CS 616 Distributed Systems
Project Report

Distributed Key-Value Store

March 2, 2025

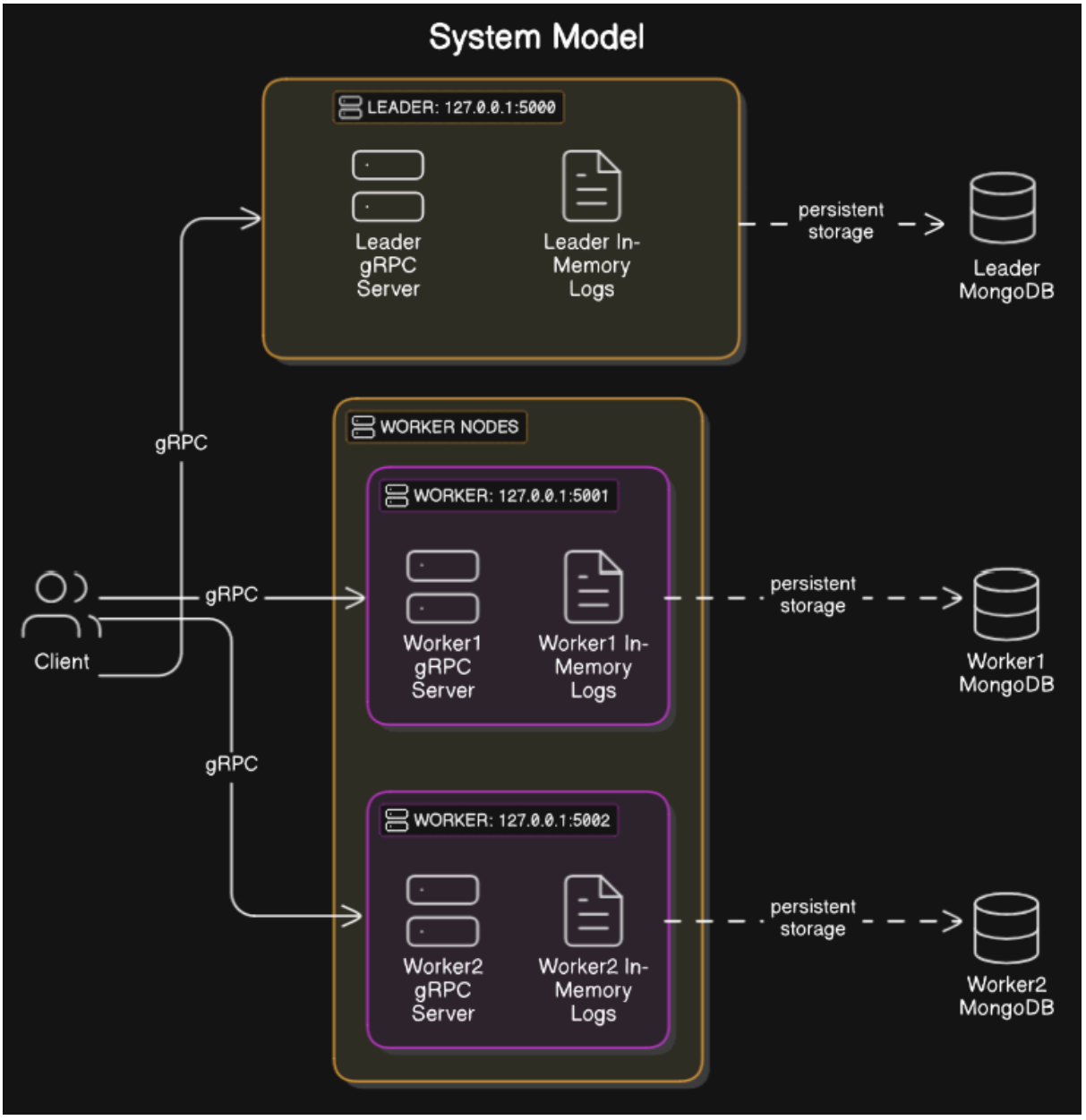
Kaushal Kothiya, 21110107 Anish Karnik, 21110098

1. Abstract

This report outlines the implementation, testing, and performance evaluation of a distributed key-value store system. The system is designed to provide consistent and reliable storage of key-value pairs across multiple server instances, with the ability to recover from failures. The client library provides an interface for interacting with the distributed store, supporting operations such as GET, PUT, shutdown, and initialization of connection with the system.

