Phase 1 System Design Report

Synchronous KV Store with LRU Cache and PostgreSQL Replication

Department of Computer Engineering,

Project Phase 1 Submission

Date: November 5, 2025

1 Introduction

This system implements a **synchronous key-value (KV) store** designed to handle high-performance GET and POST requests using both in-memory caching and persistent storage in **PostgreSQL** databases. The design uses multiple PostgreSQL instances running in containers and a lightweight HTTP server written in C++ using the `cpp-httplib` library.

The system aims to provide:

- Fast retrieval of data via an in-memory **LRU Cache**.
- Persistent fault-tolerant storage via **PostgreSQL instances**.
- Load distribution between multiple database nodes.
- A fallback read strategy for fault recovery using a third replica database.

2 System Architecture

The architecture of the system consists of the following major components:

1. **HTTP Server (C++):** Exposes RESTful endpoints for GET, POST, and DELETE requests and uses 200 concurrent threads to handle load.
2. **LRU Cache:** Maintains a fixed-size in-memory store to serve frequently accessed data.

3. **Database Cluster:** Three PostgreSQL instances (`db1`, `db2`, and `db3`) running in Docker containers.

4. **Replication:** Logical replication is configured such that `db3` subscribes to `db1` and `db2`.

The following figure illustrates the architecture of the system.

