Distributed Key-Value Store Report

Team Members:

1. Eshwar Dhande (23210038)
2. Aamod Thakur (23210038)

GithubLink : https://github.com/EshwarDhande/distributed-kv-store.git

1. Introduction

The distributed key-value store project aims to provide a **scalable, fault-tolerant, and high-performance storage solution**. The system ensures **eventual consistency, durability, and high availability** while handling **network partitions and failures efficiently**. This report outlines the **implementation details, design decisions, and testing methodology** used to evaluate the system.

2. Implementation Details

2.1 Client Implementation

- Implemented as a **gRPC client library** (`kv_client.py`) for seamless interaction with the distributed system.
- Provides functions for **Put, Get, Delete, ListKeys, and Backup operations**.
- Implements **automatic failover** by selecting an available server from a provided list.
- Uses **asyncio** for efficient non-blocking gRPC requests.

2.2 Server Implementation

- The server (`async_server.py`) is an **asynchronous gRPC server** that handles key-value requests.
- Uses **LMDB** for high-performance, disk-backed storage.
- Implements **multi-threaded processing** (`multiproc_worker.py`) to optimize request handling.
- Supports **replication** (`replication.py`) to ensure fault tolerance across multiple nodes.

2.3 Consistency Guarantees

- **Eventual consistency**: Updates propagate across all nodes within a bounded time.
- **Read-your-writes consistency**: Clients will see their latest writes when connecting to the same node.
- **Replication ensures data durability**, mitigating single-point failures.

