Cloud Computing

Mini Project Report

Implementing Raft Logic in Go

Submitted By:

VI Semester Section _J

CHANDAN KUMAR S	**PES2UG20CS804**
VIJAY J	**PES2UG20CS815**
YUVARAJ S	**PES2UG20CS819**
CHINMAY GOWDA	**PES2UG20CS902**

Under the guidance of

Prof. GURURAJ P

January - May 2023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

Electronic City, Hosur Road, Bengaluru – 560 100, Karnataka, India

SHORT DESCRIPTION AND SCOPE OF THE PROJECT

Implementing Raft Logic in Go is a project that aims to build a distributed consensus algorithm using the Raft protocol in the Go programming language. The Raft protocol is a widely-used consensus algorithm for distributed systems that ensures fault tolerance and data consistency in a cluster of servers.

The project scope includes designing and implementing the Raft protocol in Go, including leader election, log replication, and safety properties. The implementation will include the basic components of a Raft-based distributed system, such as a client, a server, and a log. The project will also involve testing and benchmarking the Raft implementation to ensure its correctness and performance.

The project requires a strong understanding of distributed systems, consensus algorithms, and the Go programming language. It will also require familiarity with network programming and testing frameworks. The outcome of the project will be a functional implementation of the Raft protocol in Go, which can be used as a building block for distributed systems that require fault tolerance and consistency guarantees.

Methodology

- The Raft algorithm is a distributed consensus protocol designed to allow a cluster of nodes to maintain a replicated log and to agree on a consistent order of updates to that log. The goal of the Raft algorithm is to ensure that the nodes in the cluster agree on the state of the system even in the presence of failures.

- Raft Node: A Raft node is a single instance of a server that participates in the Raft protocol. It communicates with other nodes in the cluster to elect a leader and replicate the log. The Raft node is responsible for persisting its state and maintaining the consistency of the cluster.

- Raft Cluster: A Raft cluster consists of a group of nodes that communicate with each other to maintain consensus. The cluster is responsible for coordinating leader election and log replication.

- Raft Election Logic: The Raft election logic implements the leader election algorithm. Nodes in the cluster vote for a leader based on a set of rules, including election timeouts and candidate conflicts. The election logic also handles the case where a node detects that the current leader has failed or become unreachable.

- Raft Leader Logic: The Raft leader logic is responsible for managing log replication. The leader receives log entries from clients and sends them to other nodes in the cluster. The leader ensures that log entries are replicated on a quorum of nodes before considering them committed.

- Raft RPC Handlers: Raft RPC handlers implement the RPC protocol used by nodes to communicate with each other. The RPC handlers receive messages from other nodes and route them to the appropriate logic for processing.

- Node Logs: Each node maintains a log of commands that have been committed. When a new leader is elected, it sends its log to the other nodes in the cluster to ensure consistency.

Testing

TEST CASE-1

```
package raft

import (
	"testing"
)

func Test1(t *testing.T) { // Simple Leader Election

	cluster := NewCluster(t, 5)
	defer cluster.Shutdown()

	sleepMs(3000) // Wait for a leader to be elected
```

```
        firstLeaderId := cluster.getClusterLeader()
        cluster.DisconnectPeer(firstLeaderId)

        secondLeaderId := cluster.getClusterLeader()
        cluster.DisconnectPeer(secondLeaderId)

        thirdLeaderId := cluster.getClusterLeader()
        cluster.DisconnectPeer(thirdLeaderId)

        sleepMs(3000)

        // Fails, no leader present
        cluster.getClusterLeader()
        sleepMs(3000)

}
```

