CS 7610: HW-2

Name: Sumeet Mahendra Gajjar

Solution 1.a: Before receiving the new message the local variables of Acceptor A1 are as follow:

- $minProposal = 10$
- $acceptedProposal = 10$
- $acceptedValue = 3$

A1 will reject the prepare($proposal = 8, value = 5$) message since $minProposal$ is 10. It will Return($acceptedProposal = 10, acceptedValue = 3$) since has already accepted the proposal.

Solution 1.b: Before receiving the new message the local variables of Acceptor A1 are as follow:

- $minProposal = 10$
- $acceptedProposal = 10$
- $acceptedValue = 3$

A1 will update it's $minProposal$ to 11 and Return($acceptedProposal = 10, acceptedValue = 3$) since has already accepted the proposal.

Solution 2: We define a PAXOS protocol run to be safe, if all the servers ends up with the same $acceptedProposal$ and same $acceptedValue$.

It is not safe for a proposer to restart and reexecutes the paxos protocol from the beginning with the same proposal number used previously, but with a different initial value of v2.

Proof: Let's assume it is safe for a proposer to restart and reexecutes the paxos protocol from the beginning with the same proposal number used previously, but with a different initial value of v2. Thus every process at the end of the protocol should have the same $acceptedValue$.

Let's consider five processes P1, P2, P3, P4, and P5. Following steps occurs while executing the PAXOS protocol:

- P1 sends prepare($proposal = x$) to P2-P5 and receives true in the response from P2-P5.
- P1 sends accept($proposal = x.value = v1$) to P2.
- P2 accepts P1's proposal, updates the $acceptedProposal = x$ and $acceptedValue = v1$.
- P1 crashes before sending accept messages to P3-P5.
- P1 restarts
- P1 sends prepare($proposal = x$) to P2-P5.

  P1 receives true in the response from P3, P4 and P5 since they have not yet accepted any proposals and still their $minProposal = x$.

  Thus P1 receives true response from majority of the Processes and thus it has the quorum.
- P1 sends accept($proposal = x.value = v2$) to P2-P5.
- P2 rejects the accept since it has already accepted a proposal.
- P3-P5 accepts the proposal, updates the $acceptedProposal = x$ and $acceptedValue = v2$.

Since P2 and P3-P5 have different $acceptedValues$, it contradicts our initial assumption and thus proves reexecuting the paxos protocol from the beginning with the same proposal number used previously, but with a different initialvalue of v2 is unsafe.

Solution 3.a: The following steps follow when server 0 crashes:

For easier understanding any variable var on server i is represented as var_i.

Server 1 has detected the leader failure, since the $Progress\_Timer_1$ expires. The $Progress\_Timer$ on servers 2, 3 and 4 have not yet expired.

- Server 1 initiates the view change for $view = 1$
  - It clears data structures: $VC_1[]$, $Prepare_1$, $Prepare\_oks_1$, $Last\_Enqueued_1[]$
  - It updates $State_1 \leftarrow LEADER\_ELECTION$
  - It updates $Last\_Attempted_1 \leftarrow 1$
  - It sends $View\_Change(server\_id = 1, attempted = 1)$ to all servers.
  - It updates $VC_1[1] \leftarrow View\_Change(server\_id = 1, attempted = 1)$
- Server 1 receives its own $View\_Change$ message and discards it because of $Conflict(View\_Change)$ returns $TRUE$ since it has sent the message to itself
- Servers 2,3 and 4 receives the $View\_Change$ message from server 1, and they discard it because of $Conflict(View\_Change)$ returns $TRUE$ since their $State_{2,3,4} \neq LEADER\_ELECTION$
- $Progress\_Timer_{2,3,4}$ at servers 2, 3 and 4 expires

