Phase 4

CSE 486/586: Distributed Systems

Name: Akash Deshmukh **Student ID**: 50408237

Note: Please check Readme.txt to run the application

1. Introduction

Consensus indicates that several servers agree on the same information, which is required for designing fault-tolerant distributed systems. **RAFT protocol** helps in achieving consensus. There are two major task involved in RAFT algorithm, **Leader Election and Log replication**. In the last phase I implemented the Leader Election and in this phase I am implementing **Safe Log Replication** along with modified Leader Election algorithm.
The below flow diagram explains the log replication process.

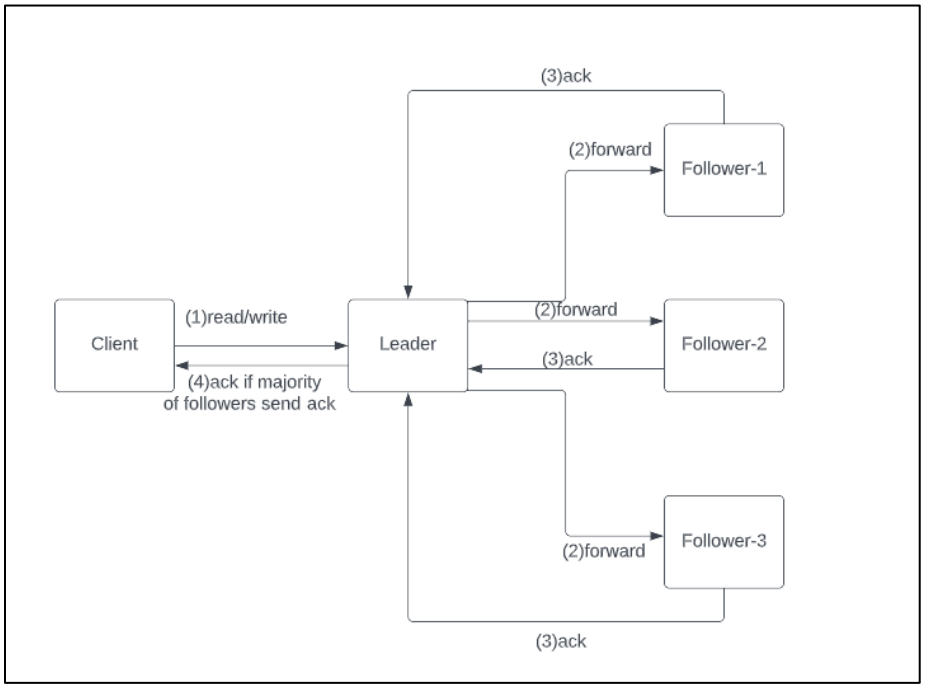


Fig 1.1 Flow diagram of LogReplication

- All read/write request goes through a single leader
- Leader appends the command to its log and sends the request to all the followers
- Leader waits for the majority to acknowledge the request, once majority of the followers sends acknowledgement then Leader acknowledge to client and commit the log

Difference in Log Replication in Phase-2 and Phase-4

- In Phase-2 the basic log replication algorithm does not involve log consistency checks.
- In Phase-2 If a node goes down for some time, some requests are made by client in that period and once the node comes up the request are not getting forwarded to that node. So that node will not have those data in its database. While in Phase-4 once the node comes up leader tries to replicate its data to the node.
- In Phase-2 a node with incomplete logs can also become a leader while in Phase4 we ensure that Nodes with outdated logs should not become leader.

- In phase-4 if a node is having extraneous entries which is not replicated to majority of the nodes will be removed by the current leader

2. Design Overview

2.1. Project Structure

- A **single** image including **client and server**
- Single image is used for creating **5 containers**
- **Client-side technology** – HTML/JSP
- **Server-side technology** – Spring(java)
- **Database** – Text File (**userInput.txt**)
- Each node information is saved in the filesystem of every node using json file. Below are the information which is saved for each node using "**nodeinfo.json**".


```
{
   "candidateID":"Node5",
   "log":[ ],
   "heartBeatInterval":50,
   "electionTimeOut":379,
   "currentTerm":5,
   "votedFor":5,
   "state":"F",
   "currentLeader":"Node3"
}
```

