



Final Project Report

Raft Engine: A Consensus Algorithm

COEN 317: Distributed Systems

Winter 2016

Professor: Ramin Moazzeni

Team:

Shilpita Roy

Sruthi Gottumukkala

TABLE OF CONTENTS

1. Abstract
2. Goals
3. Introduction
 - 3.1. Consensus Algorithm
 - 3.2. Replicated state machines
 - 3.3. Raft Consensus Algorithm
 - 3.3.1. Server States
 - 3.3.2. Leader Election
 - 3.3.3. Server State Scenarios
 - 3.3.4. Split Votes
 - 3.3.5. Log Replication
4. Distributed System challenges
 - 4.1. Consensus among Nodes
 - 4.2. Leader Election
 - 4.3. Log Replication using Two phase commit
 - 4.4. Safety and fault tolerance
5. Related work
6. Major design decisions
 - 6.1. RMI Registry
 - 6.2. JGroups
 - 6.3. Sockets
7. Implementation
 - 7.1. Design Specification of Raft
 - 7.2. Raft Interface and classes
 - 7.3. Raft Election Task
 - 7.3.1. Responding to Request Vote RPC
 - 7.3.2. Split Election
 - 7.4. Append Entry Task
 - 7.4.1. Responding to Append Entry RPC
 - 7.4.2. Log Replication using Two Phase Commit
8. Test Cases
 - 8.1. Election task
 - 8.2. Split Votes
 - 8.3. Candidate stepped down
 - 8.4. Log Replication
9. Limitations
10. Future work and improvements
11. Conclusion
12. References
13. Appendix

1. Abstract

As we began our preliminary search for a topic for distributed systems for our project we came across the topic of consensus algorithm. We began with the study of Paxos consensus algorithm and found that in several papers it has been stated that Paxos is difficult to understand and implement. Another simpler alternative to Paxos was suggested as the RAFT Consensus in the paper '*IN Search of an Understandable Consensus Algorithm*' by *Diego Ongaro and John Ousterhout of Stanford University*. This paper will henceforth be referred to as the base paper in all our project proposal. We wished to implement some of the algorithms we covered in our academic course. Raft being a consensus algorithm has leader election, log replication and fault tolerance. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos. [1]

2. Goals

Initially as we studied the Raft algorithm, we found many industrial implementation and libraries available and in use including JGroups. We studied these libraries and methods in detail and implemented the algorithm from the base in Java using RMI. We wanted to implement the Consensus engine with Leader election task and Log replication with fault tolerance. The data is a Record of x and y values and its sum. This data is stored in the state machine shared by the system. But the access to the state machine is with the Leader node. Each node has its own copy and the State machine is replicated on these local nodes using a two phase commit method through RPC.

The goal of our project is to perform **leader election** using raft protocols which involves state transition between the nodes. This should give the distributed cluster a strong leader minimizing the split voting.

Once the leader is elected it should send heartbeats in form of appendEntry RPC as well as the Records to be stored. The followers should validate these data received through RPC and respond to it. Based on these responses the leader adds the record in the state machine. This later replicated on all the followers. This is the **Log Replication with two phase commit**.

Another secondary goal was to learn the Java RMI and its classes through practical application.

3. Introduction

3.1. Consensus algorithms

Consensus algorithms allow a collection of machines to work as a fault tolerant coherent group. 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^[2]. Any Consensus problem requires an understanding or protocol among the different nodes for communication as well as data storage. Consensus involves multiple servers agreeing on values. Once they reach a decision on a value,

that decision is final. Typical consensus algorithms make progress when any majority of their servers are available.

Paxos is currently dominating discussion on the Consensus algorithm but Paxos is difficult to understand and implement from base. Raft provides the same advantages as Paxos but is easier to understand and implement. The architecture of Raft algorithm is decomposed in parts to make it more comprehensive.

Raft is similar to all consensus algorithm in the fundamental way:

Strong leader: Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.

Leader election: Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.

Membership changes: Raft's mechanism for changing the set of servers in the cluster uses a new *joint consensus* approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

3.2. Replicated state machines

Consensus algorithms are typically used in the context of *replicated state machines*. Replicated state machine has been implemented using a replicated log. Making the replicated log consistent is the job of the consensus algorithm. The server receives a command from the client and adds it to its log. Then it communicates with the other servers in the cluster to ensure the log is replicated. If the commands are replicated properly, the server notifies the client. Thus the server functions as a single highly reliable state machine.

3.3. Raft Consensus Algorithm

The Raft algorithm implements consensus by electing a unique *leader*, the leader is then responsible for managing the replicated log. The leader accepts log entries from the client, and then replicates them on other servers, the servers acknowledge the addition of the log entry to their local log. Having a leader helps simplify the management of the replicated log.^[1]

- Leader election: a new leader is chosen when a follower doesn't receive a heartbeat.
- Log replication: the leader must accept log entries from client and replicate it on the servers on the cluster.

