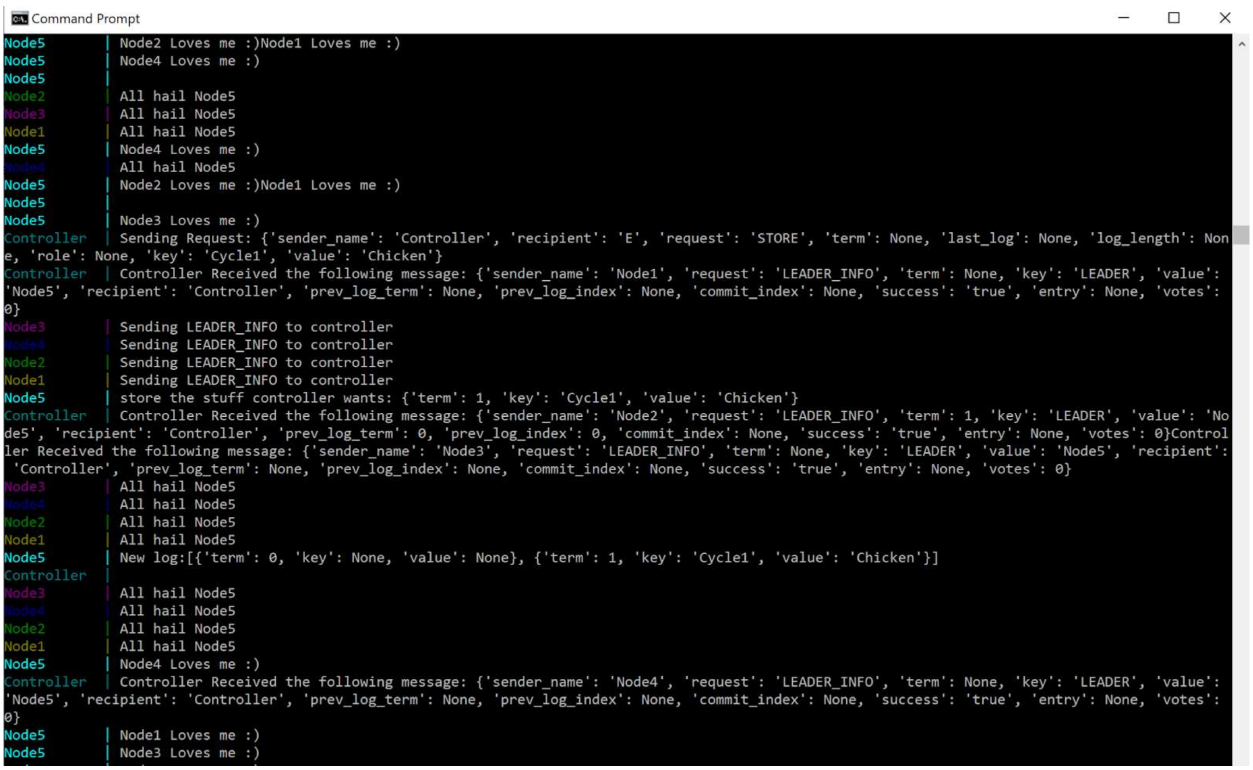
```python
        elif Request == 'STORE':
            if node_info['state'] == 'l':#store what controller wants
                storethis = {'term':node_info['term'],'key':dm['key'],'value':dm['value']}
                print(f"store the stuff controller wants: {storethis}")
                node_info['log'].append(storethis)
                #broadcasting appendrpc
                msg_c['request'] = 'HEARTBEAT'
                msg_c['recipient'] = 'E'
                msg_c['term'] = node_info['term']
                msg_c['prev_log_term'] = node_info['log'][len(node_info['log'])-2]['term']
                msg_c['prev_log_term'] = len(node_info['log'])-2
                msg_c['commit_index'] = len(node_info['log'])
                msg_c['entry'] = storethis
                send_message(msg_c,group,socket,port)
                print(f"New log:{node_info['log']}")
            else:#send leader_info to controller
                print("Sending LEADER_INFO to controller")
                msg_c['request']= 'LEADER_INFO'
                msg_c['recipient'] = 'Controller'
                msg_c['key'] = "LEADER"
                msg_c['value'] = current_leader
                send_message(msg_c,group,socket,port)
```

Figure 8: Log replication (Part 1: Node STORE functionality)

```
Controller raftbaby_controller
EXITED (137)
LOGS INSPECT STATS
Sending Request: {'sender_name': 'Controller', 'recipient': 'E', 'request': 'STORE', 'term': None, 'last_log': None, 'log_length': None, 'role': None, 'key': 'Cycle1', 'value': 'Chicken'}
Controller Received the following message: {'sender_name': 'Node1', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}
Controller Received the following message: {'sender_name': 'Node2', 'request': 'LEADER_INFO', 'term': 1, 'key': 'LEADER', 'value': 'Node5', 'recipient': 'Controller', 'prev_log_term': 0, 'prev_log_index': 0, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}Controller Received the following message: {'sender_name': 'Node3', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}

Controller Received the following message: {'sender_name': 'Node4', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}
```

