|  |  |  |
| --- | --- | --- |
| Name | UTA id | Part completed |
| Nikhil Mhatre | 1002122555 | RAFT Nodes in python |
| Imtiaz Bin Rahim |  | RAFT nodes in GO |

**Raft Node Logic (node.py)**

**RequestVote**

Handles incoming vote requests from candidates during elections. Determines whether to grant the vote based on term comparisons and voting history. Updates the node's state and resets the election timer upon granting a vote.

**AppendEntries**

Processes append entries or heartbeat messages from the leader. Validates the leader's term and log consistency before cloning the leader's log or acknowledging a heartbeat. Updates the commitIndex and applies committed log entries to the state machine.

**ClientRequest**

Manages client requests to modify the system's state. If the node is the leader, it appends the operation to its log and initiates replication to followers. If not the leader, it forwards the request to the current leader or responds indicating no leader is available.

**reset\_election\_timer**

Resets the election timer to prevent unnecessary elections. Cancels any existing election timer and starts a new one with a randomized timeout between 1 to 2 seconds. Triggers the start\_election method if the timer expires without receiving a heartbeat.

**start\_election**

Initiates a new election by transitioning the node to a candidate state, incrementing the term, and voting for itself. Sends RequestVote RPCs to all peers concurrently and counts the votes received. Becomes the leader if it secures a majority of votes; otherwise, resets the election timer to retry.

**send\_request\_vote**

Sends a RequestVote RPC to a specified peer during an election. Establishes a gRPC connection to the peer, sends the vote request, and appends the response to a shared list. Logs any errors encountered during the RPC.

**start\_heartbeat**

Initializes the heartbeat mechanism by starting a timer that periodically sends heartbeats to all followers. Cancel any existing heartbeat timer to avoid duplicate heartbeats. Ensures continuous leadership by regularly communicating with peers.

**send\_heartbeats**

Sends AppendEntries RPCs (heartbeats) to all peers to maintain leadership and replicate logs. Launches separate threads for each peer to handle concurrent heartbeat transmissions. Reschedules the next heartbeat upon completion.

**send\_append\_entries(self, peer)**

Sends an AppendEntries RPC to a specific peer to replicate log entries or send heartbeats. Determines the appropriate log entries to send based on nextIndex, constructs the request, and processes the peer's response. Updates nextIndex and matchIndex or retries upon failure.

**append\_log**

Appends a new client operation to the leader's log with the current term and the next index. Logs the append action for debugging and tracking purposes. Prepares the log entry for replication to followers.

**replicate\_log**

Initiates the replication of the leader's log to all peers by sending AppendEntries RPCs concurrently. Launches separate threads for each peer to handle the replication process. Ensures that all followers receive the latest log entries.

**apply\_logs(self)**

Applies committed log entries to the node's state machine. Iterates through the log from lastApplied up to commitIndex, executing each operation. Logs the execution of each operation for transparency and debugging.

**get\_leader\_stub**

Retrieves a gRPC stub for the current leader to facilitate forwarding client requests. Searches through the list of peers to find the leader based on leader\_id. Establishes a gRPC connection to the leader if found; otherwise, returns None.

**GetState(self, request, context)**

Provides the current state of the node for monitoring or testing purposes. Returns a StateResponse containing the node's ID, current term, role, commit index, last applied index, and the log entries. Ensures thread-safe access to state information using a lock.

**2. Client Logic**

The **Client** interacts with the Raft cluster by sending operation requests (e.g., setting values) to the nodes. The client can connect to any node using gRPC:

* If the selected node is the **leader**, it processes the request directly.
* If it’s not the leader, the request is **forwarded** to the current leader, ensuring fault tolerance.

This client mechanism abstracts the details of leader identification, allowing seamless interaction with the cluster.

**3. Test Cases (test.py)**

The **test suite** ensures that the Raft consensus implementation is functioning correctly. Key tests include:

* **Leader Election Test**: Verifies that a leader is correctly elected in the cluster.
* **Basic Log Replication Test**: Ensures that a single client request is replicated to all followers.
* **Leader Failure and Recovery**: Tests that a new leader is elected when the original leader fails and checks if the cluster continues operating correctly after the original leader restarts.
* **Concurrent Client Requests**: Tests multiple concurrent requests to ensure they are correctly replicated across nodes.
* **Client Request Forwarding**: Verifies that a client request sent to a non-leader node is forwarded to the leader and executed correctly.

The tests make use of **gRPC stubs** to communicate with the nodes and validate the **state consistency** and **log replication** across all nodes.

References  
  
<https://arorashu.github.io/posts/raft.html>

<https://medium.com/@jitenderkmr/understanding-raft-algorithm-consensus-and-leader-election-explained-faadf28fd047>

<https://towardsdatascience.com/raft-algorithm-explained-a7c856529f40>

<https://www.educative.io/answers/how-does-the-leader-election-in-raft-work>