Mandatory Hand-in 6: **Raft**

Raft Repository https://github.com/Ashniu123/raft-grpc

1. Introduction

The code in the github repository implements the management of a distributed server for saving logs using Raft algorithm. The code is organized as follow:

- within /internal/impl/raft-service.go we can find the implementation of the main gRPC functions exposed by each peer to implement Raft algorithm.
- we can find the gRPC function declaration in the raft.proto file in the /internal/proto-files folder.
- the peer is initialized as a server in /cmd/main.go and then using the newClient function in /cmd/server/server.go it connects to the cluster (set of the other peers).
- once our peer is ready (exposes gRPC services and is connected to the other peers) everything takes place in the /internal /impl/raft-service.go file.

Because each peer is a separate process, we must start each one by passing the IP and port when the program starts. We must also specify a server node to connect to, as shown below:

- first peer *go run ./cmd/main.go -id=1 -addr=localhost:20001*

- following peers *go run ./cmd/main.go -id=2 -addr=localhost:20002 -join=localhost:20001*

2. Go-specific features

- **Go routines**

We can find many go routines, which are lightweight threads of execution that can run concurrently with other parts of the program, within our code.
For example, we find the creation of a go routine at the time the peer is created, when it listens on the specified port. This allows the peer to remain listening while communicating with other peers.
Also all nodes use an active goroutine called runElectionTimer used to check whether the time elapsed is more than the timeout and will then invoke a new election of leader. This is of course only true for nodes in a follower or candidate state.
If the node is the leader, it will use a goroutine heartbeatWrapper to send constant heartbeats. This go routine is created when the peer is elected as a master and ends when he is no longer the leader.

- **Channels**

In this system channels are primarily used for waiting. For example, the server struct contains a field called ready which is in fact a channel sending and receiving booleans. When a new server is instantiated, this channel is used by awaiting a message to be received by it (see appendix 5.1). In this case the system is just waiting for true to be sent to the channel which indicates that the function new which creates the server, has now finished and the function server has been invoked.
In this way we can ensure an order in the concurrent execution of functions.
We also find the use of channels in the management of time intervals, where, when the time expires, a signal will be sent on the channel associated to the timer (see runElectionTimer and variable ticker).

- **Mutex**

The system uses mutex to protect shared resources which in this case are variables containing values like the current term, the last election time, the list of peers and the state of each node. This is done to create fail tolerance for writing and reading at the same time in the shared resource, to prevent wrong readings and therefore prevent wrong elections. We need to protect the list of peers since a new peer could join while another peer will try and access the list of available peers. (see appendix 5.2)

3. Communication method

The implementation uses gRPC with three defined services to communicate:

- *rpc appendEntries(AppendEntriesRequest) returns (AppendEntriesResponse)*

AppendEntries is invoked by Leader to replicate log entries and also used as heartbeat. In this function we see how the peer checks the received packet with its term and index values. Because this function is also invoked by heartbeat function, the value of lastElectionTime is updated so the search for the new leader is not triggered.

- *rpc requestVote(RequestVoteRequest) returns (RequestVoteResponse)*

The service requestVote is invoked by Candidates to collect votes from followers when an election has started. The message RequestVoteRequest is sent with the it and contains information about the candidate's term, id, last log index and last log term. The follower will then check if the term is correct. If it is lower, it will deny the vote. If it is higher, it will update its own term. It will then check if the candidate's log is up to date. If it is, it will grant the vote if it has not already granted it to another candidate. Finally, if the candidate obtains a quorum, it will be auto elected as the new leader.

- *rpc joinCluster(JoinClusterRequest) returns (JoinClusterResponse)*

The service joinCluster is a helper gRPC service that is used to add new peers to the existing cluster. It is called by the new nodes and the nodes in the cluster will then add the new node to the cluster. This is done by appending the new nodes id and address into the peers list which is mutex protected.

4. Properties guaranteed by the protocol implementation

Properties ensured by Raft are:

- **Election safety:** at most one leader can be elected in a given term.

When a peer is asked to vote, it checks that the term is higher than the current term and it has not already expressed a preference. In this way, for each election (identified by the term) peers will express only one preference at most, and only the one who exceeds half will become leader.

- **Leader append-only:** a leader can only append new entries to its logs (it can neither overwrite nor delete entries).

This is achieved by only making it possible to append to logs in the code and not modify old logs.

- **Log matching:** if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

The gRPC function appendEntries checks if there is a conflict between the existing log entry and the one provided from the leader. This check is essential to maintain log consistency. However, in the case where `term` and `index` are already equal, it implies that the log is already aligned, and there is no conflict to resolve. Instead, in case of a conflict, the function takes corrective action to align the follower's log with the leader's log

- **Leader completeness:** if a log entry is committed in a given term then it will be present in the logs of the leaders since this term.

This is achieved by making sure a candidate does not have a stale log. This is done by the followers checking whether the candidate logs last entry has a higher or equal term to the current term. In the code in appendEntries the commit index is also checked, which must be equal to the minimum between the one committed by the leader and the one in the node (see appendix 5.3 & 5.4).

- **State machine safety:** 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.

In this specific case, the code that is responsible for forwarding the log from the server to the client is missing (there is no external client part but only inter-peer management). In fact, the server should commit to the client only after receiving the ok from at least half of the peers, to ensure that only one server (in case of network partition) can confirm to the client.

5. Appendix

5.1

```
103
104             go func() {
105                     <-ready
106                     rs.mx.Lock()
107                     rs.LastElectionTime = time.Now()
108                     rs.mx.Unlock()
109                     rs.runElectionTimer()
110             }()
```

NewRaftService function in **internal/impl/raft-service.go**

5.2

```
366             rs.mx.Lock()
367             term := rs.CurrentTerm
368             rs.mx.Unlock()
```

Use of mutex to access variables that can be modified concurrently

5.3

```
170             // 5. If leaderCommit > commitIndex, set commitIndex =min(le
171             if in.LeaderCommit > rs.CommitIndex {
172                     if in.LeaderCommit <= rs.LastApplied {
173                             rs.CommitIndex = in.LeaderCommit
174                     } else {
175                             rs.CommitIndex = rs.LastApplied
176                     }
177             }
```

Handles new leader commits

5.4

```
207             if in.Term < rs.CurrentTerm {
208                     log.Printf("Node-%v's RequestVoteResponse: {term: %+v, votegranted: %v}", rs.NodeID, resp.Term, resp.VoteGranted)
209                     return
210             }
```

Function returns without setting voteGranted = true