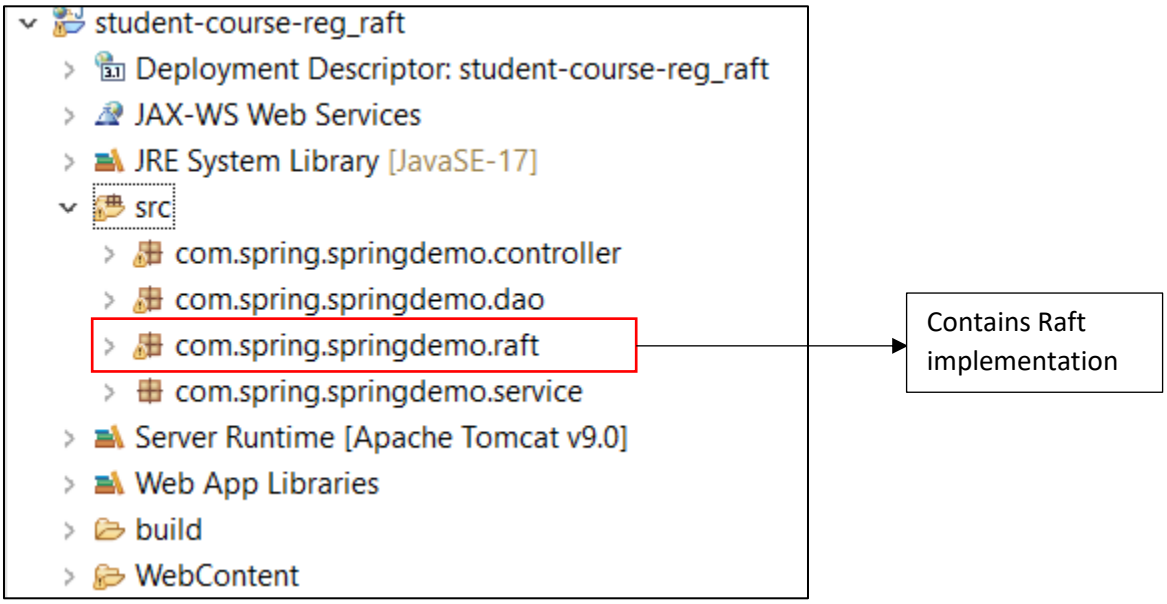



Fig 2.1 Project Structure

2.2 Architecture Details and Configurations

a. **Heartbeat –** Heartbeat which is AppendRPC sent by the leader to all the follower. The Heartbeat rate is fixed as **50 milliseconds** for each node.

b. **Election Timeout –** The timeout after which the follower becomes candidate if it does not receive heartbeat is termed as election timeout. The election timeout is getting **randomly** calculated for each node between the range **300-600 milliseconds.**

c. **AppendRPC** – Hearbeat is sent as AppendRPC to each and every follower at a hearbeat interval. The following json message is sent as heartbeat in my implementation. The entries will be null if there is no log to append. Entries is the array of logs.


```
{
   "request":"APPEND_RPC",
   "term":5,
   "leader_id":"Node1",
   "prevLogindex":5,
   "prevlogTerm":5,
   "entries":[ ]
}
```


d. **RequestRPC** – A follower after getting timed out it is converted into candidate and send RequestRPC to get votes to become leader. The following json message is sent as RequestRPC in my implementation.


```
{
   "request":"REQUEST_RPC",
   "term":5,
   "candidate_id":"Node1",
   "lastLogindex":5,
   "lastlogTerm":5
}
```


e. **Log Structure –** A log will contain term, key and value.


```
{
   "term":"2",
   "key":"K1",
   "value":"Value1"
}
```


f. **Threads** – I am using mutiple threads in my implementation to ensure that UDP packets should not get lost and all the communication happens asynchronously. Following threads are used:

1. **Controller thread** – Listen to the controller request sent by the controller for testing
2. **Listen Thread** – Listen to the AppendRPC and RequestRPC
3. **Election Thread** – Used to send and receive votes once a follower becomes candidate. Vote request is multicasted to all nodes and then the election thread will wait till election timeout for response.
4. **SendHeartbeats** – This thread is used to send Append RPC to follower from the leader
5. **AppendReplyReceive** – This thread opens when a follower becomes Leader and it is used to receive the acknowledgement from the followers when a request is forwarded