  Note: i indicates the server id
  - They clear data structures: $VC_i[]$, $Prepare_i$, $Prepare\_oks_i$, $Last\_Enqueued_i[]$
  - They update $State_i \leftarrow LEADER\_ELECTION$
  - They updates $Last\_Attempted_i \leftarrow 1$
  - They send $View\_Change(server\_id = i, attempted = 1)$ to all servers.
  - They update $VC_i[i] \leftarrow View\_Change(server\_id = i, attempted = 1)$
- Server 1 receives the $View\_Change$ message from servers 2, 3 and 4. The $View\_Change(server\_id = i, attempted = 1)$ messages are processed since $Conflict(View\_Change)$ returns $FALSE$
  - $Preinstall\_Ready(attempted = 1)$ returns $TRUE$ since server 1 has received $View\_Change$ message from majority of the servers. Server 1 updates $Progress\_Timer_1 \leftarrow Progress\_Timer_1 * 2$ and sets the $Progress\_Timer_1$
  - According to the "$My\_server\_id \equiv i \mod N$", server 1 is the leader of $Last\_Attempted_1 = 1$, hence it shifts to prepare phase
  - It installs the new view as $Last\_Installed \leftarrow 1$
  - It constructs $Prepare(server\_id = 1, view = 1, local\_aru)$
  - It applies $Prepare$ to data structures, constructs DataList and applies it to $data\_list_1$

- It constructs $Prepare\_OK$ and assigns $Prepare\_oks_1[1] \leftarrow Prepare\_OK$
- It clears $Last\_Enqueued_1[]$ and syncs to disk
- sends $Prepare(server\_id = 1, view = 1, local\_aru = 0)$ to all servers

- Server 1 receives its own $Prepare$ message and discards it because of $Conflict(Prepare)$ returns $TRUE$ since it has sent the message to itself

- Servers 2,3 and 4 receives the $Prepare$ message from server 1, and they process it because of $Conflict(Prepare)$ returns $FALSE$

  Servers 2,3 and 4 do the following since their $State_i = LEADER\_ELECTION$

  - They apply $Prepare(server\_id = 1, view = 1, local\_aru)$ to data structure
  - They construct the $data\_list_i$, using $data\_list_i$ they construct $Prepare\_OK_i$ and set $Prepare\_oks_i[i] \leftarrow prepare\_ok$
  - They shift to reg non leader where they change their $State_i \leftarrow REG\_NON\_LEADER$
  - They install the new view as $Last\_Installed \leftarrow 1$
  - clear the $Update\_Queue_i$ and syncs to disk
  - They send the leader which is server 1, the $Prepare\_OK(server\_id = i, view = 1, data\_list)$ message

- Server 1 receives the $Prepare\_OK(server\_id = i, view = 1, data\_list)$ message from servers 2, 3 and 4

  - Applies the $Prepare\_OK$ message to data structures
  - It shifts to $State\_1 \leftarrow REG\_LEADER$ since $View\_Prepared\_Ready(view = 1)$ returns $TRUE$ as it has received $Prepare_OK$ message from majority of the servers

- Thus server 1 is the new leader and the newly installed view is 1

- While the leader election was in progress, each server was also sending VC_Proof messages. For this particular case, the VC_Proof messages do not alter the path of the algorithm.

Solution 3.b:

The following steps follow when server 0 crashes:

For easier understanding any variable var on server i is represented as var_i.

Server 1 has detected the leader failure, since the $Progress\_Timer_1$ expires. The $Progress\_Timer$ on servers 2, 3 and 4 have not yet expired.

- Server 1 initiates the view change for $view = 1$
  - It clears data structures: $VC_1[]$, $Prepare_1$, $Prepare\_oks_1$, $Last\_Enqueued_1[]$
  - It updates $State_1 \leftarrow LEADER\_ELECTION$
  - It updates $Last\_Attempted_1 \leftarrow 1$
  - It sends $View\_Change(server\_id = 1, attempted = 1)$ to all servers.
  - It updates $VC_1[1] \leftarrow View\_Change(server\_id = 1, attempted = 1)$
- Server 1 receives its own $View\_Change$ message and discards it because of $Conflict(View\_Change$ returns $TRUE$ since it has sent the message to itself
- Servers 2,3 and 4 receives the $View\_Change$ message from server 1, and they discard it because of $Conflict(View\_Change)$ returns $TRUE$ since their $State_{2,3,4} \neq LEADER\_ELECTION$
- $Progress\_Timer_{2,3,4}$ at servers 2, 3 and 4 expires

