Raft Key-Value Implementation Report

1. Overview

This implementation uses the Raft consensus protocol to create a distributed key-value store. The system partitions each value in the store as per the first level of the partitioning hierarchy and lets users perform three operations on the key-value pairs:

- The System has a Put Operation which facilitates the storage of key-value pairs.
- The Append Operation enables appending of values to existing keys.
- The Get Operation permits retrieval of values.

2. Architecture

2.1 Core Components

RaftNode Structure

- The system keeps records of the current term, voted candidate, and replicated logs on a persistent state.
- The system has volatile state that includes commit index and last applied entry.
- The system keeps track of leader state with next index and match index for each server.

2.2 Node States

2.2.1 Follower:

- The follower answers RPCs from candidates and leaders.
- The follower starts election if no heartbeat packet is received.
- The follower sets 300-500ms timeout for an election.

2.2.2 Candidate

- The candidate starts as a candidate in order to take the position of the leader.
- The candidate votes for himself during the elections.
- The candidate sends out requests in hope to obtain votes from all other servers.
- The candidate becomes leader if it receives majority votes.

2.2.3 Leader

- The leader sends out heartbeats to his followers.
- All requests made by a client are processed by the leader.
- Log replication is the other task carried out by the leader.
- When a majority of servers have replicated entries, the leader marks them as committed.

3. Key Features

3.1 Leader Election

- To avoid split votes, the system implements a randomized election timeouts approach.
- The system employs majority-based voting.
- The system carries out an election by terms.
- It is not complicated to have the system recognize when a leader fails: It happens automatically.

3.2 Log Replication

- Leader ensures that the logs are consistent in all nodes.

- The followers will copy the logs from their leader.
- The system uses a commit rule that is based on majority in order to enforce control.
- The consistency of logs is verified by the system when Append Entries is called.

3.3 State Machine Replication

- The system is based on a key-value store which serves as a state machine.
- The system inputs committed logs into the store.
- The system retains permanent consistency throughout every node.
- The system performs Put, Append, and Get commands.

3.4 Fault Tolerance

- The system elects a new leader automatically on failure
- The system ensures nodes have consistent logs.
- The system performs consensus using the majority.
- The system gets the state using log replication.

4. Implementation Details

4.1 RPC Implementation

1. RequestVote RPC

- The system uses Request Vote during leader election.
- The RPC transmits term and log details.
- The system adheres to the voting policy of the Raft paper.

2. AppendEntries RPC

- The system uses Append Entries for heartbeats and log replication.
- The RPC verifies logs consistency.
- The system updates follower's log and commit index.

4.2 Client Interface

1. HTTP Endpoints

- The system has an endpoint /put for submission of key and valus to be stored.
- The system has an endpoint /append for appending values..
- The system has an endpoint /getto fetch the data.
- The system processes votes using the endpoint /requestVote.
- The system associates logs replication with the endpoint /append Entries.

2. Interactive CLI

- The system provides command-line interface for operations.
- The system provides real time feedback through that.
- The system gives the possibility to check the status of the leader.

5. Usage

5.1 Starting Nodes

```bash
# Start three nodes
go run main.go -id 0 -port 8000 -peers "localhost:8001,localhost:8002"
go run main.go -id 1 -port 8001 -peers "localhost:8000,localhost:8002"
go run main.go -id 2 -port 8002 -peers "localhost:8000,localhost:8001"
```

5.2 Client Commands

```
put <key> <value>    # Store a key-value pair
append <key> <value> # Append to existing value
get <key>            # Retrieve a value
exit                 # Exit the CLI
```

6. Fault Tolerance Features

6.1 Leader Failure

- The system automatically detects leader failure through heartbeat timeout.
- The system begins a new election to chose a leader.
- Log consistency on nodes is maintained by the system.
- The system recovers state by replications.

6.2 Network Partitions

- Conflicts are resolved using a term based mechanism.
- Consensus is achieved through voting from the majority.
- Recovery from partitioning is automatic when the partition heals.

6.3 Log Consistency

- The system enforces log matching property.
- In AppendEntries, the system checks for consistency.
- The system truncates logs on inconsistency.

7. Performance Considerations

7.1 Timing Parameters

- The system uses 50ms heartbeat interval.
- The system implements 300-500ms randomized election timeout.
- The system uses majority-based commit rule.

7.2 Concurrency Handling

- Mutexs are used for synchronization in the system.
- Goroutines are used to handle RPCs in this system.

- This system enables concurrent log replication.

Conclusion:

The provided implementation uses the Raft consensus algorithm to create a robust and fault tolerant distributed key value store. This system accomplishes essential elements of the Raft architecture like leader election, log replication, and state machine replication. This implementation is intended for educational use but could be improved for production systems with additional features such as persistence and security measures.