3.3.1. Server States

A Raft cluster typically contains five servers; this way the system can tolerate up to two failures. At any given point of time, a raft server can be in one of the three states: *leader*, *follower*, or *candidate*.^[1] There can be only one leader and the rest of the servers in the clusters are followers. The role of the followers is passive. they do not participate in request handling their primary goal is to respond to requests from leaders as well as candidates. It is the leader who handles all the requests from the client. In the case where a client contacts a follower, it is the duty of the follower to redirect the request to the leader. The third and final state is the candidate, when a

server is in this state it is ready to participate in an election.

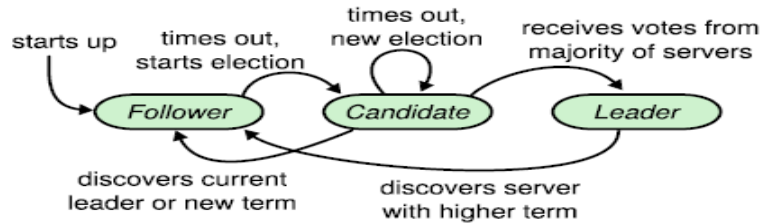


Figure 1: States of the Server

3.3.2. Leader election

The algorithm detects the need for an election with a heartbeat mechanism. When a server starts, it begins as a follower. A server maintains its follower state as long as it receives valid RPCs from a leader. A candidate who has been elected a Leader, sends out heartbeats periodically to all the followers in the cluster in order to maintain its state as a leader. In the scenario where a follower doesn't receive any kind of communication for a period of time then an *election timeout* is called, since no legit leader exists, it assumes a no leader scenario and begins an election in order to select a new leader. ^[1]

In order to start an election, a follower increments its current term and changes its state to the candidate state. Then the candidate who has called for an election votes for itself and sends RequestVote RPCs in parallel to all the servers in the cluster.

3.3.3. Scenarios of a Server State

A candidate will continue to stay in this state until one of the following three scenarios occurs. (a) if it wins an election, (b) another server becomes the leader, or (c) no winner has been declared in a predefined time period. ^[1]

If a candidate receives votes from the majority, it wins the election for that term in the cluster. Each server can vote for only one candidate in a given term, which is based on a first come first serve basis. The majority ruling makes sure that only one candidate wins the election for the current term. When a candidate wins the election, it becomes the leader/coordinator of the cluster. It starts sending heartbeats to all servers in its cluster so that a new election is prevented.

In the candidate state, while it is waiting for votes, it may receive an AppendEntries RPC from another server which means another server is the leader. It then checks if the leader's term is equal to its current term, if true, the candidate considers the leader to be legit and changes its state to follower. But if the term sent in the RPC is smaller than the current term of the candidate, then the RPC is rejected and doesn't change state.

In the third scenario, the candidate neither wins nor loses the election. This may be because there are many followers who have changed to candidates at one instance, this means the votes might have been split, hence no candidate gets a majority. When this is the outcome, the candidate times out and a new election is started by incrementing the current term and sending out RequestVote RPCs.

3.3.4. Split Votes

The algorithm makes use of election timeouts which are made random in order to ensure that if split votes occur they are immediately resolved. The election timeouts are chosen randomly so that in most scenarios only a single server times out and sends out heartbeats before the other server times out. This way all servers don't timeout at the same time and start an election at the same time. It improves the efficiency of the algorithm.

Every server in a candidate state restarts its election timer with a random timeout at the beginning of an election, it then waits for that timeout to occur before it can begin the next election. This reduces the probability of the next split vote in the new election.

3.3.5. Log Replication

When a leader has been elected during the Leader Election algorithm, it starts responding to client requests. The client request contains a log entry to be replicated on the state machines of the servers. It is the responsibility of the leader to append the new entries received from the client to its log, it then sends out AppendEntries RPCs in parallel to all the servers in the cluster to replicate the entry in their local logs. If the entry has been replicated, the leader waits for a majority of acknowledgements from the other servers, it imposes the replication on the state machine and returns the same to the client.

Log Entry Organization:

Each log entry contains the term number and the index along with the data sent from the client. The term numbers in the log are necessary as they can help detect inconsistencies between logs. Once a log entry has been safely replicated, the leader commits the operation using a **2 Phase Commit**. The log entry is said to be committed if the leader has replicated the entry on a majority of the servers. The leader also saves the highest index that has committed, and this index is included in later AppendEntries RPCs. This way the followers know what is the most complete and latest log entry. When the leader receives acknowledgement from a majority of followers that a log entry has been committed, it appends the log entry to its own local log.