2. Architecture

A leader-follower model with multiple nodes. Each node runs a gRPC server and maintains an in-memory log and persistent storage via MongoDB. The client requests are redirected to the leader by the nodes that perform GET/PUT operations.



3. Implementation

Server Implementation

The server is implemented in Go and uses gRPC for communication between nodes. The server code (`server.go`) manages a distributed key-value store with the following features:

a. **Leader Election**: The system elects a leader among the available servers. The leader is responsible for handling all write operations (`kv_put`) and ensuring consistency across the cluster. We have used the bully algorithm to elect a leader within nodes. Servers with the latest(longest) log are screened, and the one with the highest IP is selected as the leader.
b. **Log Replication**: The leader replicates log entries to all followers. Each log entry contains a key, value, and an index. Followers commit these logs to their local MongoDB instance at regular intervals.
c. **Heartbeat Mechanism**: The leader sends periodic heartbeats to all followers(sends heartbeat every 5 seconds). If a follower does not receive a heartbeat within a specified timeout(8 seconds in our case), it initiates a leader election.
d. **Consistency Mechanism**: The leader handles all Put requests, appending entries to its log and replicating them to followers via LogCommit. Followers sync with the leader if logs diverge. Data is committed to MongoDB every 2 seconds (RegularLogCommit), ensuring eventual consistency. Logs are cleared periodically (every 60 seconds) based on the minimum log index across nodes.
e. **Persistence**: Data is persisted using MongoDB. Each server instance maintains its own MongoDB database, ensuring that data is not lost even if a server crashes.
f. **Failure Recovery**: The system can recover from failures by re-electing a leader and synchronizing logs from the new leader.

Client Implementation

The client implementation for the distributed key-value store is written in Go in `libkv.go`. Then, we compile the Go code in `libkv.go` into a shared library (`.so` file) that can be used by programs written in other languages, such as C. The client code is responsible for initializing the connection to the server, performing key-value operations (`kv_get` and `kv_put`), and shutting down the connection when done. Below is an explanation of the client implementation:

a. **Server List Initialization**: The client initializes the key-value store by calling `kv_init` and providing a list of server addresses. If the first server is unavailable, it tries to find the next server on the list, and so on.
b. **Key and Value Validation**: The client ensures that keys and values are valid before performing operations. Keys must be printable ASCII strings without special characters (`[` or `]`) and ≤ 128 bytes. Values must be printable ASCII strings and ≤ 2048 bytes.

c. **Error Handling**: The client handles errors gracefully, such as failed connections, invalid keys, or invalid values. The client checks the return values of `kv_put` and `kv_get` to handle errors, edge cases, invalid keys, or values.
d. **Shutdown**: After completing all operations, the client shuts down the connection to the server by calling `kv_shutdown`.