3. Implementation

1. Using **tomcat image** as a base image for RAFT and deployed WAR- Web Application Archive inside the tomcat
2. Created a **Raft** image and **Controller** image
3. All the containers are in the same network bridge.
4. Created a Docker file (dockerfile) which contains the detail of base image, starting command and copying WAR inside the working directory
5. Created docker compose file to create five different containers from the RAFT image and a single container for the Controller image.

```
version: '3.7'

networks:
  application_network:

services:
  node1:
    image: raft                 # Same image name for 5 containers
    container_name: Node1       # Container names as per instructions
    build: Node/.
    volumes:
      - ./nodeinfo1:/data/info  # volumes to persist node information and also to store saved logs
    ports:
      - 8095:8080               # Exposed port to access UI
    environment:
      - CANDIDATE_ID=Node1      # Environment variable to identify nodes
      - CANDIDATE_KEY=0
    networks:
      - application_network     # Same application network for all the containers

  node2:
    image: raft
    container_name: Node2
    build: Node/.
    volumes:
      - ./nodeinfo2:/data/info
    ports:
      - 8096:8080
    environment:
      - CANDIDATE_ID=Node2
      - CANDIDATE_KEY=1
    networks:
      - application_network
```

Fig-2.3: Docker Compose File showing entries for 2 nodes

6. Used docker compose to build a network bridge and started 5 containers out of the Raft image and one container out of Controller image created at step 3

Execution Steps:

1. Go to the directory where docker-compose file is placed
2. Run $ **docker compose build**
3. Run $ **docker compose up**
4. Wait for some time to allow system to become stable and leader got selected

4. Validation

a. On running **$docker compose up** we get logs which shows that main thread of the 5 nodes and Controller get started and we also get some tomcat logs to see if all the containers execution is stable. Wait for the logging to get stopped.

```
D:\courses\disbsys\Phase4\Phase4>docker compose up
[+] Running 6/0
 - Container Node4       Created
 - Container Node5       Created
 - Container Node1       Created
 - Container Controller  Created
 - Container Node3       Created
 - Container Node2       Created
Attaching to Controller, Node1, Node2, Node3, Node4, Node5
```

Fig 4.1: logs after $docker compose up

b. **$ docker ps** shows the **running process, container name, image name and port** for all the five node containers and controller container.

```
C:\Users\Akash>docker ps
CONTAINER ID   IMAGE              COMMAND             CREATED         STATUS              PORTS                                       NAMES
27b5ebb5275d   raft               "catalina.sh run"   2 minutes ago   Up About a minute   1234-1235/tcp, 5555/tcp, 0.0.0.0:8098->8080/tcp   Node4
79b25964ec9f   raft               "catalina.sh run"   2 minutes ago   Up About a minute   1234-1235/tcp, 5555/tcp, 0.0.0.0:8095->8080/tcp   Node1
ee219a571932   phase4_controller  "python3"           2 minutes ago   Up 2 minutes        5555/tcp                                    Controller
8e8f1bf211b0   raft               "catalina.sh run"   2 minutes ago   Up 2 minutes        1234-1235/tcp, 5555/tcp, 0.0.0.0:8099->8080/tcp   Node5
ddfb282e5f66   raft               "catalina.sh run"   2 minutes ago   Up About a minute   1234-1235/tcp, 5555/tcp, 0.0.0.0:8097->8080/tcp   Node3
308807c3fb27   raft               "catalina.sh run"   2 minutes ago   Up 2 minutes        1234-1235/tcp, 5555/tcp, 0.0.0.0:8096->8080/tcp   Node2
```

c. To test the system, we are provided with controller implementation in python. I modified the controller code to send **target and request type as the argument** while executing .py file. Also if we want to send **STORE request to leader** we can send key and value as arguments.

Examples of request command

>>python Controller_request.py Node1 LEADER_INFO – This will send request to Node1 to request for LEADER_INFO. Node1 will return LEADER_INFO to the controller back which is printed as result in the Controller CLI.

