Winter 2024 CSE530 Distributed Systems
Assignment 2
RAFT Application

VERY IMPORTANT:

Use of LLM-based tools such as ChatGPT, etc., is ALLOWED. But if you are not able to answer questions in the demo/viva or if you are not able to explain your code, then you will be penalized. You cannot justify - "ChatGPT produced the code, I don't know the exact purpose of this line of code!"

1. Similar to assignment 1, we will be asking you to share your relevant conversations with ChatGPT and other LLM-based tools. This will not be used for judging you in any way. But it will be used for research purposes so that we can build better LLM-based tools for students (you!!). **Do note that we will have 10% weightage in this assignment's evaluation for sharing these details.**
2. **Please fill this google form for sharing all your interactions with LLM-based tools.**
    a. **Please go through the above google form before you start using LLMs so that you are aware of the kind of information you need to share with us.**

Resources

1. Important:
    a. Raft (Main Algorithm)
        i. Original Paper
    b. Medium Explanations Part 1, Part 2
    c. Raft Visualization
    d. Low Latency Reads in Geo-Distributed SQL with Raft Leader Leases | Yugabyte (To understand the problem with leader reads and how leader leases work to solve this issue)
2. Additional:
    a. Replication Layer (cockroachlabs.com)

Introduction

This assignment focuses on implementing a modified Raft system similar to those used by geo-distributed Database clusters such as CockroachDB or YugabyteDB. Raft is a consensus algorithm designed for distributed systems to ensure fault tolerance and consistency. It operates through leader election, log replication, and commitment of entries across a cluster of nodes.
We aim to build a database that stores key-value pairs mapping string (key) to string (value). The Raft cluster maintains this database and ensures fault tolerance and strong consistency. The client requests the server to perform operations on this database reliably.

Raft Modification (for faster Reads)

Traditionally, Raft requires the leader to exchange a heartbeat with a majority of peers before responding to a read request (How Raft handles read requests). If there are n nodes in the cluster, each read operation costs O(n). By requiring these heartbeats, Raft introduces a network hop between peers in a read operation, which can result in high read latencies. The latencies get much worse in the case of multi-region geo-distributed databases where the nodes are located physically far apart.

Leader Lease

It is possible to rely on a time-based “lease” for Raft leadership that gets propagated through the heartbeat mechanism. And if we have well-behaved clocks, we can obtain linearizable reads without paying a round-trip latency penalty. This is achieved using the concept of Leases.
This animation shows how leader leases work.

What is a Leader Lease?

Leases can be considered as tokens that are valid for a certain period of time, known as lease duration. A node can serve read and write requests from the client if and only if the node has this valid token/lease.
In Raft, it is possible for more than one leader to exist (which is why even read requests require a quorum to fetch the latest value), but leases are designed in such a way that, at a time, only one lease can exist. Only leader nodes are allowed to acquire a lease; therefore, the lease is known as the leader lease.

Behavior of Leader & Follower

The leader acquires and renews the lease using its heartbeat mechanism. When the leader has acquired/renewed its lease, it starts a countdown (lease duration). If the leader is

unable to renew its lease within this countdown, it needs to step down from being the leader.
The leader also propagates the end time of the acquired lease in its heartbeat. All the follower nodes keep track of this leader lease timeout and use this information in the next election process.

Leader Election

During a leader election, a voter must propagate the old leader's lease timeout known to that voter to the new candidate it is voting for. Upon receiving a majority of votes, the new leader must wait out the longest old leader's lease duration before acquiring its lease. The old leader steps down and no longer functions as a leader upon the expiry of its leader lease.

Implementation Details

0. Overview

In this assignment, you must implement the Raft algorithm with the leader lease modification. Each node will be a process hosted on a separate Virtual Machine on Google Cloud, and the client can reside either in Google Cloud's Virtual Machine or in the local machine. You can use **gRPC** (RECOMMENDED) or **ZeroMQ** for communication between nodes along with client-node interaction **[NO other communication library is allowed]**. Changes related to leader lease have been highlighted in cyan.