4. Distributed System Challenges Addressed

4.1. Consensus among nodes.

Consensus is an essential concept in a fault-tolerant distributed system. It involves various servers agreeing on a value. When they decide on a value, the choice becomes final. Classic consensus algorithms work on a task when a majority of their servers are alive. For example, if there is a cluster of 5 servers, the distributed system will continue to operate even if 2 of the servers fail. But if more than 2 fail, then the entire system fails. A consensus algorithm always satisfies the following properties: ^[2]

Termination

All the servers decide on one value.

Validity

If all the servers suggest the same value v , then all correct servers decide the value v .

Integrity

Every server decides at most one value, and if it decides on a value v , then v must have been proposed by some server.

Agreement

Every server must agree on the same value v.

4.2. Leader Election in a cluster of Nodes

Leader election is a process of choosing a single process as the coordinator of the task distributed in a cluster of nodes. Before the task can start, the nodes are unaware about which node will be selected as the leader of the task. After a leader election algorithm has run, every node recognizes a single distinctive node as the leader.^[3]

4.3. Maintaining replica of the State Machine

Replication of the state and functionality of the distributed system is the principal technique for masking a failure so that the cluster can continue to function even if some servers fail.^[4]

4.4. Safety and fault tolerance

Fault tolerance allows a system to continue its functionality even if some of its components fail. Every fault tolerant system requires to reflect the following:

- (i) **No single point of failure:** If case a system fails, it must continue to function without disruption during the fixing process.
- (ii) **Fault Isolation:** In case a failure happens, the system must be capable of isolating the failure. This would need the establishment of a failure detection mechanism that has been designed with fault isolation in mind.
- (iii) **Fault containment:** In order to to prevent propagation of the failure a mechanism should be put in place to avoid corrupting the entire system.^[5]

5. Related work

There are many implementations of Raft available in various stages of development. These implementations provide library files and predefined methods for implementing consensus in a distributed system using Raft [3]. These implementations are in variety of languages like C, C++, Java, Go Python etc.

Some of the works include-

- Jgroup Raft library
- Go-raft
- LogCabin
- Zraft-lib

For our project we have implemented the algorithm without any predefined library.

6. Major design decisions

6.1. RMI Registry

Deciding to use RMI Registry was a major design decision we had to take as it can account to a single point of failure. The RMI registry is a service used by servers to register the services they are capable of providing and the clients can query those for those services. The RMI registry can be viewed as a broker that connects the server and client.^[6]

6.2. Forming a Distributed Cluster

JGroups vs RMI

Another design decision was to decide between JGroups and RMI. Although JGroups is a simple and a reliable java library that lets the servers communicate using multicast messages we decided to use RMI since our design demanded the use of RPCs to communicate between the servers in the clusters.

This library achieves a distributed cluster by adding a grouping layer over a transport protocol while keeping a list of participants. This list ensures that:

- 1 The application is aware of the listeners in the group.
- 2 Some or all transmissions are reliable.
- 3 Allows ordered transmissions. ^[7]

Although JGroups is a relatively easy library to use to form a cluster, we decided to implement RMI which is simple, elegant and easy to use as most of the network specifics are handled by RMI

RMI vs Socket Programming

In Socket programming we need to know exactly which sockets are being used, whether TCP or UDP, all the formatting of messages streaming between a client and server are explicitly handled by the user.

Whereas RMI hides most of the network specific code as a result of which specifics about the ports used in the application are directly handled by RMI, the streaming of messages between a client and server are implicitly handled by RMI. It is essentially a layer over sockets used in distributed systems where some transparency is needed and remote methods need to be called. It also allows for stateless connections to the server. ^[8]

7. Implementation

We have implemented Raft algorithm in Java JDK 8.0. The main components of a Raft algorithm are Leader election, Log replication and Safety. We have implemented leader election and log replication. The algorithm is implemented on five nodes.

7.1. Specifications of Raft

A node can only vote once in an election term. This ensures that there is a winner in election. This was managed by a `votedCurrTerm` counter.

To reduce the amount of terms in split election with no emergence of leader the election timeout for each node is randomized. This is necessary to guarantee that cluster spends less time in election and more time performing tasks with client.

If a term number and the index number for a data entry is the same then the data in the entry is the same.

7.2. Raft Interface and classes

Class Record

The class Record is created to hold the data that is to be maintained over the cluster. It holds the value of x and y variables generated randomly. This is the data that the client is sending to the cluster. It implements the class `Serializable` as this object need to be serialized and de-serialized.

Class LogEntry

This class is created to describe what a Data entry in raft will contain. Each Log Entry object contains the term number in which it was received by the Raft engine. The sum of the values of x and y in the Record object received from the client and the record object itself.

