Anish Nethi - an34947
Shruti Sriram - ss232499

CS 380D - DISTRIBUTED COMPUTING
RAFT - PROJECT REPORT

PROJECT STRUCTURE:

config.py

- Manages the system configurations
- Contains NodeState enumeration which defines the states that a node can be in - FOLLOWER, CANDIDATE, and LEADER
- Sets the base port and host for server communication
- Defines the timing constraints for leader elections and heartbeat
- Manages the membership of nodes in the cluster; stores the list and the number of active nodes in the cluster in JSON files
    - save_servers():
        - save the IDs, ports, and addresses of the servers in the system
    - save_cluster_members():
        - saves a list of active servers in the system

raft_server.py

- Initializes a RaftNode instance with the given node_id as **raftserver{servernum}**
- Starts the RAFT node, which:
    - initializes the gRPC server
    - calls config to add the server to the list
- Keeps the process running until interrupted
- Upon receiving a shutdown signal, cleans up resources, stops the gRPC server, and calls config to update the cluster configuration

state_manager.py

- StateManager class initializes the node's state by either loading it from a file or creating a new state structure
- State structure:
    - current_term: server's term
    - voted_for: candidate for which the server voted
    - log:

        - term in which entry was created
        - log index
        - command (PUT, GET, REPLACE)
        - key, and
        - value to be stored in the key-value store
    - commit_index: highest log index that was committed
    - last_applied: highest log index applied to the state machine
    - **data: key-value store**
    - sent_length: tracks the last log index sent to each follower
    - acked_length: tracks the highest log index acknowledged by each follower
    - request_history: tracks processed client requests to avoid duplicates
- StateManager class also contains getters and setters for the state variables
- is_duplicate_request():
    - serves as an idempotent function to check for duplicate requests
- record_request():
    - stores the result of processed request
- append_log_entry():
    - creates an entry to be appended
    - checks for conflicting entries in server log and deletes them
- commit_logs_up_to():
    - commits the key-value pair into the data store from last applied to index
    - commits the client and request id pairs for duplicity check

raft.proto
- Specifies RPC methods for inter-server and client communication
- Defines messages that are passed as requests and responses during communication
- The methods were defined based on the specifications mentioned

node.py
- ConnectionManager class manages gRPC connections for internal server communication (7000 ports)
    - _calculate_source_port():

        - maintains unique source ports for each server
    - get_channel():
        - reuses and creates channels from source ports (7000s) to target ports (9000s)
    - close_all():
        - closes all connections
- KeyValueStoreService class acts as the intermediary between frontend and raft node
    - GET/PUT/REPLACE:
        - converts requests to suit the input format of the SendCommand function
    - _convert_response():
        - converts responses from SendCommand to suit frontend response format
    - GetState():
        - returns the state (leader) and term of the server
- RaftNode class holds the attributes of a server like node_id, status, logs, terms, and data
- Manages state transitions and leader elections
    - _run_election_timer():
        - sends heartbeats
        - triggers leader elections if heartbeats have not been received
    - _start_election():
        - when timeout occurs, changes the state of the server to candidate and requests votes from the other servers by sending RequestVote() RPC
- Manages log replication
    - Sends AppendEntries RPC to replicate log so that each server has an up-to-data log
    - _send_heartbeats():
        - sends empty AppendEntries() to inform that the leader is active and also to inform the commit index
- Handles client requests and ensures consistency
- Multiple other functions to ensure the running and proper functioning of the raft nodes

frontend.py

- FrontEndService as an interface for communication with clients
- Manages RAFT lifecycle - starting and stopping servers, and cleaning up
    - StartRaft():
        - starts a cluster with a specific number of servers
        - assigns sequential ports starting from the base port
    - _cleanup_servers():
        - terminates raft servers processes and ensures there is no garbage
- _find_leader():
    - determines the current leader by querying other servers by using gRPC
    - implements gRPC methods which enable the client to communicate (GET, PUT, REPLACE) with the servers
    - defines a forwarding mechanism to enable the requests to be passed to the leader in case the clients contact the follower nodes

IMPLEMENTATION DETAILS:

- **Port Assignments**

  Each node calculates its source port using the formula:
  Source Port = 7000 + (Server ID - 1) * Cluster Size + Target Server Offset

  Target server offset is adjusted to skip the node's own ID

- **Process Naming**

  Each RAFT server process is named as raftserver<ID> for easy identification:

  ```
  setproctitle.setproctitle(f"raftserver{server_num}")
  ```

- **State Persistence**

  State is saved to JSON files for durability:

  ```
  with open(self.state_file, 'w') as f:
      json.dump(self.state, f)
  ```

- **Log Replication**

  Entries are appended and replicated to followers along with the heartbeat signal

- **Leader Election**
  Nodes initiate elections on timeout (when they don't receive a heartbeat)
  ```
  if time_since_last_heartbeat > self.election_timeout:
          self._start_election()
  ```

  Votes are requested and tallied based on acknowledgments from a quorum:
  ```
  response = stub.RequestVote(request)
  if response.vote_granted:
          self.votes_received += 1
  ```

- **Client Request Handling**
  Requests are forwarded to the leader if the clients contact follower nodes:
  ```
  stub = self._get_stub_for_server(self.leader_id)
  response = stub.Put(request)
  ```

DEPENDENCIES:
Mentioned in requirements.txt

NOTE:
- We might have to run frontend.py a few times to get the right output due to the randomness in timeouts for different nodes.
- **Since Python's gRPC does not bind a server to a port the way C, C++, Go, etc., do, blocking a port using an IP table does not ensure that the inter-node communication is blocked, since the server might not be using the port. So, we attempted to simulate blocking of nodes by killing them, which is in mytest.go file.**
- Additonally, we have added a timeout of 1 second after every PUT in the loadDataset() in mytest.go file.