We divided the implementation broadly into three files:

1. **Prop_node.py**
2. **Network_ctrl.py**
3. **Client.py**

Network.ctrl.py

It contains the network implementation i.e it has the methods defined to send messages across different
servers based on their IP address. The important methods are listed below:

- **wait_for_ctrl_connections** – This method is run in a background thread in each server so that servers can read messages from other servers.
- **send_ctrl_message_with_ACK** – This method has 5 parameters :
    - Message - The data you want to pass.
    - messageType – What type of message it is.
    - extra – Any extra data you want to send.
    - requestNode – The node to which you want to send this message.
    - timeout – the timeout value

Prop_node.py

This is the main file where the BFT Raft algorithm is implemented
Main methods:

- **join_network** – Used by servers to join a cluster
- **start_leader_election** – Used by a server to start leader election
- **stabilization_routine** – Used by servers to know about other servers in the cluster.
- **heartbeat_routine** - As soon as a server becomes a leader it starts sending out heartbeat signals to existing nodes in the system to show that it is alive.
- **leader_timeout_routine** - When a node in the system does not receive heartbeat signal from the leader within its timeout period, it announces itself as a CANDIDATE and starts election by calling start_leader_election

Client.py

Client has **two major tasks.**

- Send commands to be executed to the server.
- If the command takes too long to get executed it intervenes to start a new leader election.

1. When client does not get a reply from the PBFT servers in a stipulated time frame, it asks for a new leader to be elected. We are working on certain testcases which are still troublesome.

Let us take a look at different messageTypes supported to enable smooth functioning of PBFTRaft.

Background Daemons

1. `leader_timeout_routine`
   This thread keeps track of the current leader and if it times out it asks for new leader to be elected.
2. `heartbeat_routine`
   This thread sends out continuous heartbeat signals to all replicas in the system. Only the leader node does that.
3. `stabilization_routine`

   This takes care of stabilization of the network system. i.e whenever a new node joins or a node dies, this routine takes care of updating the current status of the distributed network.

Discussing testcases

- Leader Election test cases

  Whichever server times out first, announces itself as a CANDIDATE using the method `start_leader_election()` and asks for votes from others. If it gets a majority, it sends the quorum to others and announces itself as the leader and starts sending out heartbeat signals. Others verify this quorum and accept it as the LEADER only if the quorum is valid.

  1. When only <=3 servers in the system no election is possible.

2. When 4th server joins, one of the servers times out and starts leader election. Leader is elected and heartbeat mechanism is working properly. Tested this by inserting an adversary server who sleeps for time greater than leader timeout, in which case the FOLOWER times out and announces itself as a CANDIDATE and starts election. When someone joins after leader is elected, that node is able to get heartbeats.
3. When a node joins after election, it has to figure out the current leader. With heartbeat, we also send the server information. So, the newly joined node knows who the leader is. As of now we assume that heartbeat sending node is the leader. Also, when a CANDIDATE announces itself as a LEADER, we send a quorum which the other nodes in the system verify. This quorum right now is just a list of IP addresses the CANDIDATE got its votes from.
4. I have created 5 servers. Until there are minimum 4 servers no election is possible. If leader server dies and there are still 4 nodes in the system a new leader is elected. When we kill 2 servers, only 3 remain. So, all the nodes become FOLLOWERS again as no election is possible. When a node joins again, election is triggered and new leader is elected.

- Log Replication test cases

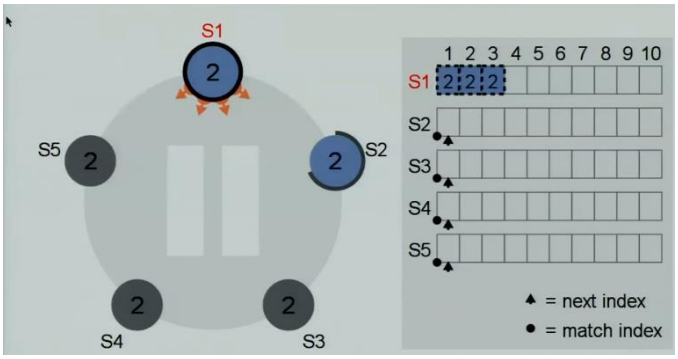
Client sends a request to one of the known replicas. Replica receives it. If the replica is the leader itself, it sends the log to be replicated to the rest of the replicas in the system, else the replica just forwards the request to the current leader of the system. The messageType we have used for this is `ACCEPT_REQUEST_FROM_CLIENTS.` If any of the replicas is lagging behind, the leader first catches them up with the current log. Once all replicas are updated the leader actually sends the new log to be replicated. Here is where PBFTRaft is a bit different from Raft. Whenever a replica records the log entry it sends `APPEND_ENTRY_RESPONSE_FOR_LOG_REPLICATION`
for that log entry to all the replicas in the system so that everyone can take their own decision about when to commit a log entry (i.e when a particular log entry gets replicated in more than the half of the network). At this point of time the response if returned to the client. I have attached a screen recording of the same.

