Mandatory Handin 6 - RAFT implementations

Nicolaj Heinrich Pedersen (nihp) & Mikkel Bistrup Andersen (mbia)

December 2022

`https://github.com/etcd-io/raft`

IT-University of Copenhagen

Contents

1 Implementation

We've chosen to take a look at github.com/etcd-io/raft, a golang implementaion of raft made by the etcd, a Cloud Native Computing Foundation project. the main raft.go class is implemented to be faithful to the protocol implementation, although only implementing the raft algorithm, and leaving out the network and disk IO, which is left to the user to choose and implement. The primary object in raft is the node, which in this case is defined in node.go. Messages between the peer nodes in the cluster/clusters and the raft algorithm is handled using rpc in a protocol buffer package called "raftpb"(raft readme). The main raft.go algorithm implements the full functionality of the raft protocol, with a barebone implementation of nodes, mainly used for testing and trial usage of the algorithm.

This implementation uses a standard leader election using timeouts in the form of a heartbeat timeout and an election timeout, as seen in the figure below, where they are declared in the raft struct at line 313 & 314 in raft.go

```
313             heartbeatTimeout int
314             electionTimeout  int
```

Once a timeout happens, an election to find a new leader happens, in this implementation the method which handles the core of this part is the "campaign" func, which handles the election and distribution of roles to the nodes as leader and replicas.

```
// campaign transitions the raft instance to candidate state. This must only be
// called after verifying that this is a legitimate transition.
func (r *raft) campaign(t CampaignType) {
        if !r.promotable() {
                // This path should not be hit (callers are supposed to check), but
                // better safe than sorry.
                r.logger.Warningf("%x is unpromotable; campaign() should have been called", r.id)
        }
        var term uint64
        var voteMsg pb.MessageType
        if t == campaignPreElection {
                r.becomePreCandidate()
                voteMsg = pb.MsgPreVote
                // PreVote RPCs are sent for the next term before we've incremented r.Term.
                term = r.Term + 1
        } else {
                r.becomeCandidate()
                voteMsg = pb.MsgVote
                term = r.Term
        }
```

This includes the functionality of dealing with split votes, using randomly set timeouts for each node, such that it in practice wont deadlock the system, and a leader will be chosen eventually.

Messages are handled using the raftpb import, using protocol buffer messages to send appends, using different message types. Each node state as follower, candidate or leader implement their own step methods called stepFollower, stepCandidate and stepLeader, which take steps depending on an incomming messagetype, such as the "msgBeat" type, which is an internal type that tells the leader to send out a heartbeat message as "msgHeartBeat" type to the followers, which will then take their respective steps foward in the algorithm.

2 Go Features

The raft implementation of the algorithm raft.go does not implement channels or go routines, purely using the base golang functionality to implement the algorithm and struct.

The node.go class however, does implement and make use of channels and go routines. The channels are mainly used to recieve and propagate messages, on top of being a sort of stopper in the form of a "ready" channel, which is to recieve the current point-in-time state of the node.
go routines in the node class are used when starting or restarting nodes, calling the "run" func, which sets up most of the important channels, and handles the main functionality of receiving, and handeling protocol buffer messages, to use in the algorithms functionality.

3 Communication Methods

Our choosen raft implementation makes use of remote procedure calls (RPC) to send and receive messages in the protocol buffer format. This is almost the exact same as gRPC, but it is not exactly the same.

Because the raft implementation does not use gRPC, but it's own implementation of the protocol buffer format, it cannot make use of services and other gRPC specific features and has resorted to very basic RPC communication. It uses self describing message types to differentiate between sent messages and responses to those messages, by using a system like this it can very simple tell if a message was received based on the message type of the response.

4 Protocol Faithfulness

The implementation faithfully implements the raft protocol with heartbeats and election timeout as the two timeout to have a leader election and log replication with an append entries method. All of this is explained in greater

detail in our *Implementation* section. The implementation is very bare bones and only has an implementation of nodes used for testing. This is so the user can implement their own nodes with the chosen raft implementation.