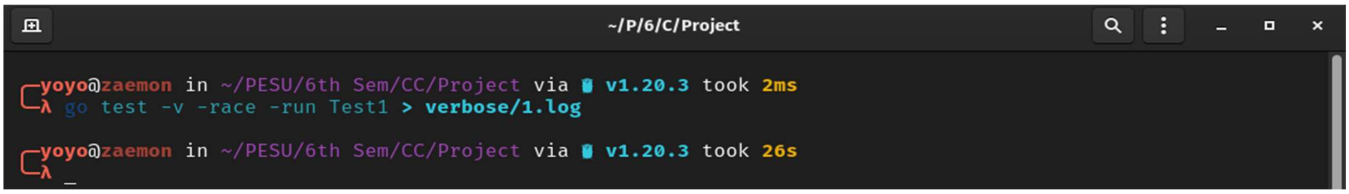
TEST CASE-2

```
func Test2(t *testing.T) {
        /* Replication failure scenario: Leader drops after committing, comes back later*/

        cluster := NewCluster(t, 5)
        defer cluster.Shutdown()

        // ReceiveClientCommand a couple of values to a fully connected nodes.
        origLeaderId := cluster.getClusterLeader()
        cluster.SubmitClientCommand(origLeaderId, "Set X = 5")
        cluster.SubmitClientCommand(origLeaderId, "Set X = 1000")

        sleepMs(3000)

        // Leader disconnected...
        cluster.DisconnectPeer(origLeaderId)

        // ReceiveClientCommand 7 to original leader, even though it's disconnected.
Should not reflect.
        cluster.SubmitClientCommand(origLeaderId, "Set X = X-5")

        newLeaderId := cluster.getClusterLeader()
```

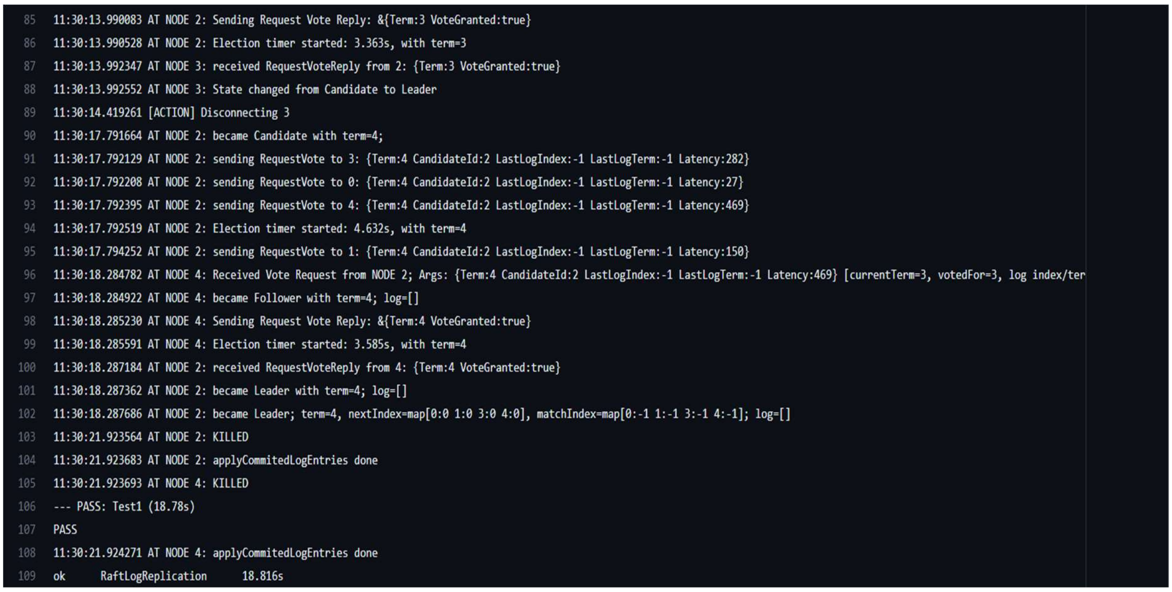
```
        // ReceiveClientCommand 8.. to new leader.
        cluster.SubmitClientCommand(newLeaderId, "Set X = X+10")
        cluster.SubmitClientCommand(newLeaderId, "Set X = X+1")
        cluster.SubmitClientCommand(newLeaderId, "Set Y = 5")
        cluster.SubmitClientCommand(newLeaderId, "Set Y = X+Y")
        cluster.SubmitClientCommand(newLeaderId, "Set Y = Y+3")
        cluster.SubmitClientCommand(newLeaderId, "Set Z = -1")
        sleepMs(3000)

        // ReceiveClientCommand 9 and check it's fully committed.
        cluster.SubmitClientCommand(newLeaderId, "Set Z = 3")
        sleepMs(3000)

        cluster.ReconnectPeer(origLeaderId)
        sleepMs(15000)
}
```