In case a server is down for a long period and others do replication for quite some time. Now suppose this server comes up and gets elected as LEADER , it will replace all committed entries. To avoid this we make sure that a replica never votes for a server whose log is less upto date than its own.

Log Replication – Example 1

- Here servers S3, S4 and S5 are inactive.

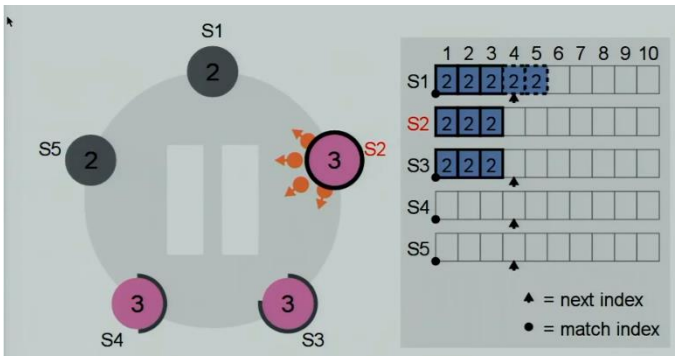
- S1 is elected as leader in term 2.
- Client sends three requests to S1.
- S1 replicates those logs to the only active server S2 as shown in the figure.
- But what is the downside here ?
- If S1 and S2 die and S3, S4 and S5 come up, they can elect a leader among themselves and the entries made by S1 will be lost forever.
- That is why we need the logs to be replicated to at least half of the members of the cluster.
- So, an entry is committed when the leader knows that entry has been sent to majority of the cluster.
- Dotted lines show that the entries are not committed
- Solid lines show that the entries are committed.



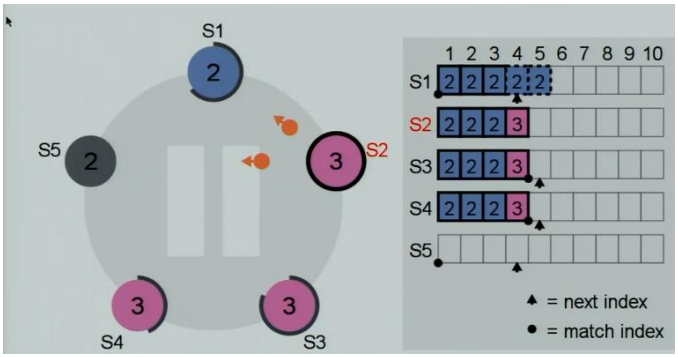
Let us have a look at a different example.

Log Replication – Example 2

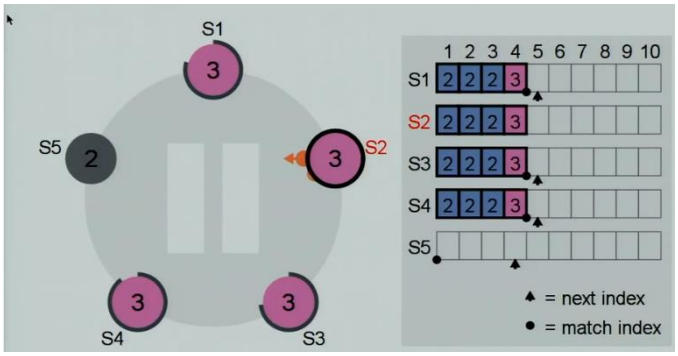
- S1 was leader. Got 2 requests from client. But before it could replicate it to followers S1 died.
- S4 comes back up and S2 is the new leader for term 3
- But S4 log entries are empty.
- S2 replicates entries to S4 before proceeding.



- Now S4 is all updated, even with new client request with term=3.
- Here the pink entries are committed (clients have been told that their requests have been recorded forever)
- Now S1 wakes up and S2 is still the leader.
- So, what should happen in this case?



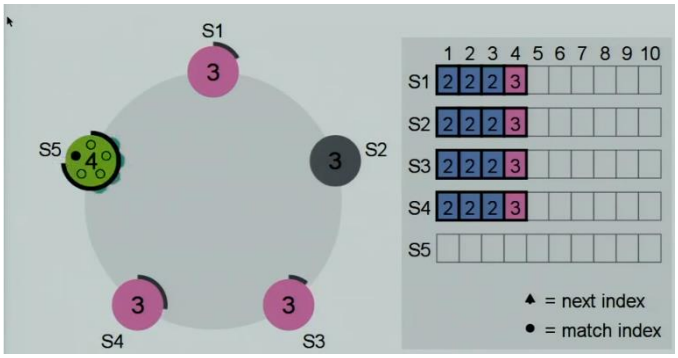
- The old entries will be overwritten as they were not committed (not applied to the state machine)



