Semester Project

Parallel and Distributed Computing

RAFT Consensus Algorithm Implementation

Submitted By:
Faraz Saeed (i160107)
Hamza Tariq (i160134)
Husnain Gauher (i160031)

Submitted to:
Dr. Ehtesham Zahoor & Dr. Shujaat Hussain

Project Overview

Before going into the details of the project, We will first discuss the problem this algorithm solves. That is, consensus. **Consensus** is a fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This often requires processes to agree on some data value that is needed during computation. Examples of applications of **consensus** include whether to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, and atomic broadcasts. The real world applications include clock synchronization, PageRank, opinion formation, smart power grids, state estimation, control of UAVs (and multiple robots/agents in general), load balancing, blockchain and others.

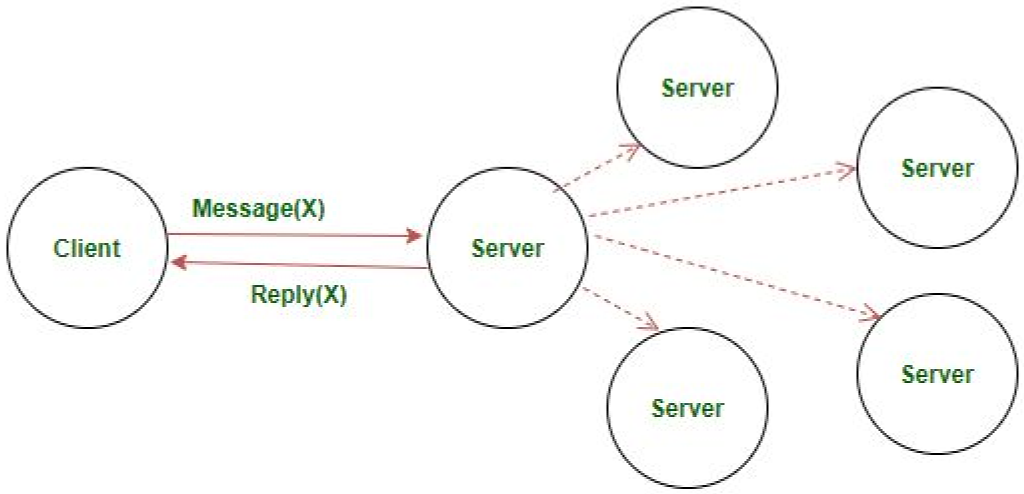
After discussing the nature of the problem, now let's discuss a solution to solve the consensus problem. There are multiple approaches to solve the consensus problem and the algorithmic approach we tend to use is **RAFT**. RAFT generally divides the problem into multiple independent subproblems and then joins them together to effectively solve the consensus. It works on the principle of leader election. From the pool of nodes, a leader is elected and then that leader is responsible for the connection with the client and maintaining the state of the pool of machines. If the leader, somehow, disconnects then a new leader is elected by another election.

We will below discuss the sub problems we had to solve for developing the RAFT.
But before that we will briefly discuss the type of system we are working on. It is termed as **Multiple Server system** and can be described as

Multiple Server system : The client interacts with a system having multiple servers. Such systems can be of two types :

- Symmetric :- Any of the multiple servers can respond to the client and all the other servers are supposed to sync up with the server that responded to the client's request, and
- Asymmetric :- Only the elected leader server can respond to the client. All other servers then sync up with the leader server.

Given below is an example of an asymmetric multiple server system which we shall use throughout this project.



Here we can see a client is connected to a server which is further connected to all nodes in the pool. The server with which the client is connected will be the termed as the leader whereas others will be termed as the followers.

We will discuss these terms after going through the task breakdown for achieving this state.