1. Pseudo Code

Many edge cases need to be handled while implementing the Raft algorithm. To ensure that all those edge cases have been handled correctly, you need to refer to the **pseudo code** (pg 60 to 66) while implementing. This lecture video explains the same. You can also refer to this medium blog implementing Raft in Python.

2. Storage and Database Operations

As mentioned in the introduction, we are building a database that stores key-value pairs mapping string (key) to string (value). The Raft Nodes serve the client for storing and retrieving these key-value pairs and replicating this data among other nodes in a fault-tolerant manner.

The nodes must be persistent, i.e., even after stopping the node, the data (logs) must be stored somewhere (in any human-readable format, e.g., .txt) along with other important metadata (commitLength, Term, and NodeID which the current node voted for in this term) and retrieved when the node starts again.

Follow the following naming convention for storing files:

For a node with node ID `x`, make a directory with the name `logs_node_x` and store the logs, metadata, and **dump file (refer to the Print Statements Section - 9 for dump file details)**. An example is shown below.

```
assignment/
├─ logs_node_0/
│   ├─ logs.txt
│   ├─ metadata.txt
│   ├─ dump.txt
├─ logs_node_1/
│   ├─ logs.txt
│   ├─ metadata.txt
│   ├─ dump.txt
```

The data (logs) will only store all the WRITE OPERATIONS and NO-OP operations as it is, along with the term number mentioned in the Logs section. An example log.txt file:

```
NO-OP 0
SET name1 Jaggu 0 [SET {key} {value} {term}]
SET name2 Raju 0
SET name3 Bheem 1
```

The operations supported for the client on this database are as follows:

SET K V: Maps the key K to value V; for example, {SET x hello} will map the key “x” to value “hello.” (WRITE OPERATION)

GET K: Returns the latest committed value of key K. If K doesn’t exist in the database, an empty string will be returned as value by default. (READ OPERATION)

Apart from the abovementioned operations, a node can also initiate an empty instruction known as the NO-OP operation. This operation has been elaborated on in the Election Functionalities section.

3. Client Interaction

Followers, candidates, and a leader form a Raft cluster serving Raft clients. A Raft client implements the following functionality:

1. Leader Information:

a. The client stores the IP addresses and ports of all the nodes.
b. The client stores the current leader ID, although this information might get outdated.

2. Request Server:

    a. It sends a GET/SET request to the leader node. (Refer to the Storage and Database Operations section)

        i. In case of a failure, it updates its leader ID and resends the request to the updated leader.

            1. The node returns what it thinks is the current leader and a failure message
            2. If there is no leader in the system, then the node will return NULL for the current leader and a failure message

        ii. The client continues sending the request until it receives a SUCCESS reply from any node.

Refer to the following RPC & Protobuf for the client:

```
rpc ServeClient (ServeClientArgs) returns (ServeClientReply) {}
message ServeClientArgs {
  string Request = 1;
}

message ServeClientReply {
  string Data = 1;
  string LeaderID = 2;
  bool Success = 3;
}
```

4. Standard Raft RPCs

Communication between nodes/servers requires two RPCs (AppendEntry and RequestVote). These RPCs have been explained in further detail in the original Raft paper:

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

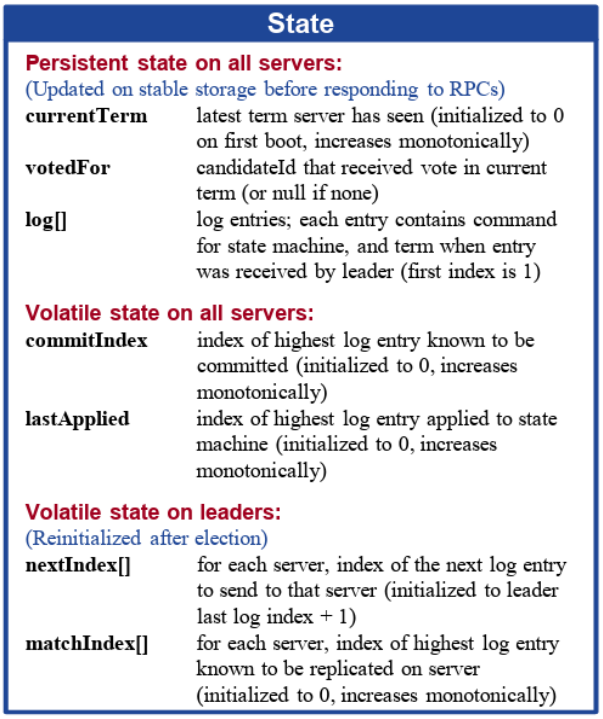
term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)



These RPCs can be modified as follows for leader lease:

The AppendEntry RPC must also send the lease interval duration whenever the leader starts the lease timer.

