CS 505: Exercise 6

Akshay Jajoo, Devesh Kumar Singh

Due: 03/26/2016 Tuesday 11:59 PM

Algorithm Overview

For achieving consensus that can handle crash failures, such that each correct process returns a response in a finite number of rounds, we try to implement our algorithm along the lines of Raft Consensus Algorithm as per the paper (https://pdos.lcs.mit.edu/6.824/papers/raft-extended.pdf).

In Raft, all the servers involved in consensus are in one of the three given states, a leader, a candidate or a follower. The elected leader is responsible for consensus, since it is being decided by other servers. The leader regularly sends a heartbeat message to all the servers, stating that it is alive and well, and each server receiving heartbeat from the leader, also known as a follower, has a timeout, which lies generally between 150 and 300 msec, also configurable and the heartbeat should be received in that timeout interval. When the heartbeat is received, this timeout is reset to a random value between the given range. In case the heartbeat is not received, it means that the leader is not alive, and a follower converts its status to candidate, and start a new leader election procedure.

In raft, we have a concept of a term, which essentially are epoch of variable lengths, starting with election of a leader, and ends when leader fails, or we cannot decide upon a leader, called a split vote. A current term number is maintained by all servers.

Now to elect a new leader, the candidate server first starts by increasing the current term's value, and then it sends a RequestVote message to all other servers, while also voting for itself. The other servers will than vote for the first candidate that has sent them RequestVote message, and reject all others. A server only votes once per term, and the candidate server which receive votes from a majority of servers, it changes itself into a new leader. A candidate server getting a message from a leader which has a term number which is equal to or greater than the current term will change itself to a follower. But if none of these two condition happen, then we get a split vote, and a new leader election is started after a timeout.

Now in our algorithm, each of our processes communicate via a UDP socket to send and receive heartbeats. At-least every one-tenth of the minimum of the range of election timeout, which is by default 15 ms in our implementation, we check if we have not received heartbeat since last election timeout reset. Leader sends heartbeat to all others every half of the minimum of the range of election timeout(this may increase to at-most .55 times minimum of the range of election timeout). Each server keeps track of all other servers which are alive. Each server responds to leader whenever a heartbeat is received. Before sending heartbeat leader checks if it received response for previous heartbeat. If it hasn't received for some node it assumes it to be crashed and propagates this information to all others, and they can then remove this server from their list of alive servers. And when the leader crashes, this information can be sent by any of the servers.

Now as per the famous lower bound for solving consensus in synchronous message passing systems with f failures, we need at least f+1 rounds of communication to ensure consensus is reached. To make this happen, a server run the leader election for f+1 rounds, and only when the same leader gets the majority votes for all the rounds, is the server chosen as a leader, else a new leader election is started for f+1 rounds.

Correctness

For the correctness of our algorithm, we need to make sure that all the three properties of consensus, i.e. agreement, validity and termination holds.

Agreement:

For agreement, a process/node declares itself as leader only if it has vote from majority of processes/nodes and that too for continuous maxCrash+1 rounds. Since every process is allowed to vote for a unique process for a term, there cannot be more than one leader for a term. Also once a process/node becomes leader, it informs all the other processes/nodes that it has become the leader. So we see that all non-faulty nodes, i.e. followers decide on a leader.

Validity:

For validity, we can see that when a leader fails to send heartbeat to the followers, then others assume it to be no longer alive, and then one(or more) of the follower(s) process/nodes changes itself to a candidate and initiate election, and the one which gets the majority votes and becomes the leader. So we see that a node, which was participating in the protocol can only becomes the leader.

Also to become leader the process/node has to initiate election. So only participating process/node can become leader.

Termination:

For termination, we can see that in a system with at most f crashes, the leader election takes f+1 rounds and finally decides on a leader. A process can only vote for a single server throughout all the rounds, and if it has voted for a leader but is crashed in the next round, running f+1 rounds ensures that the leader gets an actual majority from the active nodes, since all the nodes which might crash would have done so in the f rounds itself.

Pseudocode for process.py

```
if election timeout happened and currentState != leader:
        initiateElection()

if currentState == leader and heartbeat has timeout:
       sendHeartBeatToAll()

//To handle election initiation
initiateElection():
      setCurrentTermTo(currentTerm+1)
      setCurrentElectionRound(0)
      setCurrentStateTo(candidate)
      startNewElectionRound()

//sends heart beat to all processes
sendHeartBeatToAll():
      for host in otherhosts:
           sendHeartBeatTo(host)

//starts a new election round
startNewElectionRound():
      setCurrentElectionRoundTo(currentElectionRound+1)
      voteFor(self)
      sendVoteRequestToAll()

//votes for itself, or sends vote request to others
voteFor(this):
      if this != myId:
          sendVoteRequestTo(this)
      else:
            voteRequestAcceptedBy(self)
     refreshElectionTimeout()

//sends vote request to all servers
sendVoteRequestToAll():
      for host in otherhosts:
            sendVoteRequestTo(host)

//decide on becoming the leader
voteRequestAcceptedBy(this):
       if majority votes:
            if currentElectionRound  > maxCrashes:
                  becomeLeader()
            else:
                 startNewElectionRound()
```

```
//become the leader
becomeLeader():
      setCurrentElectionRoundTo(0)
      setCurrentStateTo(leader)
      setCurrentLeaderTo(myId)
      sendHeartBeatToAll()

//take different actions based on the message received
messageAction(sendersId,sendersTerm,sendersValue):
      if sendersValue == heartBeatResponse:
            heartBeatResponseReceivedFrom(sendersId)
            return
      if currentTerm>sendersTerm:
            return
      if currentTerm == sendersTerm:
            if sendersValue == heartbeat:
                  actOnHeartBeatReceivedFrom(sendersId)
            else if sendersValue == voteRequest:
                  actOnVoteRequestFrom(sendersId)
            else if sendersValue == acceptVoteRequest:
                  voteRequestAcceptedBy(sendersId)
      else:
            setCurrentTermTo(sendersTerm)
            setCurrentStateTo(follower)
            if sendersValue == heartBeat:
                        actOnHeartBeatReceivedFrom(sendersId)
            else if sendersValue == voteRequest:
                        actOnVoteRequestFrom(sendersId)

      //action on receiving heartbeat
      actOnHeartBeatReceivedFrom(this):
      refreshElectionTimeout()
      setCurrentStateTo(follower)
      if currentLeader != this:
            setCurrentLeaderTo(this)
      sendHeartBeatResponseTo(this)
```

Working of Leader.py

This takes the required parameters checks for its correctness. Then edits the script file process.py so that they get access to these parameters. Then executes process.py on each host by ssh. This also sets a TCP server which will listen for all these process for logs. This server dies if it doesn't listen for 2000 seconds, i.e. around 33 minutes

Design Decisions and Corner cases

Every server should have knowledge of which all servers are currently active in the system, so that when in future it contests election to become the leader, it knows this information and can choose the majority from the active servers. This knowledge is maintained by the servers themselves, as the current leader informs them about servers who have not responded to a heartbeat, and then these inactive servers are removed from the list of active servers. The concept of a heartbeat response is included to help in this.

The implementation assumes that since we are communicating over UDP, any message send will be delivered.

To handle the case that all the processes should start executing at the same time, we let each process sleep for some fixed time after it gets binded to a UDP socket where it listens for the messages, so that until everyone gets binded, no process starts sending. This may cause unnecessary delay at the start of the processes.