Class RaftRequestInterface

This class extends the class `java.rmi.Remote`. Its an interface class that declares the remote methods that are used in RAFT. Each remote method must declare `java.rmi.RemoteException` (or a superclass of RemoteException) in its throws clause, in addition to any application-specific exceptions.

Method appendEntryRequest :

```
public int appendEntryRequest (LogEntry newLogEntry
                               , int newLogIndex
                               , String leaderName
                               , LogEntry lastCommittedEntry
                               , int lastCommittedLogIndex) throws
RemoteException ;
```

Method voteRequest :

```
public int voteRequest (int newTerm
                        , String candidateName
                        , LogEntry lastCandidateCommittedEntry
                        , int lastCandidateCommittedIndex ) throws
RemoteException ;
```

Class Node

The implements the methods declared in the interface ***RaftRequestInterface***. This is also used as the main class with the raft implementation. The name of each node is decided by the timestamp

it was started on. The state of the nodes is enumerated as LEADER, FOLLOWER, CANDIDATE. At a given time, the node will be in one of these state and this state will decide the behavior and the task of each node. Each node maintains its own local log which is the replica of the state machine log. Only the leader can make updates to the state machine log. But each node can make updates on its own local log based on the log entries that the leader has committed to the state log.

The goal is to keep all the local log as complete as the state log. Also once a data has been added to the state log it cannot be changed. Any edits that are made are made on the local logs to match the state log. A cluster is said to be updated and safe when majority of the local logs in the cluster are replicated as the state log.

The Timer `RaftTimer` is used to schedule the two tasks

- `RaftElectionTask` – The election task to elect a Leader
- `AppendEntryTask` – The append entry task that sends data to all followers as well as works as heartbeats.

7.3. Raft Election Task (Leader Election Algorithm)

When the cluster is started all the nodes start off as being Followers and wait for a heartbeat from the leader. This waiting period is called the message period. IN our implementation we have set up the message period to be 4msec. After the message period elapses without heartbeat from the leader the node starts an election by performing the following steps:

- It increments its current term
- It changes from Follower to Candidate
- It votes for itself
- It sends `voteRequest` RPC to all the nodes in the RMI Registry list()

7.3.1. Responding to Request Vote RPC

The `voteRequest` RPC has the current term of the candidate node, candidate name and the details of the last committed entry of the candidate's local log (last committed data term and last committed data index)

When a node receives a `voteRequest` it verifies the credibility of the request and candidate before granting the vote. Each node can vote once in a term. The `voteRequest` method returns a idempotent value of 0 (false) or 1 (true).

The receiver can -

- Reply false if received term (candidate's) < currentTerm (receiver's)
- If `votedFor` is null for current term or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote as true.

When the candidate receives majority of the votes in the cluster it changes its state to Leader and sends a null `appendEntry` RPC to assert itself as leader over the cluster. When the other candidates receive this `appendEntry` RPC from the new leader they end the election task and change state to Follower.

7.3.2. Split Election

For an election term there can be multiple candidates campaigning to become a leader. This can result in splitting of vote count resulting in no emergence of leader for an election term. This is called a *split election*. When this happens the candidates again become follower and start a new election term.

Repeated occurrence of split election reduces the time when the cluster can do actual work by serving the client request. Therefore, to reduce the split elections and guarantee that there is an elected leader in each election term the election timeout for each node is set at randomized value and each node can give vote to only one candidate in a term. The randomized election timeout is set by method.

```
public int ComputeElectionTimeout() {  
    silencePeriod = (min + (int)(Math.random()*((max - min) + 10)))* 1000;  
    return silencePeriod;  
}
```

7.4. Append Entry Task

Once a leader is elected, it starts sending heartbeats in form of appendEntry RPC. When there is data sent from the client to the leader node it is sent over to all the nodes in the cluster through append Entry RPC. If there is no data from client, then null append entry RPC is sent.

Before sending each RPC the leader fetches the last committed log entry from the state log and sends the following details in the RPC method call –

- It's current term
- It's node name
- New data (or null if no data from client)
- Last committed term
- Last committed index

7.4.1. Responding to Append Entry RPC

Each Follower node has a flush entry and a local log. When it receives append entry RPC, it verifies the following

- If the current term (follower) \leq received term in RPC (leader)
- If the last committed term = to the flush entry term and the last committed index = to the size of the local log +1 then the flush entry is added to the local log
- And the new data entry is saved in the flush entry and send acknowledgement as true
- Else false

7.4.2. Log Replication

The log replication is done through two phase commit. When the leader receives the majority of acknowledgment from all the follower nodes it adds the data entry in the state log. This is