Through the RequestVoteReply RPC, the voters must also propagate the longest remaining duration time of an old leader's lease (known to that voter) to the new candidate it is voting for.

Please note that the heartbeat (entires[] field in the heartbeat) may not be empty. It may contain some entries (depending on what the follower is lacking).

5. Election Functionalities

Nodes must implement the following functionality with regard to the leader election process.

Start Election:

All follower nodes keep an election timer and listen to either the heartbeat event or a request for a vote event.

The election timeout should be **randomized** (5-10 seconds) to prevent multiple candidates from starting the election.

If no event is received in this duration, the node converts to a candidate, increments the term number, and sends vote requests to other nodes.

While receiving votes, the candidate must keep track of the maximum old leader lease duration received from voters.

Receive Voting Request

For a specific term, any node should vote for only one candidate.
A node votes for a candidate only when certain conditions are met.
Please refer to the pseudo-code for more information.

Leader State

If a candidate node receives a majority of votes, it moves to the Leader state.
Any new leader must also wait for the maximum of the old leader's lease timer (received through the RequestVote Reply RPC and its own) to run out before acquiring its own lease.

Once in the Leader state and the old lease timer has run out, the node starts its lease timer, appends a **NO-OP** entry to the log, and sends heartbeats to all other nodes.
Implementing Leader Lease requires additional changes, specified in detail in a later section on **Modifications to Standard Raft**.

6. Log Replication Functionalities

The nodes perform the following functionality in order to replicate the logs correctly to all nodes and maintain log integrity.

1. Periodic Heartbeats:
    a. The leader sends periodic heartbeats **(~1 second/heartbeat)** to all nodes to maintain its leader state.
    b. This heartbeat utilizes the same appendEntriesRPC.
    c. The leader also reacquires its lease at each heartbeat by restarting the lease timer and propagating the lease duration
        i. Lease duration or Lease period can be any FIXED value between 2 seconds and 10 seconds i.e. you can pre-decide and choose any value between 2 seconds and 10 seconds. Once you decide the lease duration, it remains the same throughout the execution of Raft.
    d. The leader needs to step down if it cannot reacquire the lease (did not receive a successful acknowledgment from the majority of followers within the lease duration).
    e. At each heartbeat, the followers also monitor and keep track of the duration of the lease.
2. Replicate Log Request:
    a. Whenever the leader receives a client SET request, the appendEntriesRPC is used to replicate the log to all nodes. (In the case of the GET request, the value can immediately be returned as long as the lease is acquired).

b. To synchronize a follower's log with the leader's, the leader identifies the latest matching log entry (same PrevLogIndex and PrevLogTerm), removes entries in the follower's log beyond that point, and transmits all subsequent leader entries.
c. Once the majority of nodes have replicated the log, a SUCCESS reply is sent to the client. Otherwise, it replies with a FAIL message. (Refer to the Client Interaction section)
d. Successful AppendEntries ensure log consistency for the remainder of the term.

3. Replicate Log Reply:
   a. A node accepts an AppendEntriesRPC request only when certain conditions are met. Please refer to the pseudo-code for more information.

7. Committing Entries

Conditions that need to be satisfied for the nodes to commit an entry have been specified in the pseudo-code. The following are the required functionalities:

Leader Committing an Entry

The leader should commit an entry only when a majority of the nodes have acknowledged appending the entry and,
the latest entry to be committed belongs to the same term as that of the leader.

Follower Committing an Entry

Follower nodes should use the LeaderCommit field in the AppendEntry RPC they receive in each heartbeat to commit entries.