  Note: i indicates the server id
  - They clear data structures: $VC_i[]$, $Prepare_i$, $Prepare\_oks_i$, $Last\_Enqueued[]_i$
  - They update $State_i \leftarrow LEADER\_ELECTION$
  - They updates $Last\_Attempted_i \leftarrow 1$
  - They send $View\_Change(server\_id = i, attempted = 1)$ to all servers.
  - They update $VC_i[i] \leftarrow View\_Change(server\_id = i, attempted = 1)$
- Server 1 crashes while receiving the $View\_Change$ message from servers 2, 3 and 4
- Let's assume $Progress\_Timer_2$ at servers 2 expires again

  Thus it detects the failure and starts with the leader election by sending the $VIEW\_CHANGE$ message
- Server 2 initiates the view change for $view = 2$
  - It clears data structures: $VC_2[]$, $Prepare_2$, $Prepare\_oks_2$, $Last\_Enqueued_2[]$
  - It updates $State_2 \leftarrow LEADER\_ELECTION$
  - It updates $Last\_Attempted_2 \leftarrow 2$
  - It sends $View\_Change(server\_id = 2, attempted = 2)$ to all servers.

- It updates $VC_2[2] \leftarrow View\_Change(server\_id = 2, attempted = 2)$

- Server 2 receives its own $View\_Change$ message and discards it because of $Conflict(View\_Change)$ returns $TRUE$ since it has sent the message to itself

- Servers 3 and 4 receives the $View\_Change$ message from server 2, and they process it because of $Conflict(View\_Change)$ returns $FALSE$

  Since the $attempted$(i.e. 2) > $Last\_Attempted_{3,4}$(i.e. 1) and $Progress\_Timer_{3,4}$ is not set, they shift to leader election with $attempted = 2$

  Note: i indicates the server id

  - They clear data structures: $VC_i[]$, $Prepare_i$, $Prepare\_oks_i$, $Last\_Enqueued[]_i$
  - They update $State_i \leftarrow LEADER\_ELECTION$
  - They updates $Last\_Attempted_i \leftarrow 2$
  - They send $View\_Change(server\_id = i, attempted = 2)$ to all servers.
  - They update $VC_i[i] \leftarrow View\_Change(server\_id = i, attempted = 2)$

- Server 2 receives the $View\_Change$ message from servers 3 and 4. The $View\_Change(server\_id = i, attempted = 2)$ messages are processed since $Conflict(View\_Change)$ returns $FALSE$

  - $Preinstall\_Ready(attempted = 2)$ returns $TRUE$ since server 2 has received $View\_Change$ message from majority of the servers. Server 2 updates $Progress\_Timer_2 \leftarrow Progress\_Timer_2 * 2$ and sets the $Progress\_Timer_2$
  - According to the "$My\_server\_id \equiv i \mod N$", server 2 is the leader of $Last\_Attempted_2 = 2$, hence it shifts to prepare phase
  - It installs the new view as $Last\_Installed \leftarrow 2$
  - It constructs $Prepare(server\_id = 2, view = 2, local\_aru)$
  - It applies $Prepare$ to data structures, constructs DataList and applies it to $data\_list_2$
  - It constructs $Prepare\_OK$ and assigns $Prepare\_oks_2[2] \leftarrow Prepare\_OK$
  - It clears $Last\_Enqueued_2[]$ and syncs to disk
  - sends $Prepare(server\_id = 2, view = 2, local\_aru = 0)$ to all servers

- Server 2 receives its own $Prepare$ message and discards it because of $Conflict(Prepare)$ returns $TRUE$ since it has sent the message to itself

- Servers 3 and 4 receives the $Prepare$ message from server 2, and they process it because of $Conflict(Prepare)$ returns $FALSE$

  Servers 3 and 4 do the following since their $State_i = LEADER\_ELECTION$

  - They apply $Prepare(server\_id = 2, view = 2, local\_aru)$ to data structure

- They construct the $data\_list_i$, using $data\_list_i$ they construct $Prepare\_OK_i$ and set $Prepare\_oks_i[i] \leftarrow prepare\_ok$
- They shift to reg non leader where they change their $State_i \leftarrow REG\_NON\_LEADER$
- They install the new view as $Last\_Installed \leftarrow 1$
- clear the $Update\_Queue_i$ and syncs to disk
- They send the leader which is server 2, the $Prepare\_OK(server\_id = i, view = 2, data\_list)$ message