Protocol Specification

The protocol between the client and server is defined in the kvstore.proto file. Below given are all the Services:

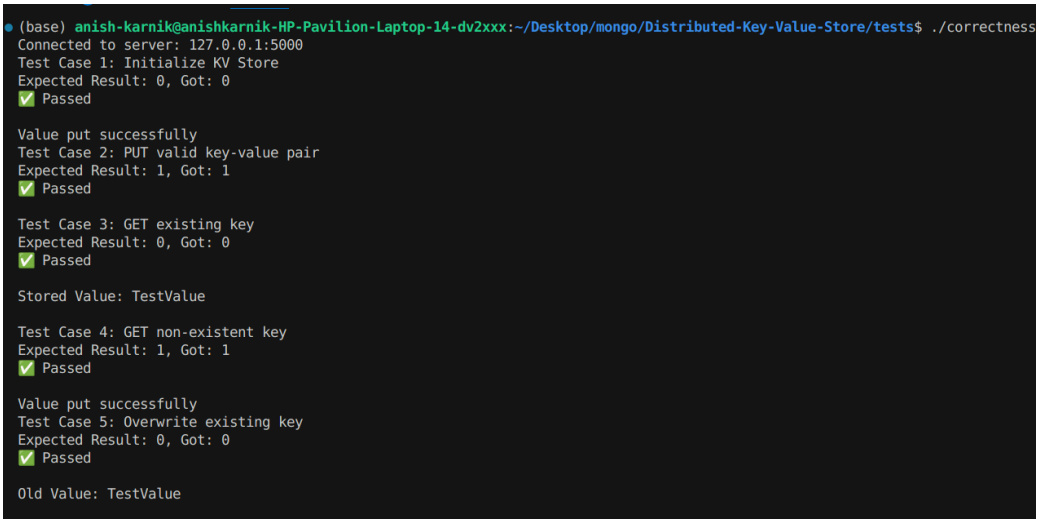
1. **Put**:
    - Handles key-value pair storage. The client sends a PutRequest containing the key and value, and the server responds with a PutResponse indicating success or failure.
2. **Get**:
    - Retrieves the value associated with a key. The client sends a GetRequest containing the key, and the server responds with a GetResponse containing the value and a boolean indicating whether the key was found.
3. **Heartbeat**:
    - Used by the leader to send periodic heartbeats to followers. The Empty message is used for both the request and response.
4. **UpdateLeader**:
    - Notifies followers of a new leader. The UpdateLeaderRequest contains the IP address of the new leader.
5. **GetLogIndex**:
    - Retrieves the last committed log index from a server. The server responds with a LogIndexResponse containing the index.
6. **SendMinLogIndex**:
    - Sends the minimum log index to all servers, ensuring that they clear logs up to that index. The server responds with a MinLogIndexResponse.
7. **ClearLogs**:
    - Clears logs up to a specified index. The ClearFromNum message contains the index from which logs should be cleared.
8. **LogCommit**:
    - Commits a log entry to the server. The LogCommitRequest contains the key, value, and log index, and the server responds with a LogCommitResponse indicating success or failure.
9. **GetLogEntry**:
    - Retrieves a specific log entry from the server. The GetLogEntryRequest contains the log index, and the server responds with a GetLogEntryResponse containing the key, value, and log index.