8. Print Statements & Dump.txt

Add print statements wherever necessary to decipher what state each node is in and what operations are being performed. **You are required to print a dump file for each node with the naming convention mentioned in Section 2.**
This dump file **MUST** contain the following print statements:

- Before the leader sends heartbeats to the followers:
  "Leader {NodeID of Leader} sending heartbeat & Renewing Lease"
- If the leader fails to renew its lease:
  "Leader {NodeID of Leader} lease renewal failed. Stepping Down."
- When a new leader is selected, it waits for the old leader’s timeout to end:
- “New Leader waiting for Old Leader Lease to timeout.”
- When the election timer times out for any node:
  “Node {NodeID} election timer timed out, Starting election."

- Whenever a node becomes a leader:
  "Node {NodeID} became the leader for term {TermNumber}."
- When the follower node has crashed:
  “Error occurred while sending RPC to Node {followerNodeID}."
- Whenever a follower node successfully **commits (not appends)** an entry to its logs:
  "Node {NodeID of follower} (follower) committed the entry {entry operation} to the state machine."
  Example of an entry operation: “SET x 6”
- Whenever a leader node receives a request (SET or GET) from the client:
- “Node {NodeID of leader} (leader) received an {entry operation} request.”
- Whenever a leader node successfully **commits (not appends)** an entry to its logs:
  "Node {NodeID of leader} (leader) committed the entry {entry operation} to the state machine."
- Whenever a follower node receives **and accepts** an append entry request and appends the operation to its logs:
  "Node {NodeID of follower} accepted AppendEntries RPC from {NodeID of leader}."
- Whenever a follower node receives **and rejects** an append entry request:
  "Node {NodeID of follower} rejected AppendEntries RPC from {NodeID of leader}."
- Whenever a node **grants** a vote to a candidate in a particular term:
  "Vote granted for Node {Candidate NodeID} in term {term of the vote}."
- Whenever a node **denies** a vote to a candidate in a particular term:
  "Vote denied for Node {Candidate NodeID} in term {term of the vote}."
- Whenever a leader node steps down and becomes a follower:
  "{NodeID} Stepping down"

9. Test Case Scenarios

0. Start the cluster with 5 nodes. Wait for the leader to be elected and a NO-OP entry to be appended in all the logs.

[Leader Election]

1. Have the client perform 3 set requests and then 3 get requests

Verify if the set request has been replicated in each log.

Get requests should be returned immediately without any additional heartbeats by the leader.
[Log Replication, Basic Functionality]
2. Terminate one or two of the follower nodes and perform 3 SET requests and 3 GET requests. Restart the terminated followers.
Verify if the terminated follower rejoins the cluster with updated logs and state.
[Fault Tolerance, Follower Catch-up]
3. Terminate the current leader process. Wait for the new leader to be elected & perform 2 set and 2 get requests. Restart the terminated (old leader) process.
Verify if a new leader is elected & cluster continues operation without data loss in the logs. Also, ensure the old leader node should join the cluster as a follower with an updated state.
[Fault Tolerance, Leader Failure, Leader Election, Log Replication]
4. Terminate 3 (majority) follower nodes, and before the lease times out, send a Get request to the leader. Also, observe if the leader's lease times out after some time and the leader steps down to become a follower node.
The leader should be able to return the correct value for the GET request if it's within the lease interval. Timing is crucial, and the dump.txt can verify if the request was received timely. The lease should also time-out since the leader can't reach any of the follower nodes.
[O(1) Read Request, Leader Lease Timeout]
5. Terminate all the nodes except two follower nodes, send a Get and Set request to any of the followers.
Both Get and Set requests should fail ultimately as there is no leader present in the system. No node will be able to become the leader.
[Failure in absence of leader in the system]
Note: *We will not be checking network partitions explicitly as it is difficult to simulate a network partition without any hard-coding or changes to the default algorithm. But the above test case scenarios cover most of the aspects related to network partition.*

Assumptions

- You need only to create one cluster of the database using Raft. This cluster contains multiple nodes, but the number of nodes can remain fixed from the start (nodes need not be dynamically added or removed).
- You can use interrupts to stop a node/program (for example, Ctrl+C to stop the process). To emulate restarting the node, the program for the node will be executed again.