CSE 586 Distributed Systems

Project – Understanding Consensus in Distributed Systems

Submitted by Madhavi Sajja (50417103), Sai Srinivas Chetti (50418655)

PHASE-3 – RAFT Algorithm – Leader Election, Heartbeats and Timeouts

1. Introduction

The aim is to implement a successful Leader Election algorithm among the multiple (used 5) containers by incorporating Heartbeats, Timeouts, remote-procedure-calls (RPCs) on each node. A leader needs to be elected when old leader server is not reachable, or under few other circumstances. Consensus is a fundamental problem in fault-tolerant distributed systems. It involves multiple servers agreeing on values. It typically arises in the context of replicated state machines, a general approach to building fault-tolerant systems. Each server has a state machine and a log. Leader election is the first stage of RAFT Consensus algorithm.

2. Design Overview

The design includes a connected network of five docker containers called nodes and one controller node used for interactively testing the leader election operations. All the containers are communicating with each other using UDP protocol and socket programming. The 5 nodes operate using multithreading to receive packets/message requests (through listener thread) and to send out packets (through messenger thread) and to read the received packets and change server states based on the received requests. Each server can take any one of 3 states LEADER, CANDIDATE, FOLLOWER. Only one server is supposed to become a leader.

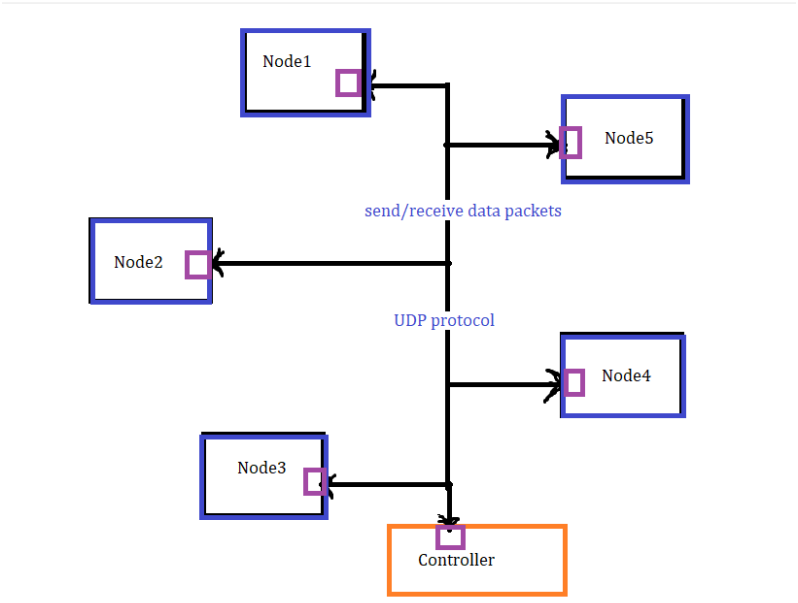


Fig. System Design Overview of Docker Networking using UDP protocol

3. Implementation

Created main file node.py in Node directory. Used techniques like randomized timeout intervals, multithreading, send/receive RPCs.

3.1 Docker-compose YML file

Create docker-compose.yml file with 5 server nodes and a controller node (can take in std inputs).

3.2 Listener Thread

Implemented Listener thread to continuously try to receive data packets sent over to the node through socket programming. The received packets are put in an input message buffer. These messages are popped and read by the main thread to act upon the received message requests. The type of message requests can be one of APPEND_RPC (heartbeat), VOTE_REQUEST, VOTE_ACK, APPEND_RPC_ACK. The controller node can also send message requests like TIMEOUT, CONVERT_FOLLOWER, SHUTDOWN, LEADER_INFO.

3.3 Messenger Thread

When a node wants to send message request to other node(s), it puts the information as (msg, nodes) into an output message buffer. The messenger thread is created to continuously send the messages in the buffer to the specified node(s). Threading helps to perform actions in parallel with actions in other threads.