Task Breakdown

- Leader Election
- Heartbeat
- Split Vote
- Log Replication
- Partition

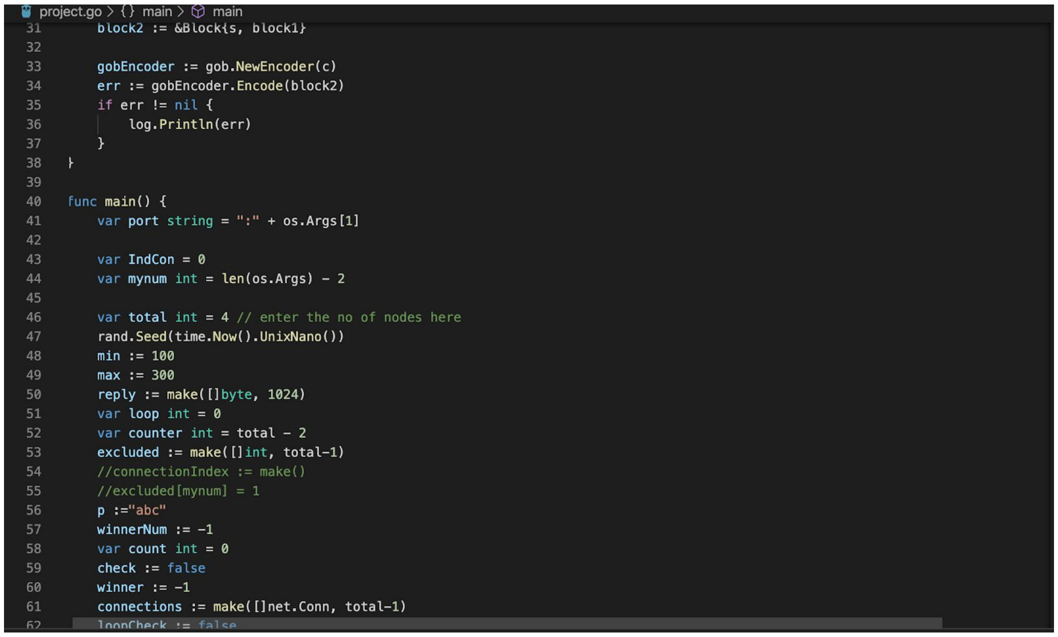
Leader Election

For the implementation of leader election protocol in GOLANG, we connected multiple nodes together with a random time between **100ms** to **300ms.** The node whose timer would expire the earliest would send out a stop message to other nodes so they could respond back or vote for the leader.
The algorithm works in a way that we know the read statement is blocking so we use a command(SetReadDeadline(time.Now().Add(time.Millisecond))) with each connections's

read which makes it non blocking. This serves the purpose of listening for a ‘stop’ message as well as timing out for election.

```
randtimer := rand.Intn(max-min+1) + min
loop=0
For i:=0;i<randtimer;i++{
connections[loop].SetReadDeadline(time.Now().Add(time.Millisecond))
               _, err = connections[loop].Read(reply[0:])
if err != nil {
               } else {
                   println("Message Received: = ", string(reply), " From Node: ",
loop)
                   check = true
                   winner = loop // this node's timer was ended the first
                   break
               }

loop++
}
```

For running the project, you first specify the **number of nodes at line number 46. In our example we are using 4 nodes.**


```
project.go > {} main > main
    block2 := &Block{s, block1}

    gobEncoder := gob.NewEncoder(c)
    err := gobEncoder.Encode(block2)
    if err != nil {
        log.Println(err)
    }
}

func main() {
    var port string = ":" + os.Args[1]

    var IndCon = 0
    var mynum int = len(os.Args) - 2

    var total int = 4 // enter the no of nodes here
    rand.Seed(time.Now().UnixNano())
    min := 100
    max := 300
    reply := make([]byte, 1024)
    var loop int = 0
    var counter int = total - 2
    excluded := make([]int, total-1)
    //connectionIndex := make()
    //excluded[mynum] = 1
    p :="abc"
    winnerNum := -1
    var count int = 0
    check := false
    winner := -1
    connections := make([]net.Conn, total-1)
    loopCheck := false
```