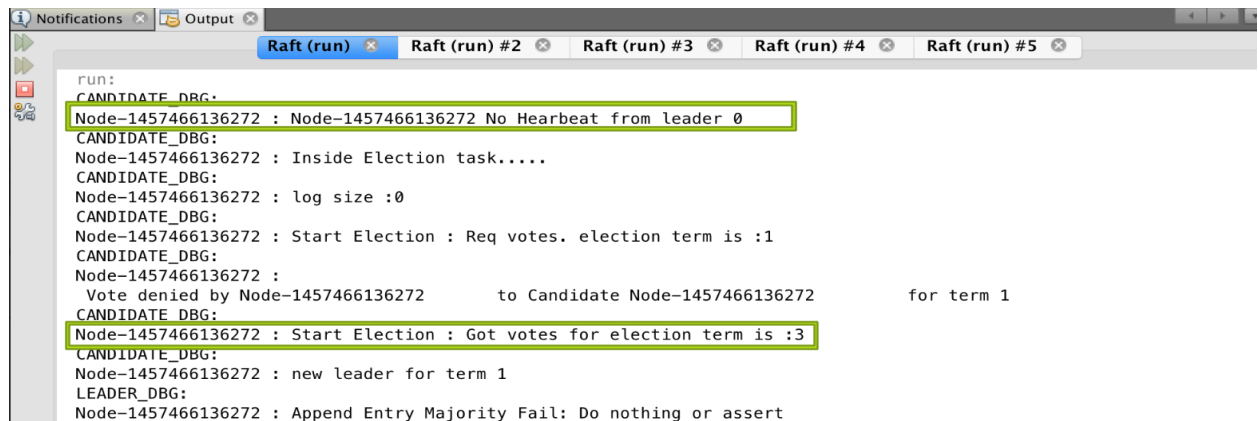
the first phase which makes sure that this entry will be added in all/majority of the local logs of the followers at some time.

When the next append entry RPC is sent over the cluster the data in the flush entry is compared again with the last committed data and it is added to the local log, thus completing the second phase of the two phase commit.

8. Test Cases

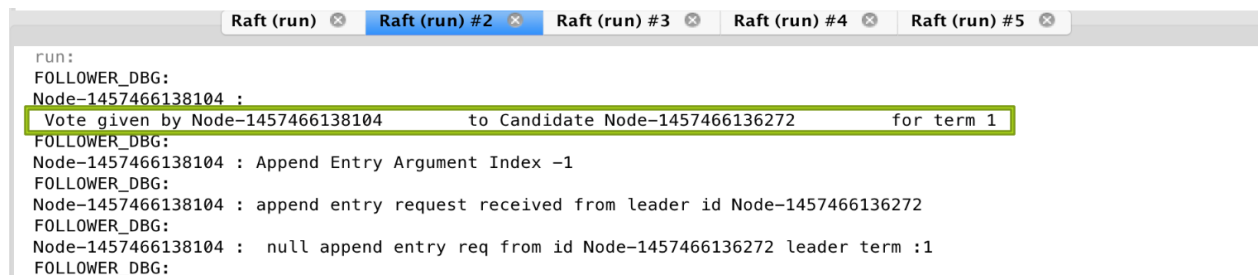
8.1. Leader election

The RMI registry is started at the default port 1099. The five nodes are started in the cluster.



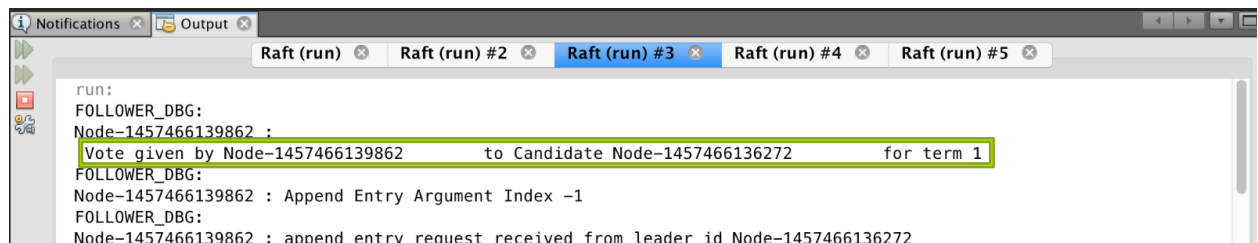
```
run:
CANDIDATE_DBG:
Node-1457466136272 : Node-1457466136272 No Hearbeat from leader 0
CANDIDATE_DBG:
Node-1457466136272 : Inside Election task....
CANDIDATE_DBG:
Node-1457466136272 : log size :0
CANDIDATE_DBG:
Node-1457466136272 : Start Election : Req votes. election term is :1
CANDIDATE_DBG:
Node-1457466136272 :
Vote denied by Node-1457466136272 to Candidate Node-1457466136272 for term 1
CANDIDATE_DBG:
Node-1457466136272 : Start Election : Got votes for election term is :3
CANDIDATE_DBG:
Node-1457466136272 : new leader for term 1
LEADER_DBG:
Node-1457466136272 : Append Entry Majority Fail: Do nothing or assert
```