- Server 2 receives the $Prepare\_OK(server\_id = i, view = 2, data\_list)$ message from servers 3 and 4
  - Applies the $Prepare\_OK$ message to data structures
  - It shifts to $State\_2 \leftarrow REG\_LEADER$ since $View\_Prepared\_Ready(view = 2)$ returns $TRUE$ as it has received $Prepare\_OK$ message from majority of the servers
- Thus server 2 is the new leader and the newly installed view is 2
- While the leader election was in progress, each server was also sending VC_Proof messages. For this particular case, the VC_Proof messages do not alter the path of the algorithm.

Problem 4.a: Prove that the leader election is safe
Solution 4.a: We define leader election as safe if there is at most one leader elected per term.

Proof: In order to win the leader election, a candidate should receive votes from the majority of servers in the full cluster for the same term. For a cluster of N servers, a majority is defined as $\left\lfloor \frac{N}{2} \right\rfloor + 1$. Thus two majority set of servers from a same cluster will always have one server common.

Also each server in the cluster votes for at most one candidate in a given term, on a first-come-first-served basis.

Let's assume the leader election will result into more than one leader.

Case with Odd number of active servers:

Since there exists more than one leader, in order to win the election all the leaders should have received votes from the majority servers in the full cluster. Since each server can only vote once both majority sets should contain different servers. This cannot happen since two majority set will always have one common server. So one server voted twice, which contradicts with the voting rules. Thus our assumption that more than one leader exists is contradictory and thus the leader election is safe.

Case with even number of active servers:

In case the number of servers is even then two candidates for leader election will receive equal number of votes and it will be a split vote. In this case no leader will be elected as none of the candidates have votes $\geq$ majority. This will result into a re-election until one of the candidate has majority of the votes.

Case with network partition:

During the network partition, if the number of active servers are equal on both sides of partition then the above argument for "Case with even number of active servers" applies with one candidate present on one side and the other candidate present on the other side resulting into no leader.

If the number of active servers are not equal on both sides of partition and one side contains majority of the servers then the above argument for "Case with odd number of active servers" applies resulting into at most one leader.

Problem 4.b: Discuss liveness (e.g. is it live, yes/no, why/why not).
Solution 4.b:

In order to win the leader election, a candidate should receive votes from the majority of servers in the full cluster for the same term. For a cluster of N servers, a majority is defined as $\left\lfloor \frac{N}{2} \right\rfloor + 1$. Thus two majority set of servers from a same cluster will always have one server common.

Also each server in the cluster votes for at most one candidate in a given term, on a first-come-first-served basis.

Theoretical Viewpoint: The candidate issues $RequestVoteRPC$ to all the peers in the full cluster to receive votes. we assume it's an asynchronous model, there is no bound on the amount of time a server will take to reply. Thus it is not possible to say whether a server has crashed or is simply taking too long to respond. Having said that the FLP result shows that given such an asynchronous setting, where only one server might crash, there is no distributed algorithm that solves the consensus problem.

Assuming that none of the servers are faulty and there is just one candidate in the election, then the RAFT leader election algorithm will terminate and progress will always be made.

However, when there are more than one candidates in the election, then there is a chance that two candidates will receive equal number of votes and it results in to a split vote and thus a re-election. Given RAFT uses randomness in the election timeouts to reduce the split vote scenario, there is a rare possibility of two servers having the same randomized timeouts. Thus every election will result into a split vote followed by a re-election showing no progress or liveness.

Practical Viewpoint: The theoretical argument when the is just one candidate for the election holds true in the practical case as well showing progress or liveness.

Coming to the case when of split vote, there is always going to be some sort of randomization introduced while sending the messages to peers because of the network, maybe because of the scheduling mechanism of the kernel, so on and so forth. Thus split vote cases would be rare in practise.

The authors of RAFT empirically show in Figure 16 of the paper how that adding a randomness to timeouts reduces the leader election time significantly during split vote scenario. Without randomness the split vote leader election scenario was taking longer than 10s to complete, and adding a randomness of mere 5ms to the timeouts reduce that time to a median of 287ms which is huge improvement.