We can run the project file by giving port numbers to each node such as

How To Run

Commands

Go run project.go 1010

Go run project.go 1011 1010

Go run project.go 1012 1010 1011

Go run project.go 1013 1010 1011 1012

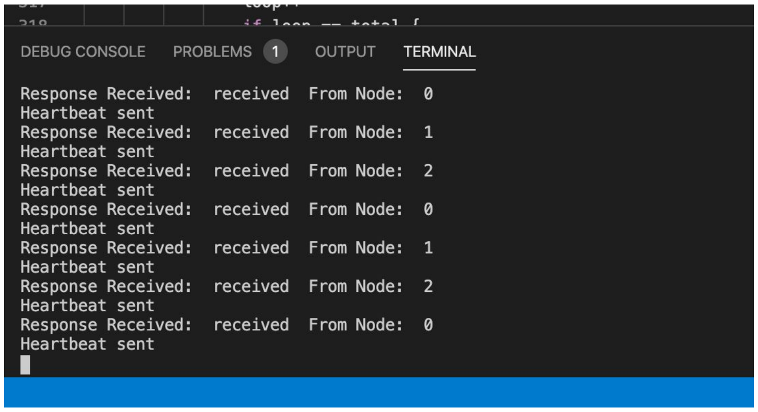
Where as first port number is the server port number of current node and rest of all are the port number of other nodes to which it needs to be connected

Heartbeat

A heartbeat is a message being sent to followers of a leader and the leader then receives the acknowledgment of the heartbeat before sending another one. Heartbeat makes sure the leader is alive and responding.

Leader node

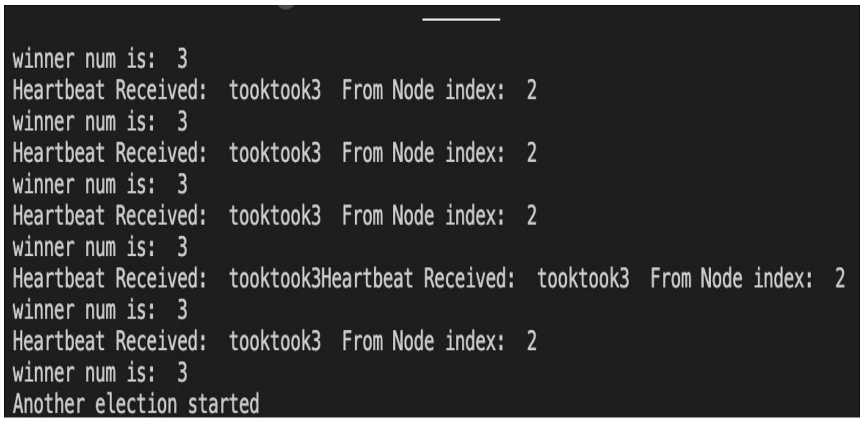
The below screenshot is the leader node which shows that after being elected, it is sending heartbeats and receiving response to and from the other nodes/followers connected to it.



```
Response Received:  received  From Node:  0
Heartbeat sent
Response Received:  received  From Node:  1
Heartbeat sent
Response Received:  received  From Node:  2
Heartbeat sent
Response Received:  received  From Node:  0
Heartbeat sent
Response Received:  received  From Node:  1
Heartbeat sent
Response Received:  received  From Node:  2
Heartbeat sent
Response Received:  received  From Node:  0
Heartbeat sent
```

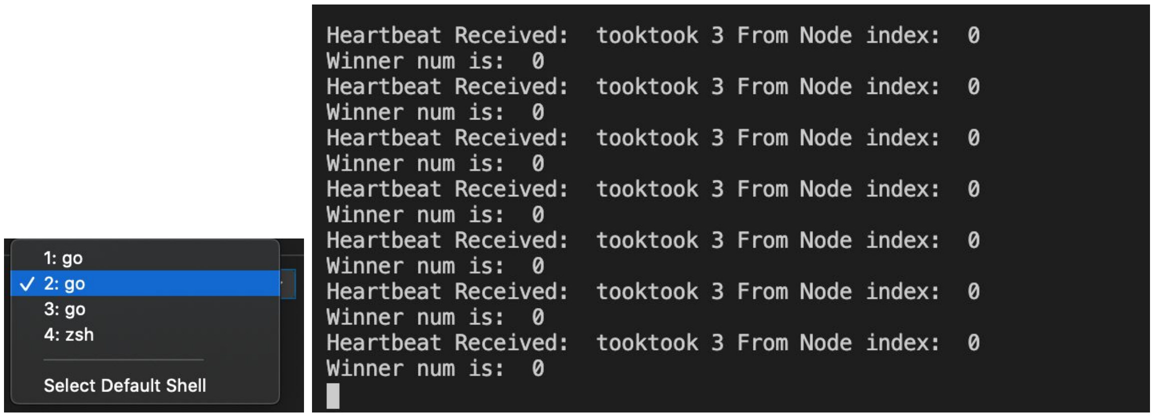
Follower node
Below is a screenshot of one of the follower node receiving heartbeats from the winner/leader node.

