Yifei Gao(yg578), Yunhao Hu(yh2233)

This C++ program implements a basic version of the Raft consensus algorithm, designed to manage a distributed chat application. The program includes functionalities for node election, log replication, and handling client messages among distributed nodes. Below, the implementation, current limitations, and potential improvements are discussed.

Preface: Why Lab 2 Implementation Won't Work

Our goal is to always maintain and present the same sequence of messages that users type to their applications. However, our implementation of Lab 2 cannot guarantee this. Lab 2 implementation involves a distributed chat system that uses peer-to-peer communication to share chat messages across multiple processes, each running as a separate server instance. However, it primarily employs a gossip protocol for message dissemination and does not guarantee that all users will see messages in exactly the same order. Here is why:

- **Non-Deterministic Message Ordering:** The current gossip-based approach used for spreading messages (rumors) does not ensure a deterministic order of messages across all nodes:
- **Message Delay and Duplication:** Messages may arrive at different nodes at different times, and the same message may be received multiple times due to the gossip nature of the protocol.
- **Sequence Number Management:** Sequence numbers are generated independently based on each server's view of the message sequence, which may differ from one server to another due to network delays or message loss.
- **Lack of Global Consensus:** The implementation lacks a mechanism to achieve global consensus on the state of the chat log:
- **Independent Log Maintenance:** Each server maintains its own chat log independently, without a coordinated consensus mechanism ensuring all nodes agree on the exact same log entries at any given index.
- **Status and Rumor Messages:** The handling of `STATUS` and `RUMOR` messages is aimed at achieving eventual consistency rather than immediate consistency. Nodes respond to status queries with messages they believe others might be missing, which does not guarantee that all nodes will converge on a log with the same messages in the same order promptly.

Implementation Overview

- **Node State Management:** Nodes can exist in one of three states: Follower, Candidate, or Leader, managed using an enum class NodeState. State transitions are triggered by election timeouts, receipt of higher term numbers, or election results.
- **Message Types:** Different message types (e.g., RequestVote, AppendEntries, ClientMessage) are defined using an enum class MessageType to facilitate various operations within the cluster.
- **Networking and Message Handling:** Each node operates as a server that can accept connections and messages from other nodes or clients. Messages are parsed and processed based on their type to perform actions like voting, appending entries, or handling client requests.
- **Log Management:** Each node maintains a log of LogEntry objects, which contain commands and their terms. Log entries are replicated across nodes to ensure consistency, a core feature of the Raft algorithm. When a follower receives a message (`msg`) from a proxy, it directly forwards it to the

leader. Upon receiving the `msg`, the leader first adds it to the log list and then sends this log entry to all other nodes. When followers receive it, they respond and add the entry to their log list. After the leader receives responses from more than half of the followers, it commits the message — adding it to the `state_machine` and sending an acknowledgment (`ack`) back to the proxy. Subsequently, the leader notifies all followers that the message has been committed, prompting the followers to also add this message to their `state_machine`. When any node receives a request to `get chatLog`, it directly returns the contents of its `state_machine` to the proxy.

- **Election and Heartbeat Mechanism:** Nodes initiate elections and become candidates if they do not receive heartbeats within a random timeout. The leader sends periodic heartbeat messages to followers to maintain authority and prevent new elections.

Current Limitations

- **Error Handling and Robustness:** The program has minimal error handling, particularly in network operations and thread management, which could lead to stability issues.
- **Security:** The communication between nodes lacks encryption or authentication, making the system vulnerable to malicious attacks.
- **Scalability:** The implementation may not scale well with a large number of nodes due to its simplistic handling of network connections and state management.
- **Efficiency:** The current log replication mechanism does not handle log compaction, which can lead to excessive memory use over time.

Potential Improvements

- **Enhanced Raft Features:**
- Log Compaction: Implement snapshotting and log compaction to manage memory usage effectively.
- Dynamic Membership: Introduce changes to the cluster membership dynamically, allowing nodes to join or leave without disrupting the service.

Implementing Other Protocols:

- Paxos: As an alternative to Raft, Paxos also provides a solution for reaching consensus in distributed systems. Comparing its performance and complexity with Raft could be insightful.
- Zab: Used by Apache ZooKeeper for achieving consensus, Zab handles crash recovery more explicitly, which could enhance the robustness of the chat application.

Security Enhancements:

- Implement TLS/SSL for encrypted communications. Introduce authentication mechanisms to ensure that only authorized nodes participate in the network.

Performance Optimization:

- Optimize the use of threads and networking to handle larger clusters.
- Implement more efficient data structures and algorithms for managing logs and state information.

Usability Features:

- Provide a more sophisticated client interface for interacting with the chat system.

- Implement user authentication and commands within the chat application to enhance functionality.