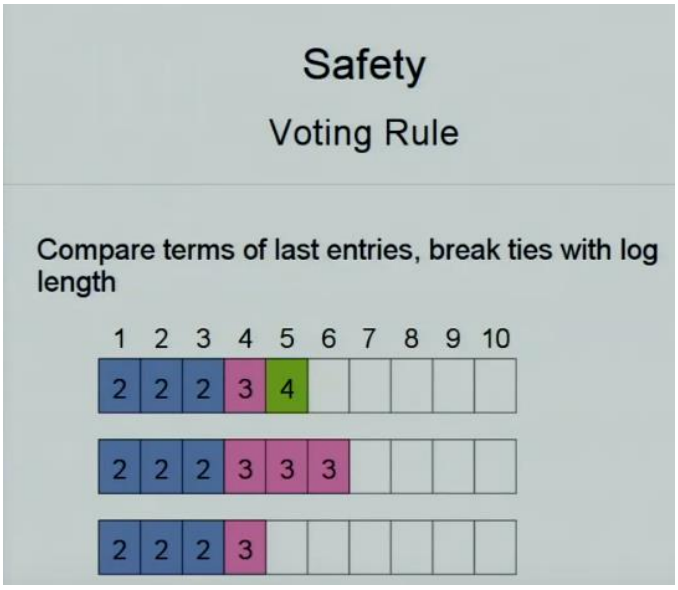
Now let us have a look at an example of Safety.

Safety Example

- Suppose S5 times out first and is appointed as the leader. It creates a problem.
- It has completely empty log record. So, it does not know that there are committed entries out there and it will replace it with entries of its own.
- How can this problem be countered?



- A server will not vote for a server whose log entry is less up to date than its own.
- So here S5 will not get any votes except for its own.
- Voting rules are shown in the figure.



- Client Intervention test cases

  Whenever a leader is stalling the processing of a message beyond the client's timeout, the client sends a message to elect a new leader. Upon the leader's election, the client can resend the message to the leader. To test, we reduced the client's timeout to 15 seconds so that it timed out fairly quickly starting a new election. As expected, the client was able to resend the message to the new leader.

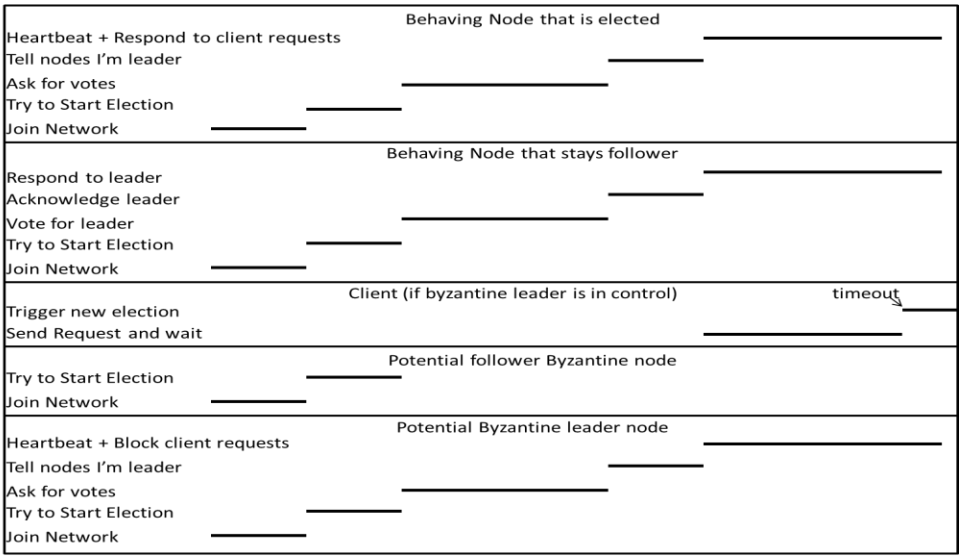
- Election Verification test cases

  When an election is happening, the leader can spoof the election by sending out a quorum of voters who didn't actually vote for them. To rectify this, all followers receive the quorum that the leader claims and verifies that those voters actually voted for the leader by sending a message to them. Only if enough voters respond that they did indeed vote for the leader is the election verified. To test this, we made an identical node that upon starting an election immediately claimed to have won and sent out a fake quorum of voters. As expected, this quorum was rejected and another node was voted into leadership.

- Lazy Voter test cases

  A byzantine node could break the system by repeatedly triggering elections, thus stalling any progress. To fix this, nodes only vote in elections if they perceive that the leader is faulty. This happens only if the leader times out or a client intervention is received. To test this, a byzantine node was created that repeatedly tries to trigger elections. As expected, after the initial election this node was ignored by all the other voters which knew the leader wasn't faulty.

- Timing Diagram



Challenges Faced

Going from Raft to a Byzantine-fault-tolerant Raft presented many challenges, including the following:

1. Significant slowdown – Because the network was not structured to allow broadcasting of messages, the speed of the message processing was directly correlated to the number of messages sent. Because BFT Raft required a much larger number of messages, it took a fairly long time to run. This could be mitigated by adding in broadcast messages and responses, but this would require some additional work and changing of the control flow.
2. Lazy voters – The way the control flow was originally structured, making the voters lazy would end up with all of them repeatedly declaring candidacy at once. To fix this, we added a timeout that only allowed them to declare candidacy every 3 seconds at most.

Code Location

https://github.com/venkatvandy/Distributed/tree/master/PBFTRaft