Use Case: starting another election if heartbeat not received from leader in time
The image below shows a **USE CASE** where the a follower wouldn't receive heartbeat from leader in the time bound and it starts another election thinking leader is inactive



```
winner num is:  3
Heartbeat Received:  tooktook3  From Node index:  2
winner num is:  3
Heartbeat Received:  tooktook3  From Node index:  2
winner num is:  3
Heartbeat Received:  tooktook3  From Node index:  2
winner num is:  3
Heartbeat Received:  tooktook3Heartbeat Received:  tooktook3  From Node index:  2
winner num is:  3
Heartbeat Received:  tooktook3  From Node index:  2
winner num is:  3
Another election started
```

This is the **output** after re-election. Now node 0 is the leader instead of node 3



```
Heartbeat Received:  tooktook 3 From Node index:  0
Winner num is:  0
Heartbeat Received:  tooktook 3 From Node index:  0
Winner num is:  0
Heartbeat Received:  tooktook 3 From Node index:  0
Winner num is:  0
Heartbeat Received:  tooktook 3 From Node index:  0
Winner num is:  0
Heartbeat Received:  tooktook 3 From Node index:  0
Winner num is:  0
Heartbeat Received:  tooktook 3 From Node index:  0
Winner num is:  0
Heartbeat Received:  tooktook 3 From Node index:  0
Winner num is:  0
```

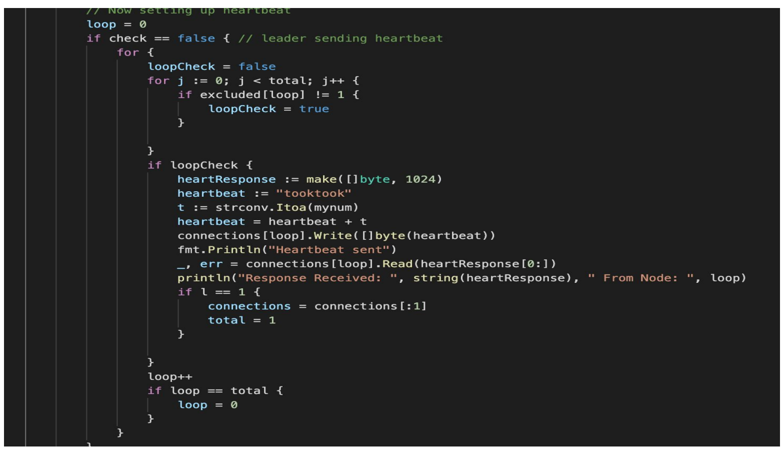
Split Vote

This is the case where two nodes timeout at the same time at the leader election phase. We have catered this use case by checking if a node that has timed out receives a stop message from another node then we can say there has been a collision between two candidates and we make all the nodes contest another election.

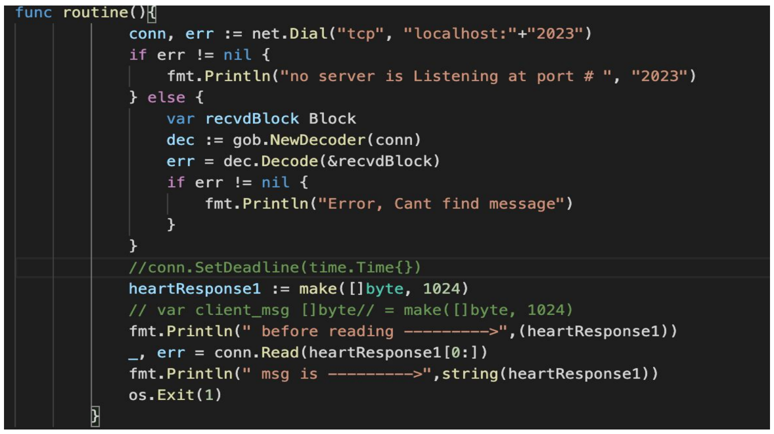
Log Replication

For this module, a client gets connected to the leader and sends a message to him. The leader then broadcasts that message to its followers and they send back acknowledgement. The leader then also sends back an ack to the client.
Client → **Leader** → **Followers** and then **Followers** → **Leader** → **Client**