Results

Screenshot1



Screenshot2



Screenshot3

```
~/P/6/C/Project

╭─yoyo@zaemon in ~/PESU/6th Sem/CC/Project via  v1.20.3 took 32ms
╰─λ go test -v -race -run Test2 > verbose/2.log

╭─yoyo@zaemon in ~/PESU/6th Sem/CC/Project via  v1.20.3 took 37s
╰─λ _
```

Screenshot4

```
11:34:02.565398 AT NODE 0: Election timer started: 4.01s, with term=2
11:34:02.608014 AT NODE 2: Received AppendEntries from NODE 0; args: {Term:1 LeaderId:0 PrevLogIndex:1 PrevLogTerm:1 Entries:[{Command:Set X = X-5 Term:1}] LeaderCommit:1 Late
11:34:02.608105 AT NODE 2: Sending AppendEntries reply: {Term:2 Success:false}
11:34:02.864138 AT NODE 1: Received AppendEntries from NODE 0; args: {Term:1 LeaderId:0 PrevLogIndex:1 PrevLogTerm:1 Entries:[{Command:Set X = X-5 Term:1}] LeaderCommit:1 Late
11:34:02.864220 AT NODE 1: Sending AppendEntries reply: {Term:2 Success:false}
11:34:02.865780 AT NODE 3: Received AppendEntries from NODE 0; args: {Term:1 LeaderId:0 PrevLogIndex:1 PrevLogTerm:1 Entries:[{Command:Set X = X-5 Term:1}] LeaderCommit:1 Late
11:34:02.865877 AT NODE 3: Sending AppendEntries reply: {Term:2 Success:false}
11:34:02.981697 AT NODE 2: sending AppendEntries to 0: currentPeer_nextIndex=2, args={Term:2 LeaderId:2 PrevLogIndex:1 PrevLogTerm:1 Entries:[{Command:Set X = X+10 Term:2} {Co
11:34:03.444347 AT NODE 0: Received AppendEntries from NODE 2; args: {Term:2 LeaderId:2 PrevLogIndex:1 PrevLogTerm:1 Entries:[{Command:Set X = X+10 Term:2} {Command:Set X = X+
11:34:03.444727 AT NODE 0: Log is now: [{Set X = 5 1} {Set X = 1000 1} {Set X = X+10 2} {Set X = X+1 2} {Set Y = 5 2} {Set Y = X+Y 2} {Set Y = Y+3 2} {Set Z = -1 2} {Set Z = 3
11:34:03.444953 AT NODE 0: Sending AppendEntries reply: {Term:2 Success:true}
11:34:03.446314 AT NODE 2: AppendEntries reply from NODE 0 success: nextIndex := map[0:9 1:9 3:9 4:9], matchIndex := map[0:8 1:8 3:8 4:8]
11:34:17.397543 AT NODE 0: KILLED
11:34:17.397634 AT NODE 0: applyCommitedLogEntries done
11:34:17.397662 AT NODE 1: KILLED
11:34:17.397764 AT NODE 1: applyCommitedLogEntries done
11:34:17.397826 AT NODE 2: KILLED
11:34:17.397875 AT NODE 2: applyCommitedLogEntries done
11:34:17.397994 AT NODE 3: KILLED
11:34:17.398132 AT NODE 3: applyCommitedLogEntries done
11:34:17.398233 AT NODE 4: KILLED
11:34:17.398297 AT NODE 4: applyCommitedLogEntries done
--- PASS: Test2 (32.29s)
PASS
ok      RaftLogReplication      32.313s
```

Conclusions

Project Implementing Raft Logic in Go using consensus algorithm is a successful implementation of the Raft consensus algorithm. The project demonstrates the ability of the Raft algorithm to ensure consistency in a distributed system by replicating the log and agreeing on the order of updates. The Raft algorithm is implemented using Go programming language and is designed to handle network partitions and node failures.

The project includes components such as Raft election logic, leader logic, RPC handlers, and the Raft node and cluster infrastructure. These components work together to elect a leader and replicate the log on a quorum of nodes. The project also includes tests that verify the correct behavior of the Raft algorithm in various scenarios, including network partitions and node failures.

Overall, the Project Implementing Raft Logic in Go using consensus algorithm is a valuable contribution to the field of distributed systems and provides a reliable framework for developing distributed systems that require consensus among a group of nodes.