How to Run

Run the below from SBT to run different RAFT Participants

runMain com.raft.RAFT\_Implementation\_App **<port number>**  
e.g.   
 **Seed Nodes** runMain com.raft.RAFT\_Implementation\_App 25251  
 runMain com.raft.RAFT\_Implementation\_App 25252  
  
 **Random RAFT Nodes**   
 runMain com.raft.RAFT\_Implementation\_App 25253  
 runMain com.raft.RAFT\_Implementation\_App 25254  
 runMain com.raft.RAFT\_Implementation\_App 0

Run the below from SBT to run different clients

runMain com.raft.client.RAFT\_Client

Attached is the final output of the RAFT (state machine info for each node) along with the leader election log.

**I have also attached an invalid\_data file to help validate the use case where leader deletes any extra info in the follower’s log.  
You can create a new file ‘RAFT\_SEED\_25256.json’ and paste the content of the invalid data.   
Assuming you already have RAFT nodes running at port 25251, 25252, 25253 with the attached json files. Start a new RAFT node at port 25256, once the node joins the cluster, the leader will sync up the log for 25256.**



Technical Design

Used akka cluster to set up the RAFT framework where seed node are as below (**RAFT\_CLUSTER.conf**)

"akka:**//RaftSystem@127.0.0.1:25251",**"akka:**//RaftSystem@127.0.0.1:25252"]**

The cluster needs atleast 3 RAFT nodes for optimal performance (with 2 nodes the voting might result into split voting an which in turn can cause no leader election)

Every node that joins the RAFT cluster/framework, reads its state from its state machine, in this from a json file RAFT\_SEED\_<PORT\_Number>.json e.g RAFT\_SEED\_25256.json.

The json file contains entries in below format where each row is persistent state/data with current term, log index and client data information.

|  |
| --- |
| {"term":2,"index":1,"data":{"data":"hi"}}  {"term":2,"index":2,"data":{"data":"Nice"}}  {"term":2,"index":3,"data":{"data":"Perfect"}} |

The last row in the json file (if exists) indicates the default term the nodes starts with.

Below messages are used for Election/log replication among the nodes

|  |  |
| --- | --- |
| INIT\_TIMER | This instantiates a timer actor which runs a timer for either of the below If the node is the leader then timeout duration is for heartbeat controlled by the value of raft.timer.heartBeatTimeout  If the node is a candidate/follower then the timeout is for election and below property entries are used  raft.timer.minValue=**6000** raft.timer.maxValue=**12000**  **Sample Code** timeOut = **if**(parentState == LEADER) {( heartBeatTimeout + generator.nextInt(heartBeatTimeout))} **else** { (minValue + generator.nextInt((maxValue - minValue) + 1))} |
| *ELECTION\_TIMEOUT* | Upon the successful timeout,  if the node is the leader,  broadcasts Heartbeat message [***APPEND\_ENTRIES*** *with empty dataset*]  else   * steps up as a candidate * adds 1 to its current term * Broadcasts a ‘**RequestVote**’ message indicating election |
| *RequestVote* | As state above, it is used to held election |
| RESULT\_APPEND\_ENTRIES | It serves two purposes   * Election Vote:   When a node receives ‘RequestVote’  it compares the candidates info such as term, logindex with its local set of data and votes in favor/against it accordingly via RESULT\_APPEND\_ENTRIES.  If majority of the nodes vote in favor then the candidate steps up as LEADER otherwise steps down to FOLLOWER and up   * Heartbeat Response   When a node receives a heartbeat from the leader, it compares its term with leaders’ term and replies in favor/against it accordingly via RESULT\_APPEND\_ENTRIES |
| APPEND\_ENTRIES | It serves two purposes   * Heartbeat   This is used to let the nodes be aware of the leaders’ presence   * Get consensus on new data   When the leader receives data/command from the client, it broadcasts the ‘APPEND\_ENTRIES’ message to get the consensus on the received data  The nodes sends their response via RESTULT\_APPEND\_ENTRIES.  If the majority of the nodes vote in favor ,the data is committed in its state machine (json file) else the data is rejected and client is informed accordingly.  The APPEND\_ENTRIES is broadcasted regularly based on heartbeat timeout which is less than the election timeout |
| APPEND\_ENTRIES\_LOG\_Inconsistency | When a node receives APPEND\_ENTRIES and if its log is not as updated as the leaders then the node sends out the ‘APPEND\_ENTRIES\_LOG\_Inconsistency’ with its conflict index and term and the leader works with the node to update the node’s log as per the leader via one to one ‘APPEND\_ENTRIES’ message communication |

Messages used to communicate with the state machine

|  |  |
| --- | --- |
| LOAD\_FROM\_FILE | When a RAFT node starts up, it uses this message to get the latest info from the state machine (json file) and set its state accordingly |
| Get\_Entries | This is similar to LOAD\_FROM\_FILE but instead of reading from the physical file, the state machine reads the data from its cache which is refreshed after every update of the state machine (json file) |
| PersistState | This is used to persist the data the leader decided to commit |
| COMMIT\_STATUS | This is to let the leader inform of the status of the last commit request. |

Discussion

**Synchronization**

Since the client commands can only be processed by the leader and once the leader commits the client command (after consensus from majority) the leader asks the majority of the nodes to replicate the same log entry/command in the same order i.e. all nodes see the updates in the same order

Liveness -  Raft uses election timeout between T-2T which guarantees that eventually one node will be elected as leader

**Replication**

The client command is persisted in the leader’s log once majority of followers agree to commit the command and once the leader receives a consensus it writes the command to its log state and asks the majority of nodes who voted for the commit to commit the same entry as well.

Thus the log is always replicated on majority of the nodes

**Fault tolerance**

Since the logs are always replicated in the majority of the nodes so in case the current leader crashes, one of the remaining majority nodes will be elected as leader and would own log replication.

As long as the majority of the nodes are up and running the system would be able to tolerate upto # of majority - 1  nodes"