Figure 1: Node 1-won leader election for term 3



```
run:
FOLLOWER_DBG:
Node-1457466138104 :
Vote given by Node-1457466138104 to Candidate Node-1457466136272 for term 1
FOLLOWER_DBG:
Node-1457466138104 : Append Entry Argument Index -1
FOLLOWER_DBG:
Node-1457466138104 : append entry request received from leader id Node-1457466136272
FOLLOWER_DBG:
Node-1457466138104 : null append entry req from id Node-1457466136272 leader term :1
FOLLOWER_DBG:
```

Figure 2: Node 2 vote grated for election term 3



```
run:
FOLLOWER_DBG:
Node-1457466139862 :
Vote given by Node-1457466139862 to Candidate Node-1457466136272 for term 1
FOLLOWER_DBG:
Node-1457466139862 : Append Entry Argument Index -1
FOLLOWER_DBG:
Node-1457466139862 : append entry request received from leader id Node-1457466136272
```

Figure 3: Node 3 vote grated for election term 3

8.2. Split election

The node 1 and node 2 received 2 votes each resulting in split election.

```

CANDIDATE_DBG: Node-1457673170346 : log size :0
CANDIDATE_DBG: Node-1457673170346 : Start Election : Req votes. election term is :1
CANDIDATE_DBG: Node-1457673170346 :
Vote denied by Node-1457673170346 to Candidate Node-1457673170346 for term 1
CANDIDATE_DBG: Node-1457673170346 : Start Election : Got votes for election term is :2
CANDIDATE_DBG: Node-1457673170346 :
Vote given by Node-1457673170346 to Candidate Node-1457673173401 for term 2
CANDIDATE_DBG: Node-1457673170346 :
Vote denied by Node-1457673170346 to Candidate Node-1457673173401 for term 3
CANDIDATE_DBG: Node-1457673170346 :
Vote denied by Node-1457673170346 to Candidate Node-1457673173401 for term 3
CANDIDATE_DBG: Node-1457673170346 :

```

Figure 4: Node 1 got two votes for election term 1,2 and 3

```

FOLLOWER_DBG: Node-1457673173401 :
Vote given by Node-1457673173401 to Candidate Node-1457673170346 for term 1
CANDIDATE_DBG: Node-1457673173401 : Node-1457673173401 No Hearbeat from leader of term 1
CANDIDATE_DBG: Node-1457673173401 : Inside Election task.....
CANDIDATE_DBG: Node-1457673173401 : log size :0
CANDIDATE_DBG: Node-1457673173401 : Start Election : Req votes. election term is :2
CANDIDATE_DBG: Node-1457673173401 :
Vote denied by Node-1457673173401 to Candidate Node-1457673173401 for term 2
CANDIDATE_DBG: Node-1457673173401 : Start Election : Got votes for election term is :2
CANDIDATE_DBG: Node-1457673173401 : Inside Election task.....
CANDIDATE_DBG: Node-1457673173401 : log size :0
CANDIDATE_DBG: Node-1457673173401 : Start Election : Req votes. election term is :3
CANDIDATE_DBG: Node-1457673173401 :

```

Figure 5: Node 2 got two votes for election term 1,2 and 3

8.3. Log Replication

Log replication over the cluster after the leader election

Leader Node

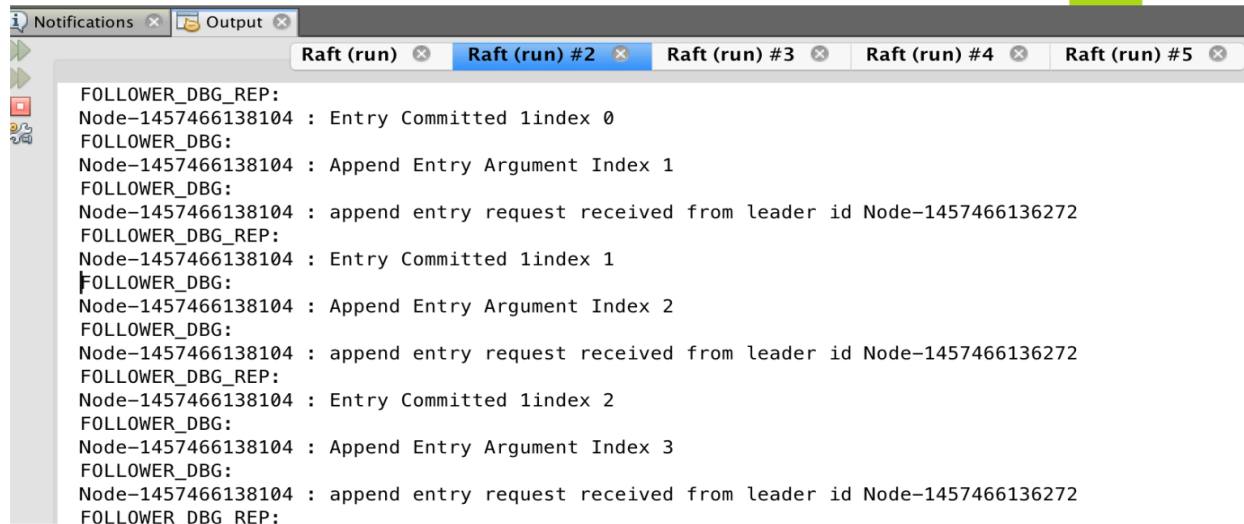


```

Raft (run) Raft (run) #2 Raft (run) #3 Raft (run) #4 Raft (run) #5
Node-1457466136272 : Append Entry Majority Fail: Do nothing or assert
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 0
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 1
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 2
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 3
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 4
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 5
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 6
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 7
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 8
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 9
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 10
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 11
LEADER_DBG_REP:
Node-1457466136272 : Entry Committed : Term 1 index 12
LEADER_DBG_REP:

```

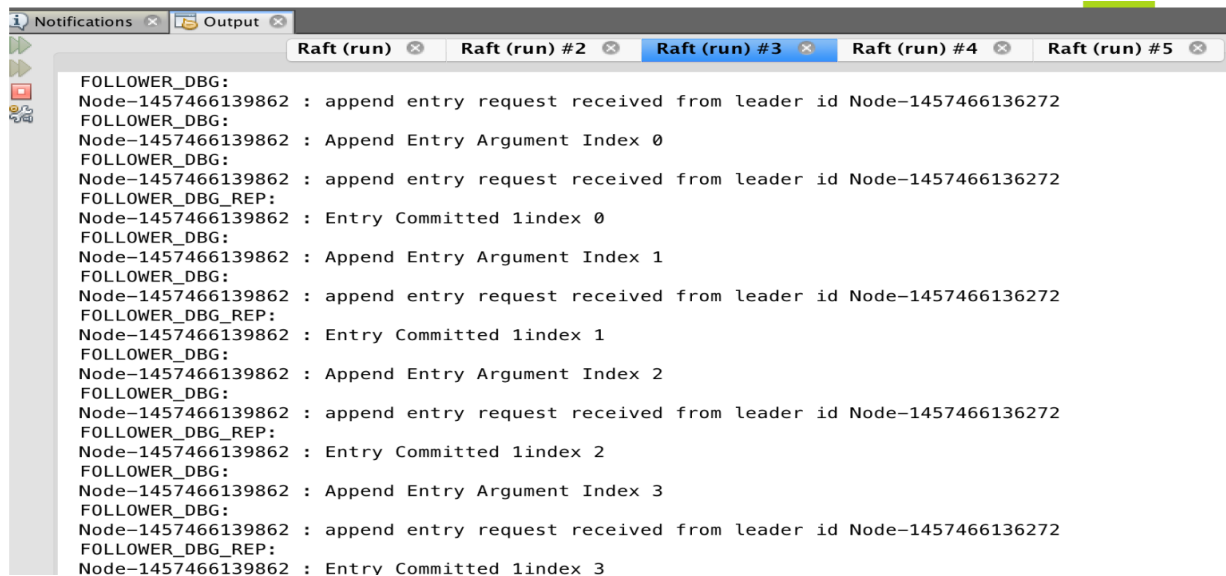
Figure 6: Leader Node committed entries in the state machine



```
Notifications x Output x
Raft (run) x Raft (run) #2 x Raft (run) #3 x Raft (run) #4 x Raft (run) #5 x

FOLLOWER_DBG_REP:
Node-1457466138104 : Entry Committed 1index 0
FOLLOWER_DBG:
Node-1457466138104 : Append Entry Argument Index 1
FOLLOWER_DBG:
Node-1457466138104 : append entry request received from leader id Node-1457466136272
FOLLOWER_DBG_REP:
Node-1457466138104 : Entry Committed 1index 1
FOLLOWER_DBG:
Node-1457466138104 : Append Entry Argument Index 2
FOLLOWER_DBG:
Node-1457466138104 : append entry request received from leader id Node-1457466136272
FOLLOWER_DBG_REP:
Node-1457466138104 : Entry Committed 1index 2
FOLLOWER_DBG:
Node-1457466138104 : Append Entry Argument Index 3
FOLLOWER_DBG:
Node-1457466138104 : append entry request received from leader id Node-1457466136272
FOLLOWER_DBG_REP:
```

Figure 7: Follower Node2 committed entries in the local machine for log replication



```
Notifications x Output x
Raft (run) x Raft (run) #2 x Raft (run) #3 x Raft (run) #4 x Raft (run) #5 x

FOLLOWER_DBG:
Node-1457466139862 : append entry request received from leader id Node-1457466136272
FOLLOWER_DBG:
Node-1457466139862 : Append Entry Argument Index 0
FOLLOWER_DBG:
Node-1457466139862 : append entry request received from leader id Node-1457466136272
FOLLOWER_DBG_REP:
Node-1457466139862 : Entry Committed 1index 0
FOLLOWER_DBG:
Node-1457466139862 : Append Entry Argument Index 1
FOLLOWER_DBG:
Node-1457466139862 : append entry request received from leader id Node-1457466136272
FOLLOWER_DBG_REP:
Node-1457466139862 : Entry Committed 1index 1
FOLLOWER_DBG:
Node-1457466139862 : Append Entry Argument Index 2
FOLLOWER_DBG:
Node-1457466139862 : append entry request received from leader id Node-1457466136272
FOLLOWER_DBG_REP:
Node-1457466139862 : Entry Committed 1index 2
FOLLOWER_DBG:
Node-1457466139862 : Append Entry Argument Index 3
FOLLOWER_DBG:
Node-1457466139862 : append entry request received from leader id Node-1457466136272
FOLLOWER_DBG_REP:
Node-1457466139862 : Entry Committed 1index 3
```

Figure 8: Follower Node3 committed entries in the local machine for log replication

9. Limitations

RMI Registry is used to invoke request vote and append entry RPC over the cluster. RMI Registry for Java is single point of failure. If the registry fails or if any of the nodes fails before unbinding from the registry, then the cluster goes down.

RMI Registry uses a default port 1099 for listening over the cluster. Therefore, it is difficult to bind a client server with the leader server. An alternative would be using socket factory in RMI.

Safety of the cluster after new leader is elected can be hampered by the ghost leader coming to life.

Using bully algorithm or Ring Algorithm can give rise to many edge cases therefore raft has its own leader election algorithm. To implement safety in this algorithm more scenario need to be tested and handled.

10.Future work and improvements

1. The raft can be implemented with peer to peer cluster using JGroups or Multicast communication. This will remove the possibility of cluster failure by removing single point of failure.
2. Connecting client to the Leader server to send real time data. And sending large amount of data over the cluster.
3. Achieving safety over the cluster to improve fault tolerance and Log replication when leader changes can be another improvement.

11. Conclusion

Raft Consensus Algorithm decomposes the issues faced in achieving consensus among distributed system into smaller sections. Thus this algorithm is more understandable than the more complex Paxos. Also, Raft used lesser RPC and makes the implementation of this algorithm easier for the developer.

12. References

1. In Search of an Understandable Consensus Algorithm - Diego Ongaro and John Ousterhout Stanford University.
2. [https://en.wikipedia.org/wiki/Consensus_\(computer_science\)](https://en.wikipedia.org/wiki/Consensus_(computer_science))
3. https://en.wikipedia.org/wiki/Leader_election.
4. [https://en.wikipedia.org/wiki/Replication_\(computing\)](https://en.wikipedia.org/wiki/Replication_(computing))
5. https://en.wikipedia.org/wiki/Fault_tolerance
6. <http://stackoverflow.com/questions/5658929/what-is-rmi-registry>
7. <https://en.wikipedia.org/wiki/JGroups>
8. <https://www.quora.com/What-are-the-differences-between-RMI-and-Sockets>
9. Demo and animation <http://thesecretlivesofdata.com/raft/>

10. RAFT lecture

https://www.youtube.com/watch?v=YbZ3zDzDnrw&index=2&list=PLbCwFjjNexB0SR8XwIYD_wO0Njj7aCRhY

11. All about RAFT <https://raft.github.io/>

12. BOLOSKY, W. J., BRADSHAW, D., HAAGENS, R. B., KUSTERS, N. P., AND LI, P.

Paxos replicated state machines as the basis of a high-performance data store. In Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation.

13. BURROWS, M. The Chubby lock service for loosely- coupled distributed systems. In Proc. OSDI'06, Symposium on Operating Systems Design and Implementation

14. RMI Registry <https://docs.oracle.com/javase/7/docs/api/java/rmi/registry/Registry.html>

13. Appendix

1. The github link to the code

<https://github.com/Shilpita/COEN317>

2. The presentation link

<https://drive.google.com/a/scu.edu/file/d/0Bxq-jOiY35R0S3VyUUluWHJZY3c/view?usp=sharing>