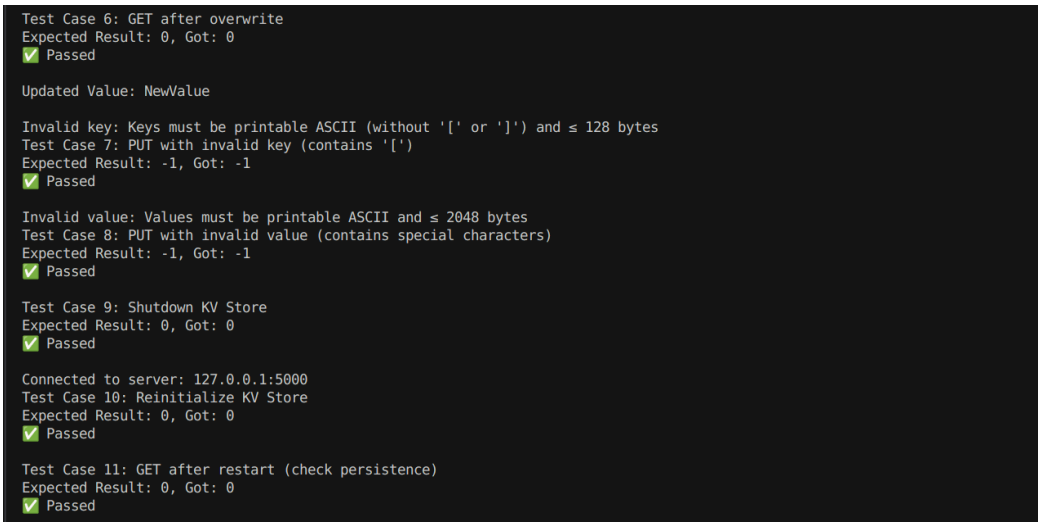
4. Testing

Correctness Tests

The correctness tests (`correctness.c`) ensure that the system behaves as expected under various conditions. The tests cover the following scenarios:

1. **Initialization**: Verifies that the client can initialize the key-value store.
2. **Key-Value Operations**: Tests the `kv_put` and `kv_get` operations, including overwriting existing keys and handling non-existent keys.
3. **Invalid Inputs**: Ensures that the system correctly handles invalid keys and values.
4. **Persistence**: Verifies that data persists across client restarts.





Recovery Tests

The recovery tests (`recoverytest.c` and `recoveryresult.c`) ensure that the system can recover from failures. The tests simulate a leader failure and verify that the system can still serve requests after a new leader is elected. The test checks whether the leader has sent the logs and if the workers have committed them. If, after shutting down the leader, we can still get our value, that means the workers have elected a new leader, and the data is not lost

```
(base) anish-karnik@anishkarnik-HP-Pavilion-Laptop-14-dv2xxx:~/Desktop/mongo/Distributed-Key-Value-Store/tests$ ./recoverytes
t
Initializing the key-value store
Connected to server: 127.0.0.1:5000
kv_init succeeded.
Putting the key-value pair
Value put successfully
Run the recoveryresult.c to find out whether after shutting the leader down, the system is still able to serve the requests
Make sure you dont put the leader IP in the servers list of recoveryresult.c
```

.

```
(base) anish-karnik@anishkarnik-HP-Pavilion-Laptop-14-dv2xxx:~/Desktop/mongo/Distributed-Key-Value-Store/tests$ ./recoveryres
ult
Initializing the key-value store
Connected to server: 127.0.0.1:5000
kv_init succeeded.
Expected Result: Yesssir, Got: Yesssir
✅ Passed

Stored Value: Yesssir
```

(In the above case, the leader was 127.0.0.1:5002, which was then stopped)

Performance Tests

The performance tests (`perfTest.c`) measure the system's throughput and latency under different workload distributions:

1. **Uniformly Random Distribution**: Tests the system's performance when keys are accessed uniformly at random.
2. **Hot/Cold Distribution**: Tests the system's performance when 90% of requests are directed to 10% of the keys (hotkeys) while 10% of requests hit one of the remaining "cold" keys.

Results on the Same Machine:

```
Performance Test: Uniformly Random Distribution
Uniform Random: 100 ops in 270889760 ns, throughput = 369.15 ops/sec, avg latency = 2708.90 µs, errors = 0

Performance Test: Hot/Cold Distribution (10% hot, 90% of requests on hot keys)
Hot/Cold: 100 ops in 206633147 ns, throughput = 483.95 ops/sec, avg latency = 2066.33 µs, errors = 0
```

Results on the Different Machines:

```
Performance Test: Uniformly Random Distribution
Uniform Random: 1000 ops in 21979208028 ns, throughput = 45.50 ops/sec, avg latency = 21979.21 µs, errors = 0

Performance Test: Hot/Cold Distribution (10% hot, 90% of requests on hot keys)
Hot/Cold: 1000 ops in 18172454670 ns, throughput = 55.03 ops/sec, avg latency = 18172.45 µs, errors = 0
```