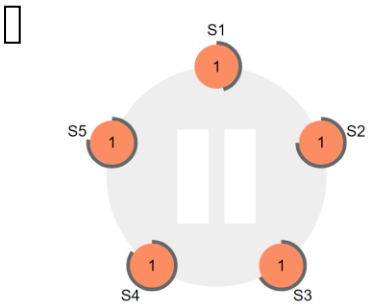
3.4 Timeouts

There are 2 types of timeouts used in the implementation. One is the election timeout, the time interval/elapsed time (here, randomly chosen in between 250ms – 350ms for each node) since the time it has last seen a message request, after which a follower time out. Other timeout is for heartbeat 100ms, the time gap maintained by leader node to send out heartbeats to other nodes.

3.5 Server States

At each node, the messages received in input buffer are read one by one, and actions are taken depending on the type of request, state of the node and the current term. There are 3 possible states for each node, LEADER, FOLLOWER, CANDIDATE.

A follower becomes a candidate when it times out. Candidate can become a leader only if it gets majority votes. Leader stays in the same state until a request of higher term is seen or if it crashes or any forced intervention happens. A candidate/leader always returns to follower state if a request of higher term is seen.



As shown in figure, every server has its own timer. When timeout happens at S1, S1 will request election and all other nodes send their votes to S1. If there is no other candidate available, S1 can get majority votes and becomes leader. S1 starts sending heartbeats (APPEND_RPC requests) to all other nodes. If S1 crashes, next timeout server starts election. If S3 timeout, then S3 is new candidate and starts election. If at least 2/4 servers vote to S3, then S3 becomes new leader, and all others are followers. S3 sends heartbeats.

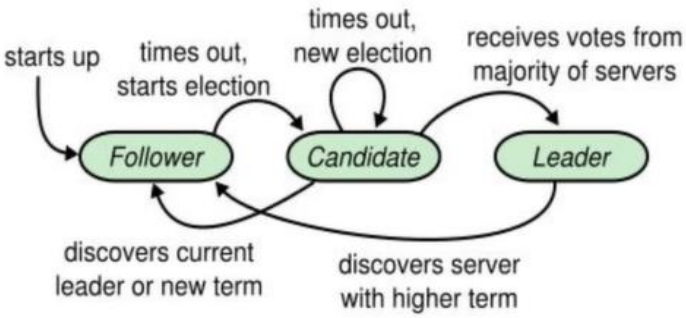


Fig. State Diagram for Leader Election

4. Validation

The controller files convert_follower_node.py, timeout_node.py, leader_info.py, shutdown_ node.py are created for the testing purpose.

Step-1: >> docker-compose -f docker-compose.yml up --build -d

Docker application starts running. Docker containers are created and starts running. All the nodes are initialized to FOLLOWER state, and the node that reaches timeout becomes a candidate and starts election. Multiple nodes can become candidates. The consensus of leadership among the nodes is communicated, and one of the nodes will become a leader.

```
Creating Controller ... done
Creating Node1      ... done
Creating Node4      ... done
Creating Node2      ... done
Creating Node3      ... done
Creating Node5      ... done
Attaching to Node5, Node2, Node3, Node1, Node4, Controller
Node1          | Starting Node Node1
Node1          | Setup listener thread and messenger thread.
Node1          | RUNNING..
Node2          | Starting Node Node2
Node2          | Setup listener thread and messenger thread.
Node2          | RUNNING..
Node3          | Starting Node Node3
Node3          | Setup listener thread and messenger thread.
Node3          | RUNNING..
Node5          | Starting Node Node5
Node5          | Setup listener thread and messenger thread.
Node5          | RUNNING..
Node2          | TIME_OUT: Node2 is now candidate.
Node5          | TIME_OUT: Node5 is now candidate.
Node4          | Starting Node Node4
Node4          | Setup listener thread and messenger thread.
Node4          | RUNNING..
Node3          | TIME_OUT: Node3 is now candidate.
Node5          | Older candidate available. Candidate will become Follower
Node3          | Older candidate available. Candidate will become Follower
Node2          | -----
Node2          | Node2 -> ELECTED LEADER WITH 3 VOTES
Node2          | -----
```

Step-2: Open Command Line interface of Controller node in Docker desktop app. The working of the leader election can be tweaked using the controller test cases.