Configuring timeouts is an important thing when it comes to leader election in RAFT and the authors claim that RAFT will be able to elect and maintain a steady leader given the following equality satisfies:

$$broadcastTime << electionTimeout << MTBF$$

Here the $broadcastTime$ is the time taken to send RPC and receive response, $electionTimeout$ is the time a server will wait for communication from leader after which it will start a election and $MBFT$ is the average time between failures for a single server.

The above equality holds true for almost all the practical cases. The $broadcastTime$ is in range of $0.5 - 20ms$ (depending on the underlying H/W), this results into a $electionTimeout$ of range $10 - 500ms$ and generally a server $MBFT$ is in months.

Thus from a practical viewpoint the RAFT leader election algorithm should demonstrate progress and liveness with a very high likelihood.

Solution 5: Node-3 cannot become the leader when Node-1 crashes.

In order to win the leader election, a candidate should receive votes from the majority of servers in the full cluster for the same term. For a cluster of N servers, a majority is defined as $\left\lfloor \frac{N}{2} \right\rfloor + 1$. Also the candidate's log should be at least up-to-date with majority of the servers in the full cluster.

During the leader election, Node-3 sends a $RequestVoteRPC$ to Node-2. Node-2 returns false since the Node-3's log is not up-to-date as Node-2's log. Thus Node-3 did not receive majority votes and hence it cannot become the leader.

Problem 6.a: What are the conditions that enforce safety in RAFT?
Solution 6.a: For a consensus algorithm, we define safety as all servers eventually having the same committed value. Following are the conditions which enforce safety in RAFT:

- Election Safety/Election Restriction: This condition ensures at most one leader is elected during the leader election.

  During the election, if a candidate's log is not at least as up-to-date as that of majority of the servers in the full cluster, then that candidate cannot become the leader for that term.

- Leader Append-Only: A leader will never delete or overwrite entries in its log, it will only append new entries.

- Log Matching: This condition ensures two things:

  Two entries in different logs with same index and term, store the same command

  If any two logs contains a entry with same index and term, then all entries in the logs are identical up to that index.

- Leader Completeness/Committing entries from previous terms: This condition ensures that If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all the higher-numbered terms.

- State Machine Safety: This condition ensures if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

Problem 6.b: Explain why these conditions will guarantee safety
Solution 6.b:

- Election Safety/Election Restriction: we have already proven in Question 4.a that leader election is safe in RAFT. Thus at most one leader can exists for a given term in RAFT. Leader is one who is responsible for replicating and committing an entry with followers and since there can exists only one leader, there cannot arise a situation where two logs contains a different entry at the same index and same term.

  Let's assume a candidate whose log is not up-to-date as that of majority servers in the full cluster, is elected as leader. The lagging leader can overwrite the follower logs which will result into information loss since the followers were already ahead of the leader. Thus not allowing such a lagging candidate to become the leader enforces safety guarantee.

  Preventing a lagging leader also ensures the flow of logs is unidirectional(i.e. from Leader to Followers) which reduces the log replication complexity from other algorithms like Viewstamped Replication.

- Leader Append-Only: This condition helps leader to not take any special actions to restore log consistency when it comes to power. The leader can proceed with the normal operation and logs automatically converges.

- Log Matching: Since two entries on different logs with same index and term represent the same command, while executing the command at that entry on all servers will result into same state on all the servers.

  And since if two entries in different logs have the same index and term, then the logs are identical in all preceding entries, executing the sequence of commands till that entry will result into same final state on all the servers.

  Both sub-conditions ensures the cluster has consensus on a result in event of a server failure.

- Leader Completeness/Committing entries from previous terms: Leader completeness property aids the State Machine Safety ensuring same set of logs entries exists on all servers.

  Also leader Completeness Property ensures that only servers with the joint consensus ($C_{old,new}$) can be elected as leader, ensuring it is safe the leader to create a log entry describing C_new and replicate it to the cluster

- State Machine Safety: Since all servers will apply exactly the same set of log entries to their state machines, in the same order, the final result will be same preventing any inconsistencies ensuring safety.

  Thus the above properties as a whole enforces and guarantees safety in RAFT.