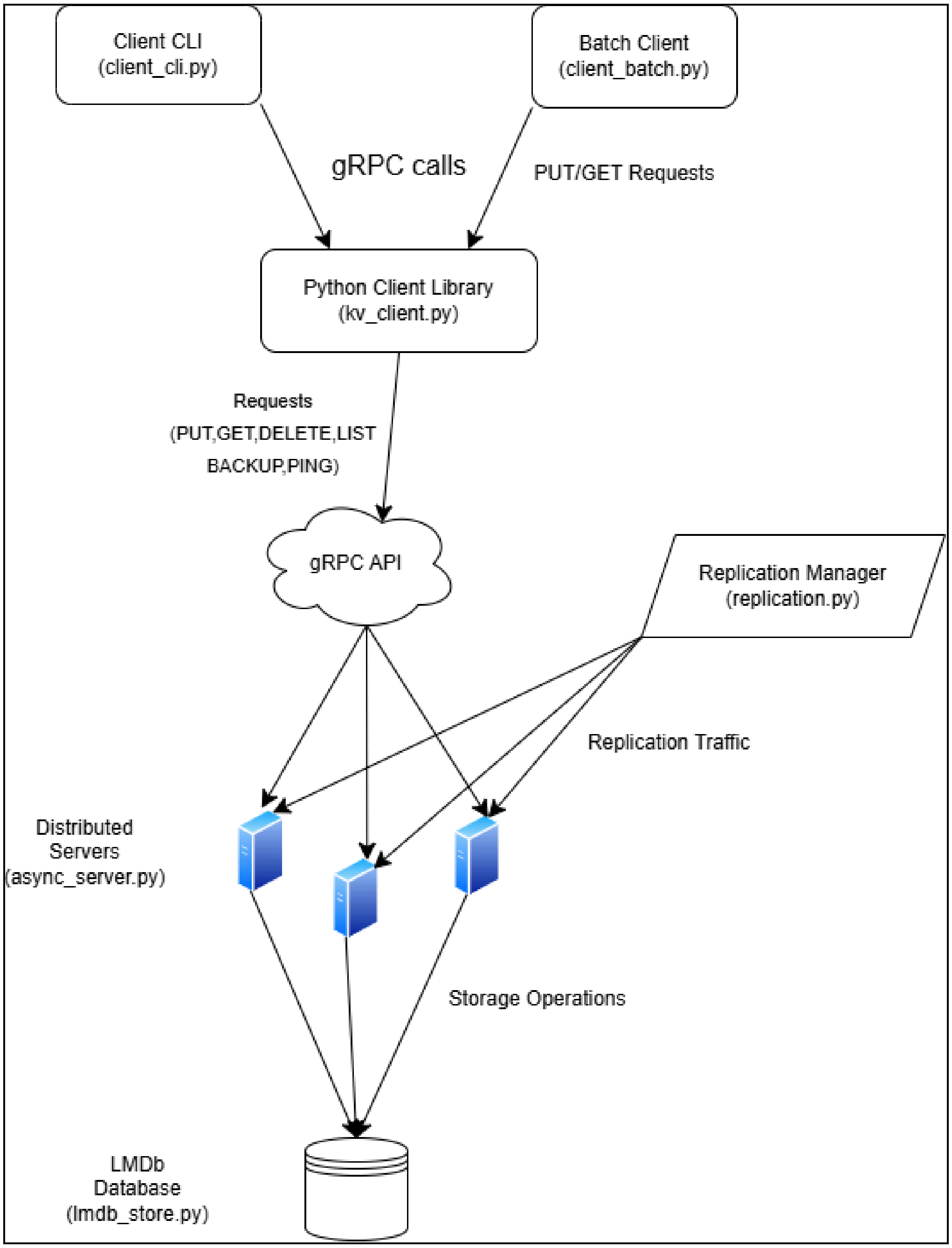

Fig: Project working Diagram

3. Tests

3.1 Correctness and Performance Tests

Test 1: Hot/Cold Key Distribution

This test evaluates how the key-value store handles a **hot/cold key distribution**, where a small set of keys (10%) receives the majority of requests (90%).

- **Purpose**: Measure **throughput and latency** for both read and write operations under skewed access patterns.
- **Significance**: Simulates real-world workloads in caching layers and database indexing, ensuring **efficient hot key contention management** while maintaining overall system balance.

Test 2: Mixed Read/Write Workload

This test evaluates the system under a **realistic mixed workload** where **70% of requests are GETs and 30% are PUTs**, following a **Zipfian distribution** to mimic real-world patterns.

- **Purpose**: Measure **throughput and latency** for read-heavy workloads while ensuring write operations remain efficient.
- **Significance**: Tests how well the system balances **read-heavy traffic** and **write throughput**, ensuring scalability and low-latency operations.

Test 3: Performance Under Failure

This test simulates **a node failure** while handling high-load operations to assess system resilience.

- **Purpose**: Evaluate how the system maintains **throughput and data availability** when a server crashes and restarts.
- **Significance**: Ensures that the system remains **operational, responsive, and capable of restoring normal functionality** after failure.

Test 4: Recovery from Failure

This test verifies that the **key-value store maintains data persistence** even after a **server crash**.

- **Purpose**: Store a key-value pair, **forcefully terminate the server**, restart it, and verify data integrity.
- **Significance**: Ensures that **LMDB provides durability guarantees**, confirming that writes persist across process restarts.

Test 5: Partial Failure Recovery

This test evaluates the system's **resilience to partial failures**, ensuring availability when one server crashes.

- **Purpose**: Store data, kill one node, and verify that the data is still accessible from other nodes.
- **Significance**: Ensures **high availability** in distributed systems, confirming that **remaining nodes continue serving requests correctly**.

Test 6: Network Partition

This test evaluates how the system handles **network partitions** and whether it correctly **propagates updates** after reconnection.

- **Purpose**: Write data, disconnect a server, update values, restart the disconnected node, and verify synchronization.
- **Significance**: Verifies **eventual consistency**, ensuring that **updates made during partitions propagate correctly once the network is restored**.

Test 7: Concurrent Writes

This test evaluates how the system handles **concurrent writes and reads** while ensuring **eventual consistency**.

- **Purpose**: Continuously update a key while another process reads values, ensuring that the final value matches the last written update.
- **Significance**: Confirms that **replication functions correctly** and that the system **eventually converges to the latest correct state** despite concurrent operations.

4. Documentation Overview

You can find detailed reports, test results, methodologies, API references, and design decisions in the **Docs** folder of the repository. This folder contains various files, including **Project_report.md**, **api_reference.md**, **design_decisions.md**, and specific testing methodology documents such as **tests_correctness_methodology.md**, **tests_failure_methodology.md**, and **tests_performance_methodology.md**.