>>python Controller_request.py Node2 TIMEOUT – This will send request to Node2 to timeout immediately

>>python Controller_request.py Node2 SHUTDOWN – This will send request to Node2 for shutting it down Node2 will exit without throwing any error.

>>python Controller_request.py Node3 CONVERT_FOLLOWER – This will send request to Node3. If Node 3 is leader, then it will become follower and stop sending heartbeats.

>>python Controller_request.py Node3 RETRIEVE – This will send request to Node3. If Node 3 is leader, then it will reply with log entries and if it is follower then it will reply the LEADER_INFO.

>>python Controller_request.py Node3 STORE key1 Value1 - This will send request to Node3. If Node 3 is leader, then it will add key1 and value1 to its log along with the term and if it is follower then it will reply the LEADER_INFO.

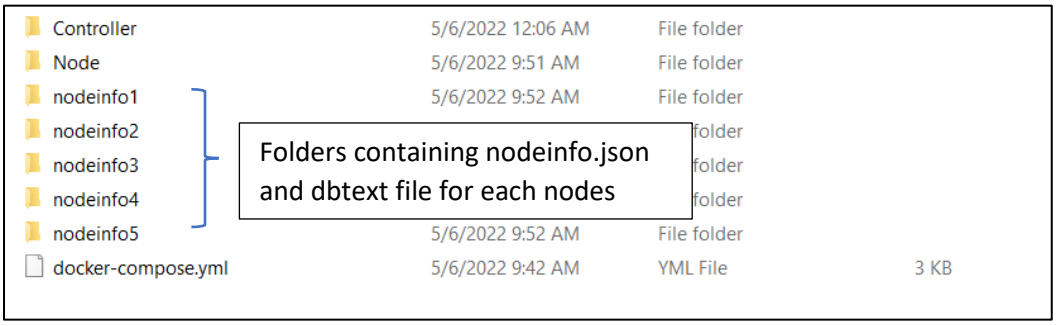
d. Open Controller CLI from docker windows and execute all the controller request

e. Request **LEADER_INFO** –
The command **>>python Controller_request.py Node1 LEADER_INFO** gives the leader info provided by Node1

```
/ # python Controller_request.py Node1 LEADER_INFO
['Controller_request.py', 'Node1', 'LEADER_INFO']
Message Received : {'sender_name': 'Node1', 'request': 'LEADER_INFO', 'term': 2, 'key': 'LEADER', 'value': 'Node1'}
 : ('192.168.208.7', 5555)
Exiting Listener Function
/ #
```

We can see the message received from Node1 which shows **Leader is Node1**

f. Request **STORE** –
The command **>>python Controller_request.py Node1 STORE key1 value1** as Node1 is the leader log will be appended to the leader's log and is replicated to all the nodes.
Once it is replicated to majority of node it gets committed and saved to the DB which is a text file in my implementation.



After sending STORE command to the Leader check the json and text file.

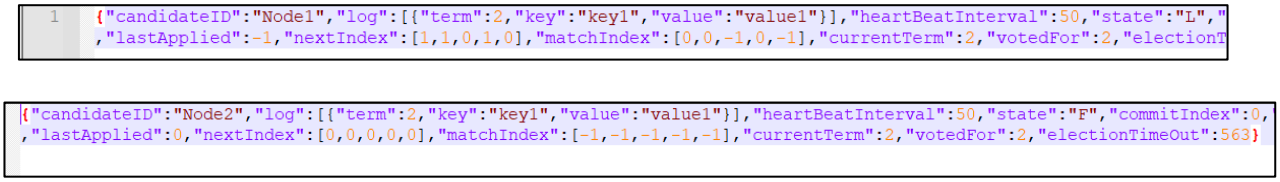


Fig 4.2: nodeinfo.json for Node1 and Node2

We can see the log with term=2 key=key1 and value=value1 is appended to logs of Node1 and Node2.