Figure 8: Log replication (Part 2: STORE)



```
Command Prompt
Node5      | Node2 Loves me :)Node1 Loves me :)
Node5      | Node4 Loves me :)
Node5      |
Node2      | All hail Node5
Node3      | All hail Node5
Node1      | All hail Node5
Node5      | Node4 Loves me :)
Node4      | All hail Node5
Node5      | Node2 Loves me :)Node1 Loves me :)
Node5      |
Node5      | Node3 Loves me :)
Controller | Sending Request: {'sender_name': 'Controller', 'recipient': 'E', 'request': 'STORE', 'term': None, 'last_log': None, 'log_length': None, 'role': None, 'key': 'Cycle1', 'value': 'Chicken'}
Controller | Controller Received the following message: {'sender_name': 'Node1', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}
Node3      | Sending LEADER_INFO to controller
Node4      | Sending LEADER_INFO to controller
Node2      | Sending LEADER_INFO to controller
Node1      | Sending LEADER_INFO to controller
Node5      | store the stuff controller wants: {'term': 1, 'key': 'Cycle1', 'value': 'Chicken'}
Controller | Controller Received the following message: {'sender_name': 'Node2', 'request': 'LEADER_INFO', 'term': 1, 'key': 'LEADER', 'value': 'Node5', 'recipient': 'Controller', 'prev_log_term': 0, 'prev_log_index': 0, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}Controller Received the following message: {'sender_name': 'Node3', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}
Node3      | All hail Node5
Node4      | All hail Node5
Node2      | All hail Node5
Node1      | All hail Node5
Node5      | New log:[{'term': 0, 'key': None, 'value': None}, {'term': 1, 'key': 'Cycle1', 'value': 'Chicken'}]
Controller |
Node3      | All hail Node5
Node4      | All hail Node5
Node2      | All hail Node5
Node1      | All hail Node5
Node5      | Node4 Loves me :)
Controller | Controller Received the following message: {'sender_name': 'Node4', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}
Node5      | Node1 Loves me :)
Node5      | Node3 Loves me :)
```

Figure 8: Log replication (Part 3: Nodes react to STORE)

As you can see, the nodes have elected Node 5 as their leader. When the controller sends a STORE request, following nodes reply with Leader Information, while the Leader 'stores the stuff the controller wants'. Node 5's New Log print statement was added for testing.

```
Sending Request: {'sender_name': 'Controller', 'recipient': 'E', 'request': 'RETRIEVE', 'term': None, 'last_log': None, 'log_length': None,
 'role': None, 'key': 'Cycle1', 'value': 'Chicken'}
Controller Received the following message: {'sender_name': 'Node1', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node
5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'vote
s': 0}
Controller Received the following message: {'sender_name': 'Node3', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node
5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'vote
s': 0}Controller Received the following message: {'sender_name': 'Node4', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value':
 'Node5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None,
 'votes': 0}

Controller Received the following message: {'sender_name': 'Node5', 'request': 'RETRIEVE', 'term': 1, 'key': 'COMMITED_LOGS', 'value': [{'t
erm': 0, 'key': None, 'value': None}, {'term': 1, 'key': 'Cycle1', 'value': 'Chicken'}], 'recipient': 'Controller', 'prev_log_term': 1, 'pr
ev_log_index': 0, 'commit_index': 2, 'success': None, 'entry': None, 'votes': 0}
Controller Received the following message: {'sender_name': 'Node2', 'request': 'LEADER_INFO', 'term': 1, 'key': 'LEADER', 'value': 'Node5',
 'recipient': 'Controller', 'prev_log_term': 0, 'prev_log_index': 0, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}
Sending Request: {'sender_name': 'Controller', 'recipient': 'E', 'request': 'STORE', 'term': None, 'last_log': None, 'log_length': None, 'r
ole': None, 'key': 'Cycle3', 'value': 'Chicken'}
Controller Received the following message: {'sender_name': 'Node3', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node
5', 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'vote
s': 0}
Controller Received the following message: {'sender_name': 'Node2', 'request': 'LEADER_INFO', 'term': 1, 'key': 'LEADER', 'value': 'Node5',
 'recipient': 'Controller', 'prev_log_term': 0, 'prev_log_index': 0, 'commit_index': None, 'success': 'true', 'entry': None, 'votes': 0}Con
troller Received the following message: {'sender_name': 'Node1', 'request': 'LEADER_INFO', 'term': None, 'key': 'LEADER', 'value': 'Node5',
 'recipient': 'Controller', 'prev_log_term': None, 'prev_log_index': None, 'commit_index': None, 'success': 'true', 'entry': None, 'votes':
 0}
```

Figure 8: Log Replication (Part 4: Retrieval of Stored Messages)

See video for a better visualization of the nodes' interactions, and what happens when you kill the leader.

References:

[1]: http://www.steves-internet-guide.com/introduction-multicasting/#:~:text=Note%3A%20multicast%20uses%20UDP%20and,part%20of%20a%20multicast%20group

[2]: https://flask-socketio.readthedocs.io/en/latest/getting_started.html#initialization

[3]:https://medium.com/@abhishekchaudhary_28536/building-apps-using-flask-socketio-and-javascript-socket-io-part-1-ae448768643