>> python leader_info.py

Controller sends message request “LEADER_INFO” to one of the nodes (here Node1). And the leader information is sent over as message request from that node (here Node1) back to controller. The output is displayed. Since Node2 is elected as leader, output displays Node2.

```
/ # python leader_info.py
Request Created : {'sender_name': 'Controller', 'request': 'LEADER_INFO'
, 'term': None, 'key': None, 'value': None} and sent to Node1
..........
LEADER_INFO: Node2 is current leader.
..........
```

>> python convert_follower_node.py Node2

Controller sends “CONVERT_FOLLOWER” request to Node2. This changes the state of Node1 to Follower as the original state is not follower. Since Node2 is a leader, it becomes a follower. As there is no leader, a re-election is started when one of the followers times out and becomes a leader. Here we can see that Node1 became a leader.

```
/ # python convert_follower_node.py Node2
Request Created : {'sender_name': 'Controller', 'request': 'CONVERT_FOLL
OWER', 'term': None, 'key': None, 'value': None} and sent to Node2
/ #
```

```
Node2   | -----
Node2   | Node2 has changed from LEADER to FOLLOWER.
Node3   | TIME_OUT: Node3 is now candidate.
Node1   | TIME_OUT: Node1 is now candidate.
Node5   | TIME_OUT: Node5 is now candidate.
Node2   | TIME_OUT: Node2 is now candidate.
Node4   | TIME_OUT: Node4 is now candidate.
Node2   | Older candidate available. Candidate will become Follower
Node4   | Older candidate available. Candidate will become Follower
Node5   | Older candidate available. Candidate will become Follower
Node3   | Older candidate available. Candidate will become Follower
Node1   | -----
Node1   | Node1 -> ELECTED LEADER WITH 3 VOTES
Node1   | -----
```

>> python timeout_node.py Node2

If the input node Node2 is a follower at present. So, on timeout, it increments the current term, becomes a candidate, and starts a new election. If no other nodes are expected to have timed out, Node2 becomes a leader.

```
..........
/ # python timeout_node.py Node2
Request Created : {'sender_name': 'Controller', 'request': 'TIMEOUT', 't
erm': None, 'key': None, 'value': None} and sent to Node2
/ # python leader_info.py
Request Created : {'sender_name': 'Controller', 'request': 'LEADER_INFO'
, 'term': None, 'key': None, 'value': None} and sent to Node1
..........
LEADER_INFO: Node2 is current leader.
```

>> python shutdown_node.py Node2

The input node Node2 will be stopped from working (reading/receiving/sending messages). Since, Node2 is current leader, shutting down Node2 will cause a re-election and Node3 is selected as Leader.

```
/ # python leader_info.py
Request Created : {'sender_name': 'Controller', 'request': 'LEADER_INFO'
, 'term': None, 'key': None, 'value': None} and sent to Node1
..........
LEADER_INFO: Node2 is current leader.
..........
/ # python shutdown_node.py Node2
Request Created : {'sender_name': 'Controller', 'request': 'SHUTDOWN', '
term': None, 'key': None, 'value': None} and sent to Node2
/ # python leader_info.py
Request Created : {'sender_name': 'Controller', 'request': 'LEADER_INFO'
, 'term': None, 'key': None, 'value': None} and sent to Node1
..........
LEADER_INFO: Node3 is current leader.
..........
```

5. References

Used the sample codes provided by TAs for sample socket programming and Controller tests.

https://raft.github.io/

https://raft.github.io/raft.pdf

http://thesecretlivesofdata.com/raft/