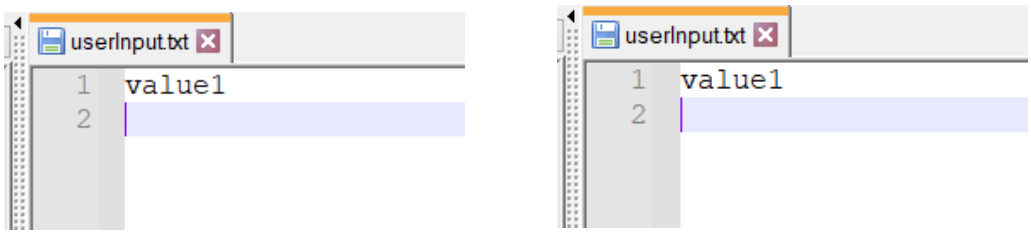


Fig 4.3: userInput.txt for Node1 and Node2

g. Request **STORE** –
The command **>>python Controller_request.py Node2 STORE key1 value1** as Node2 is not the leader it will return the LEADER_INFO

```
/ # python Controller_request.py Node2 STORE key1 value1
['Controller_request.py', 'Node2', 'STORE', 'key1', 'value1']
Message Received : {'sender_name': 'Node2', 'request': 'LEADER_INFO', 'term': 2, 'key': 'LEADER', 'value': 'Node1'}
 : ('192.168.208.3', 5555)
Exiting Listener Function
/ #
```

h. Request **RETRIEVE** –
The command **>>python Controller_request.py Node1 RETRIEVE** as Node1 is the leader it will return the committed log entries.

```
/ # python Controller_request.py Node1 RETRIEVE
['Controller_request.py', 'Node1', 'RETRIEVE']
Message Received : {'sender_name': 'Node1', 'request': 'RETRIEVE', 'term': 2, 'key': 'COMMITED_LOGS', 'value': '[{"term":2,"key":"key1","value":"value1"}]'}
rom : ('192.168.208.7', 5555)
Exiting Listener Function
/ #
```

i. Request **RETRIEVE** –
The command **>>python Controller_request.py Node2 RETRIEVE** as Node2 is not the leader it will return the LEADER_INFO

```
/ # python Controller_request.py Node2 RETRIEVE
['Controller_request.py', 'Node2', 'RETRIEVE']
Message Received : {'sender_name': 'Node2', 'request': 'LEADER_INFO', 'term': 2, 'key': 'LEADER', 'value': 'Node1'}
Exiting Listener Function
/ #
```

j. Request **SHUTDOWN** –
The command **>>python Controller_request.py Node1 SHUTDOWN** will shutdown the leader and a new leader will be elected. Added new entries in the log with new leader and new term. Made the Node1 up and checked if the Node1 is having all the entries as Leader or not.

Note: On SHUTDOWN I am getting warning from the tomcat as there is some limitation in the tomcat due to which it cannot stop multiple threads. It is just a warning the node get shutdown and system performs in the same way as required.
Reference: https://stackoverflow.com/questions/28105803/tomcat8-memory-leak

UI VALIDATION

We can access the UI using the below URL based on which node is LEADER

http://localhost:8095/student-course-reg/ - If Node1 is Leader

http://localhost:8096/student-course-reg/ - If Node2 is Leader

http://localhost:8097/student-course-reg/ - If Node3 is Leader

http://localhost:8098/student-course-reg/ - If Node4 is Leader

http://localhost:8099/student-course-reg/ - If Node5 is Leader

If a Node is Follower, its URL mentioned above is not accessible

We can hit the below end points (exposed for verification) in the followers' servers to check data replication

If Node 1 is Follower- **http://localhost:8095/student-course-reg/student/getCourse**

If Node 2 is Follower - **http://localhost:8096/student-course-reg/student/getCourse**

If Node 3 is Follower - **http://localhost:8097/student-course-reg/student/getCourse**

If Node 4 is Follower - **http://localhost:8098/student-course-reg/student/getCourse**

If Node 5 is Follower - **http://localhost:8099/student-course-reg/student/getCourse**

References

1. https://docs.docker.com/storage/volumes/
2. https://docs.docker.com/engine/reference/builder/
3. https://docs.docker.com/compose/
4. https://docs.docker.com/compose/networking/
5. https://raft.github.io/#implementations
6. https://howtodoinjava.com/java/multi-threading/java-thread-pool-executor-example/
7. http://thushw.blogspot.com/2011/06/asynchronous-udp-server-using-java-nio.html

Proof of Ownership of Laptop: