

Mini-Project Report

Implementation of Raft Consensus Algorithm

Distributed Computing Systems CO367



**National Institute of Technology Karnataka
Surathkal**

Date: 17/4/17

Submitted To: Dr. Annappa

Group Members:

14CO133	Prajwala TM
14CO246	T. Guru Pradeep Reddy

Table of Contents

Abstract	2
1. Introduction	3
2. Implementation	3
3. Code	6
4. Results and Output	10
5. Conclusion and Future Work	12
6. References	12

Abstract

The main aim of this project is to implement a consensus algorithm named Raft as described in the paper: “*In Search of an Understandable Consensus Algorithm*”.

Raft, is a distributed consensus algorithm for managing replicated logs. Results obtained using the Raft consensus algorithm are similar to those obtained using Paxos, with similar efficiency, but however, its structure is quite different. Raft provides better understandability and provides a better platform for building practical systems.

This is achieved by separating the key elements of consensus- Leader Election, Log Replication and Safety and provides a better degree of coherency by reducing the number of states. Raft also has an upperhand over the existing consensus algorithms by including *strong leadership*, randomised timers for *leader election* and simplification of the *membership changes* process as its novel features.

The algorithm has been implemented using the Go-Programming Language. Raft is implemented as a Go instance, with associated methods, meant to be used as a module in a larger service. A set of Raft instances talk to each other with RPC to maintain replicated logs. The implementation provides detailed insights into the working of the main aspects of Raft, and successfully passes a wide variety of test cases, including exceptional conditions.

1. Introduction

A replicated service (e.g., key/value database) achieves fault tolerance by storing copies of its data on multiple replica servers. Replication allows the service to continue operating even if some of its servers experience failures (crashes or a broken or flaky network). The challenge is that failures may cause the replicas to hold differing copies of the data.

Raft manages a service's state replicas, and in particular it helps the service sort out what the correct state is after failures. Raft implements a replicated state machine. It organizes client requests into a sequence, called the log, and ensures that all the replicas agree on the contents of the log. Each replica executes the client requests in the log in the order they appear in the log, applying those requests to the replica's local copy of the service's state. Since all the live replicas see the same log contents, they all execute the same requests in the same order, and thus continue to have identical service state. If a server fails but later recovers, Raft takes care of bringing its log up to date. Raft will continue to operate as long as at least a majority of the servers are alive and can talk to each other. If there is no such majority, Raft will make no progress, but will pick up where it left off as soon as a majority can communicate again.

2. Implementation

Raft is implemented as a Go object type with associated methods, meant to be used as a module in a larger service. A set of Raft instances talk to each other with RPC to maintain replicated logs. The Raft interface supports an indefinite sequence of numbered commands, also called log entries. The entries are numbered with *index numbers*. The log entry with a given index will eventually be committed. At that point, Raft sends the log entry to the larger service for it to execute.

Raft decomposes the consensus problem into three relatively independent problems:

1. **Leader election:** A new leader must be chosen when an existing leader fails.
2. **Log replication:** The leader must accept log entries from the clients, and replicate them across the cluster, forcing the other logs to agree with its own.

3. **Safety:** The key to Safety is the State Machine Safety property which says that, if a server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index.

Raft cluster contains 3-5 servers (here, 3) and each of the servers is always in one of the three states: Leader, follower, or candidate. There is one leader at any point of time, and followers are passive responding only to leaders and candidates. Candidates are used during leader election. The leader handles all the client requests. Time is divided into terms, where each term is numbered with consecutive integers. Each term begins with an election, and re-elections are carried out in the next term in case of split votes.

RequestRPCs and AppendEntries RPCs are the two main RPCs used for communication among the raft servers.

2.1 Leader Election

A heartbeat mechanism is used to trigger elections. All servers start as followers, and remain in the same state till they receive valid RPCs from the leader. If a follower does not receive an RPC from a leader for more than the election timeout period, it begins an election assuming there is no viable leader. This is done by incrementing its term , and changing to candidate state. It votes for itself and then sends *RequestVoteRPCs* in parallel to other servers, and the follower receiving the majority of the votes becomes the leader. Voting is done based on a first-come-first-served-basis. Once a candidate wins, it changes into the leader state and sends *heartbeat RPCs (AppendEntriesRPCs with blank entries)* to the other servers, which then convert to follower state. Another important condition is that the candidate's term must be higher than the server's term for vote to be granted.

In case of a split vote, re-elections take place in the next term. Raft makes use of randomised timers to ensure that this occurs rarely.

2.2 Log Replication

The elected leader receives client requests, which contain commands to be executed on the replicated state machines. The leader appends the command to its log entry, and then issues

AppendEntriesRPCs in parallel, to all the servers, for replication. When the entry has been safely applied, the leader applies it to the state machine and returns the request to the client.

Each log contains the *Command*, and the *Term* when it was received by the leader. Once the command is replicated on a majority of servers, it is committed. This *committedIndex* is also updated, and once an entry is committed, there is no going back and all entries presuming it are also committed. The coherency is maintained using the *Log Matching Property*:

1. If two entries in different logs have the same index and term, then they store the same command.
2. If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

The inconsistencies in the follower's logs are removed by the leader by making them follow its own entries. This is done using the *nextIndex* and *matchIndex* attributes, where, all the entries after that point are deleted by the leader.

2.2 Safety

Just leader election and Log replication are not quite sufficient to ensure that each state machine executes exactly the same commands in the same order. For example, a follower might be unavailable while the leader commits several log entries, then it could be elected leader and overwrite these entries with new ones; as a result, different state machines might execute different command sequences. So we need to impose additional restriction on which servers may be elected as Leader

2.2.1 Election Restriction

The leader has to eventually store all the committed log entries. If the leader is elected even if it doesn't have all the previous committed entries then we have to some mechanism by which we should be able to send those entries to the newly elected leader. Raft uses a simpler approach where it guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election, without the need to transfer those entries

to the leader. This means that log entries only flow in one direction, from leaders to followers, and leaders never overwrite existing entries in their logs. While voting only it prevents the candidate from winning elections if it doesn't have all committed entries. This can be done by checking if candidate's log is at least up-to-date as any other log in that majority.

In *SendRequestVote* RPC we send the candidate *LastLogTerm* and *LastLogIndex* . We ensure that it is up date by using two conditions. At least one of them should hold true.

1. If the *LastLogTerm* is greater than Term of the last committed entry of the peer, then we can say that candidate is more upto dated.
2. If the *LastLogTerm* and Term are equal then *LastLogIndex* of candidate should be greater than equal to index of last committed entry of peer.

2.2.2 Committing entries from previous terms

A leader knows that an entry from its current term is committed once that entry is stored on a majority of the servers. If a leader crashes before committing an entry, future leaders will attempt to finish replicating the entry. However, a leader cannot immediately conclude that an entry from a previous term is committed once it is stored on a majority of servers. Because that may be overwritten by future Leader. To avoid this problem raft never commits the log entries from previous terms based on the replica count. Only leader's current term and committed this may. Sometimes leader could safely conclude that older log entries can be committed, but raft takes more conservative approach.

3. Code

The implementation in Go is explained below:

The heartbeat interval is set to 50ms.

```
const (  
HBINTERVAL = 50* time.Millisecond  
)
```

Each raft peer is implemented as a struct defined below:

```
type Raft struct {  
  
    mu    sync.Mutex // mutex lock for secure access to peers state  
    peers []*labrpc.ClientEnd //RPC endpoints of raft peers  
    me     int // this peer's index into peers[]  
  
    //Persistent state  
    currentTerm int  
    votedFor    int  
    log         []LogEntry  
  
    //Volatile state of all servers  
    commitIndex int  
    lastApplied int  
  
    //Leader state  
    nextIndex []int  
    matchIndex []int  
  
    isLeader bool  
    isCandidate bool  
    isFollower bool  
  
    chanApply chan ApplyMsg  
    chanLeader chan bool  
    chanCommit chan bool  
    chanAppendEntry chan bool  
    chanGrantVote chan bool  
  
    vote_count int  
}
```

The mutex locks are used to provide mutual exclusion access to the go routines and methods for the Raft peer attributes , in order to implement coherency.

Go-routines are light-weight threads used for concurrently execution. Here, they are used for sending the RPCs and other important operations in parallel.

The Go channels are used to send values from one go-routine to another.

chanApply: To notify that an ApplyMsg should be sent to the service on the same server when log entries are committed.

chanLeader: To notify the initialisation of the leader state after elections

chanCommit: Set to true when an entry is committed and applymsg struct has to be initialised

chanAppendEntry: To notify a heartbeat/log entry has been sent

chanGrantVote: To indicate whether vote has been granted or not

RPC Handler sending RequestVote RPCs and identifying the leader:

```
func (rf *Raft) sendRequestVote(server int, args RequestVoteArgs,
reply *RequestVoteReply) bool {
    ok := rf.peers[server].Call("Raft.RequestVote", args, reply)
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if ok {
        term := rf.currentTerm
        if rf.isCandidate==true{
            return ok
        }
        if args.Term != term {
            return ok
        }
        if reply.Term > term {
            rf.currentTerm = reply.Term
            rf.isFollower=true
            rf.votedFor = -1
            rf.persist()
        }
        if reply.VoteGranted {
            rf.voteCount++
            if rf.isCandidate==true && rf.voteCount >
len(rf.peers)/2 {
                rf.isLeader=true
            }
        }
    }
}
```

```

        rf.chanLeader <- true
    }
}
}
return ok
}

```

RPC handler sending log entries and heartbeats after heartbeat timeout to all the raft peers in parallel:

```

func (rf *Raft) sendAppendEntries(server int, args
AppendEntriesArgs, reply *AppendEntriesReply) bool {
    ok := rf.peers[server].Call("Raft.AppendEntries", args, reply)
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if ok {
        if rf.state != STATE_LEADER {
            return ok
        }
        if args.Term != rf.currentTerm {
            return ok
        }

        if reply.Term > rf.currentTerm {
            rf.currentTerm = reply.Term
            rf.isFollower=true
            rf.votedFor = -1
            rf.persist()
            return ok
        }
        if reply.Success {
            if len(args.Entries) > 0 {
                rf.nextIndex[server] =
args.Entries[len(args.Entries) - 1].LogIndex + 1
                //reply.NextIndex
                //rf.nextIndex[server] = reply.NextIndex
                rf.matchIndex[server] = rf.nextIndex[server] -
1
            }
        } else {
            rf.nextIndex[server] = reply.NextIndex

```

```

    }
}
return ok
}

```

4. Results and Output

To make the algorithm is working properly, it was has been run with various test cases. Those Test cases are described below.

1. *Initial Election*

Here we configure three servers, then we will for some time so that a leader will be elected. If leader is not elected during this time interval, error will be displayed. Then we check for uniqueness of the leader. After that we compare the term and the term after waiting for election time out duration. Both should be same, because in the absence of network failure leader and term should remain same.

2. *Election After Network Failure*

Here we will disconnect the leader, then it should elect a new leader and when again the old leader is added it shouldn't affect anything. This condition is tested, after that we disconnect multiple servers. If the no of servers disconnected are more than half, then a new leader shouldn't be elected as it needs majority votes to become a leader.

3. *Basic Agreement*

Here we check if there are any committed entries before the raft has even started and we match the index of elements with the index values from other servers.

4. *Agreement Despite Follower Disconnection*

Here we disconnect one of the follower. Then we send a commit entry to the leader which in turn broadcasts this request to all the connected peers. Here we check for complete agreement. It indirectly checks that the servers agree on same value.

5. *No agreement If too many Followers Disconnect*

In the above scenario only server is disconnected, so it's still possible to come to complete agreement. However, if majority of servers are disconnected, the entries shouldn't be committed.

6. *Concurrent Start*

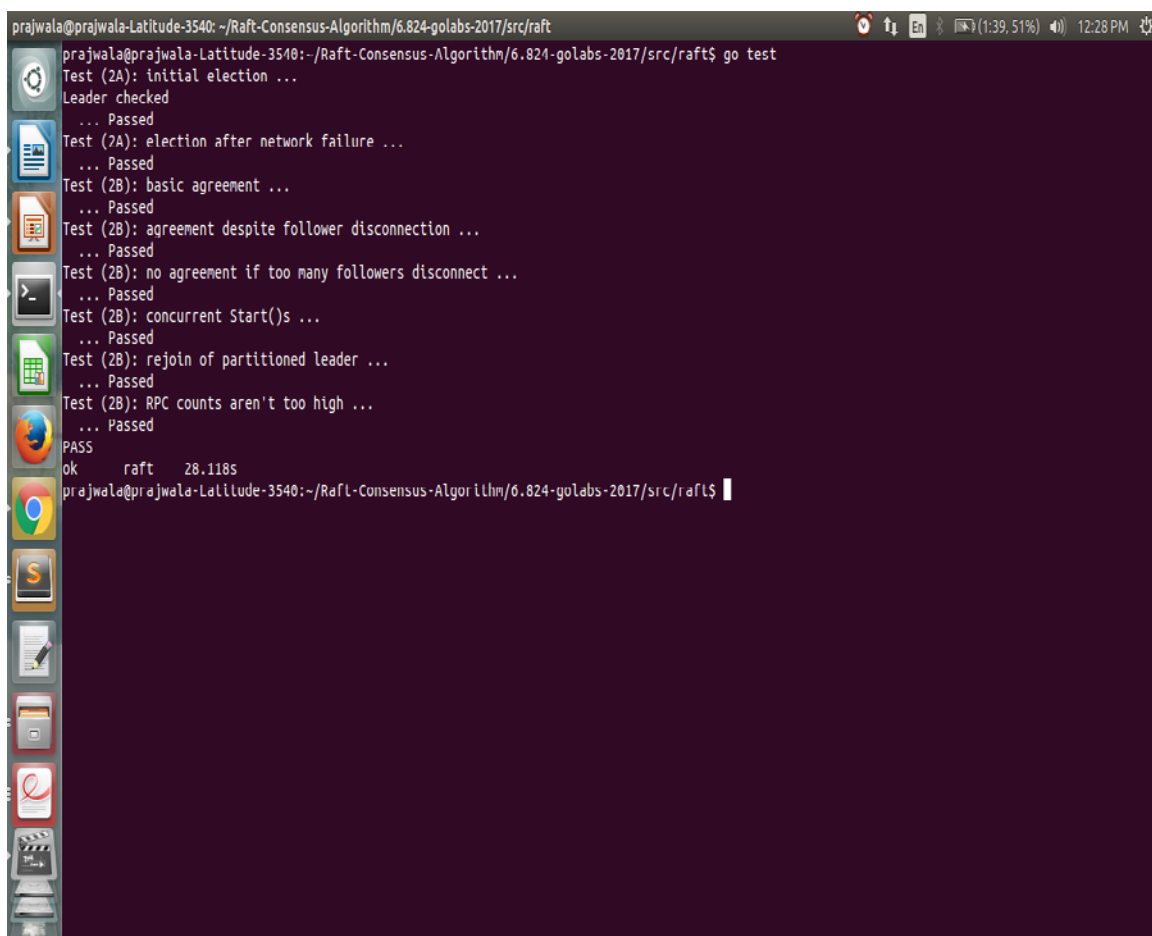
This test is performed to check the independence of the start of each of the raft peers, and how quick the *Start()* function call returns. Thus, implementation is carried out concurrently using threads in the form of GoRoutines.

7. *Rejoin of Partitioned Leader*

This is to test how Raft functions in case of network partitioning, where two leaders exist, one for each partition. The two leaders are unaware of the existence of the other and try to reach out to the majority of the servers by sending out *AppendEntriesRPCs*. The system safely recovers during the rejoin of the network partition, with one effective final leader.

8. *RPC counts are not too high*

Network congestion is one of the main factors which is responsible for effective functioning of the consensus of the distributed client-server system. In order to prevent the same, this check is performed to limit the number of RPCs sent out on the network.



```
prajwala@prajwala-Latitude-3540: ~/Raft-Consensus-Algorithm/6.824-golabs-2017/src/raft
prajwala@prajwala-Latitude-3540:~/Raft-Consensus-Algorithm/6.824-golabs-2017/src/raft$ go test
Test (2A): initial election ...
Leader checked
... Passed
Test (2A): election after network failure ...
... Passed
Test (2B): basic agreement ...
... Passed
Test (2B): agreement despite follower disconnection ...
... Passed
Test (2B): no agreement if too many followers disconnect ...
... Passed
Test (2B): concurrent Start()s ...
... Passed
Test (2B): rejoin of partitioned leader ...
... Passed
Test (2B): RPC counts aren't too high ...
... Passed
PASS
ok      raft    28.118s
prajwala@prajwala-Latitude-3540:~/Raft-Consensus-Algorithm/6.824-golabs-2017/src/raft$
```

5. Conclusion and Future Work

With reference to the paper, have implemented the Raft consensus algorithm successfully in Go, ensuring that it passes all the exceptional and corner cases also, apart from the main ones. It includes three main features- Leader Election, Log Replication and Safety. Further work can be carried out to implement persistence, for enabling Raft to safely survive a reboot and start where it left off.

6. References

1. Paper link :<https://raft.github.io/raft.pdf>
2. Reference link for Raft in Go: <https://pdos.csail.mit.edu/6.824/labs/lab-raft.html>
3. Guidelines: <https://raft.github.io/>
4. Official video for Raft: <https://www.youtube.com/watch?v=LAqyTyNUYSY>
5. Link for Go documentation: <https://golang.org/>
6. Go tutorials: <https://www.tutorialspoint.com/go/>