The leader connects with the client and waits for the message from the client. All this functionality is done in a function which is then called by goroutine so it could run in a separate thread. When that thread of leader receives a message, it forwards it to the clients through the heartbeat buffer.



```
        // Now setting up heartbeat
        loop = 0
        if check == false { // leader sending heartbeat
            for {
                loopCheck = false
                for j := 0; j < total; j++ {
                    if excluded[loop] != 1 {
                        loopCheck = true
                    }

                }
                if loopCheck {
                    heartResponse := make([]byte, 1024)
                    heartbeat := "tooktook"
                    t := strconv.Itoa(mynum)
                    heartbeat = heartbeat + t
                    connections[loop].Write([]byte(heartbeat))
                    fmt.Println("Heartbeat sent")
                    _, err = connections[loop].Read(heartResponse[0:])
                    println("Response Received: ", string(heartResponse), " From Node: ", loop)
                    if l == 1 {
                        connections = connections[:1]
                        total = 1
                    }

                }
                loop++
                if loop == total {
                    loop = 0
                }
            }
```



```
func routine(){
            conn, err := net.Dial("tcp", "localhost:"+"2023")
            if err != nil {
                fmt.Println("no server is Listening at port # ", "2023")
            } else {
                var recvdBlock Block
                dec := gob.NewDecoder(conn)
                err = dec.Decode(&recvdBlock)
                if err != nil {
                    fmt.Println("Error, Cant find message")
                }
            }
            //conn.SetDeadline(time.Time{})
            heartResponse1 := make([]byte, 1024)
            // var client_msg []byte// = make([]byte, 1024)
            fmt.Println(" before reading --------->",(heartResponse1))
            _, err = conn.Read(heartResponse1[0:])
            fmt.Println(" msg is --------->",string(heartResponse1))
            os.Exit(1)
        }
```

Partition

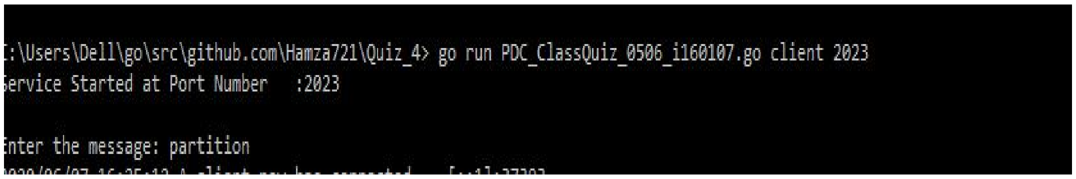
For the partition module, we make changes to the **connection** array stored in each node. This connection array serves the purpose of connecting each node with others. The client sends a message to its connected leader to partition and when a leader receives that message, the following changes are made.

Command to connect with client and run to get partition message:

Go run project.go client <port number>

Example

Go run project.go client 2022



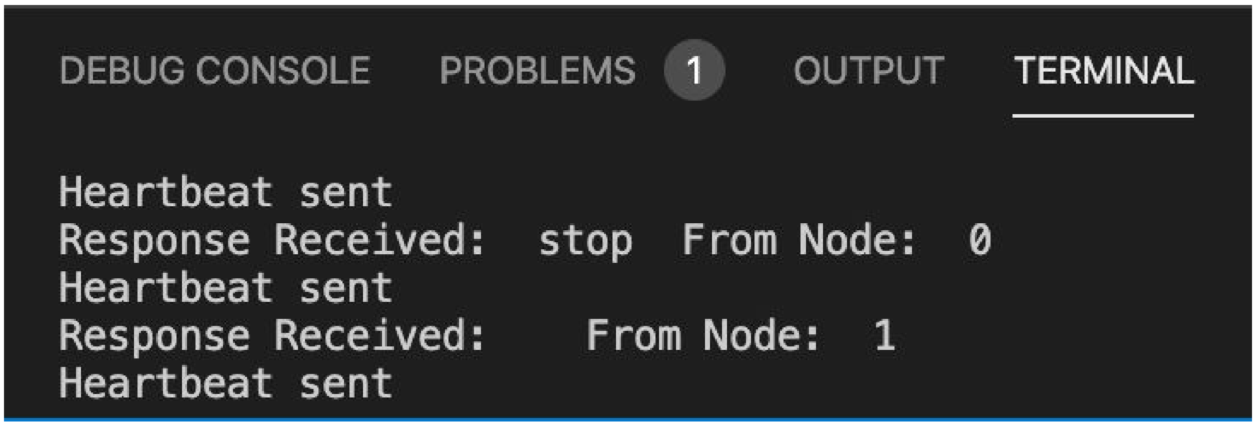
```
:\Users\Dell\go\src\github.com\Hamza721\Quiz_4> go run PDC_ClassQuiz_0506_i160107.go client 2023
ervice Started at Port Number   :2023

nter the message: partition
020/06/07 16:25:12 A client now has connected    [::1]:27202
```

```
If leader node
    connections= connections[:0]
If follower node
    If my number==leader.connection[0]
        connections=connections[leader]
    Else
        connections.remove(leader)
        connections.remove(leader.connections[0])
```

The above algorithm partitions the nodes in a way such that after partition, the leader is connected to only one node and the rest of the nodes are separated from its connections array. When those nodes don't receive a heartbeat, they presume the leader to be inactive and re contest elections to choose another leader.

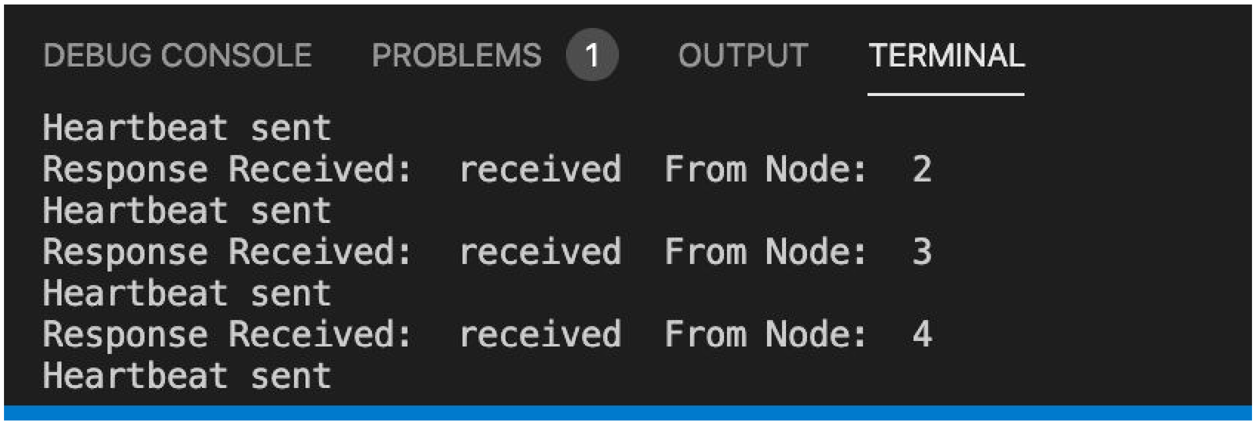
The following example is run by 7 nodes. Where after partition, 0 and 1 are connected to node 5 and 3,4 and 4 are connected to node number 6.

Output of node number 5



```
Heartbeat sent
Response Received:  stop  From Node:  0
Heartbeat sent
Response Received:    From Node:  1
Heartbeat sent
```

Output of node number 6



```
Heartbeat sent
Response Received:  received  From Node:  2
Heartbeat sent
Response Received:  received  From Node:  3
Heartbeat sent
Response Received:  received  From Node:  4
Heartbeat sent
```