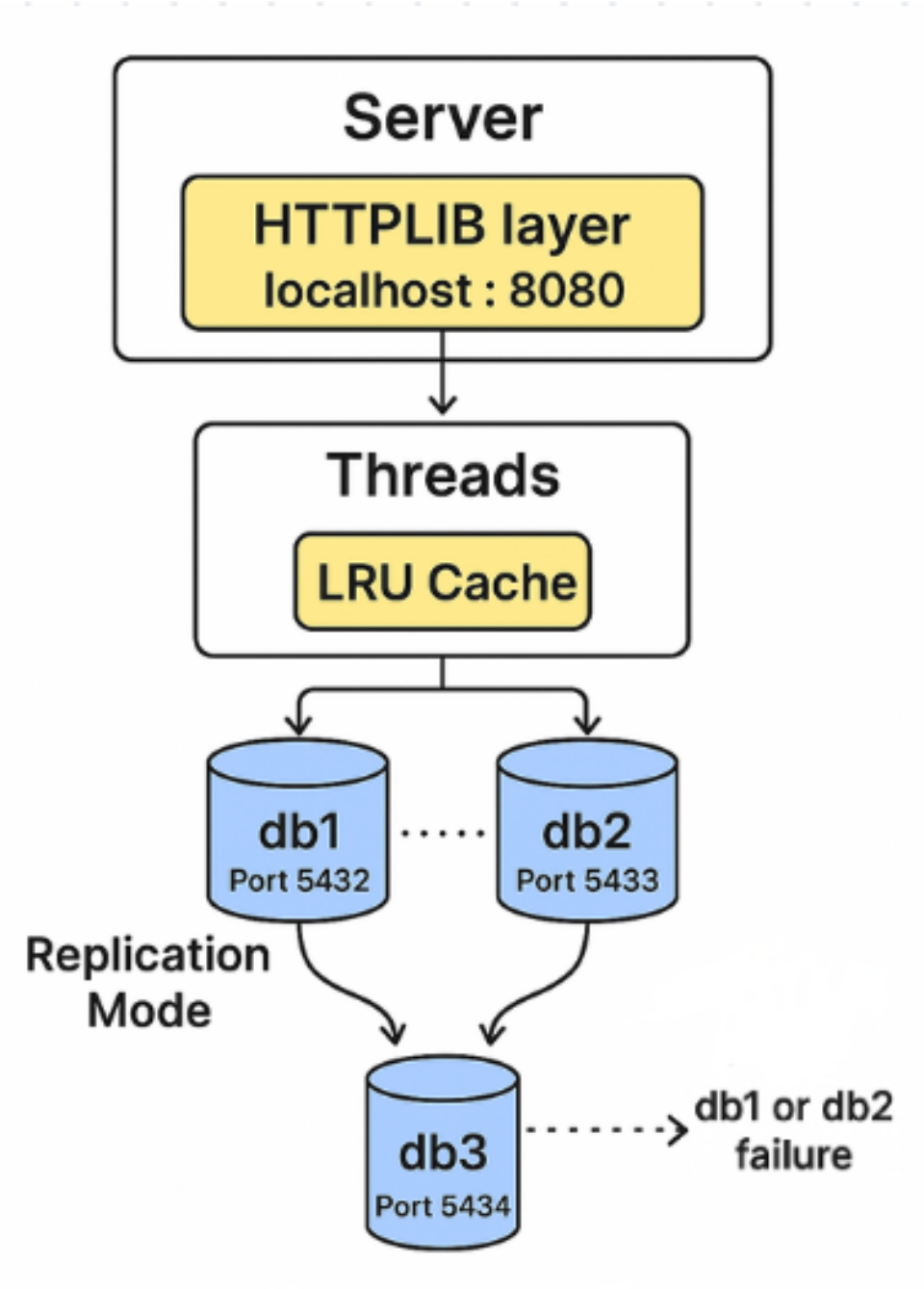

Figure 1: System Architecture of Synchronous KV Store with LRU Cache and Multi-Database Backend

3 System Components and Functionality

3.1 HTTP Server

The server is implemented using the `cpp-httplib` library. It listens on port 8080 and supports the following endpoints:

- `GET /<key>` – Retrieve a value for a given key.
- `POST /<key>` – Insert or update a key-value pair.
- `DELETE /<key>` – Remove a key-value pair.
- `GET /metrics` – Return cache and database performance metrics.

Each HTTP request follows one of two main execution paths — one that accesses **in-memory data** and another that accesses the **disk (database)**.

3.2 LRU Cache Layer

The cache layer is implemented using an `unordered_map` and a `list` for LRU ordering. It stores up to 100 key-value pairs in memory. The cache supports:

- Constant-time `get()`, `put()`, and `erase()` operations.
- Automatic eviction of the least recently used items when the cache is full.
- Thread-safety using `std::shared_mutex` for concurrent reads and writes.

Cache performance metrics are tracked using atomic counters for hits, misses, and access latencies.

3.3 Database Configuration

Three PostgreSQL containers are used in the deployment, each serving a specific role within the system architecture, as summarized in Table 1.

This configuration ensures data consistency and fault tolerance across all database nodes, allowing the system to recover from primary node failures without service disruption.

Table 1: PostgreSQL Container Configuration

Database	Port	Role and Configuration
db1	5432	Primary database; publishes changes via pub_db1.
db2	5433	Second Primary; publishes changes via pub_db2.
db3	5434	Replica; subscribes to both db1 and db2 for fault tolerance.

4 Execution Paths

There are two distinct execution paths in the system, depending on where the data is served from:

4.1 Path 1: In-Memory Access (Cache Hit)

When a GET request is received:

1. The server extracts the key from the request path.
2. It queries the **LRU Cache**.
3. If the key is found, the value is returned immediately from memory.
4. This is a low-latency path that avoids any database access.

Result: Fast response time and low resource usage.

4.2 Path 2: Disk Access (Cache Miss / Database Query)

If the key is not found in the cache:

1. The server uses a **hash-based routing** strategy to choose between db1 and db2.
2. The key is queried from the appropriate PostgreSQL instance.
3. If the database lookup succeeds, the value is cached in memory for future requests.
4. In case of database connection failure, a fallback query is made to db3, which holds replicated data from both primaries.

Result: Guaranteed data availability even during partial database failures.

4.3 Write / Delete Path (POST / DELETE)

For a `POST` request:

1. The key-value pair is inserted or updated in the corresponding database.
2. The same value is also stored in the LRU cache.
3. If one primary database fails, replication ensures that updates are available in `db3`. But if primary database fails no more requests of create or delete are accepted.

4.4 Operational Modes: Replicated and Direct

The system can operate in two distinct modes depending on performance, availability, and security requirements:

1. Replicated Mode (High Availability and Data Safety)
2. Direct Mode (High Performance)

Mode selection between **Replicated Mode** and **Direct Mode** is performed manually by the system administrator. This involves modifying the deployment configuration rather than dynamic server control. For example:

- In **Replicated Mode**, DB3 is initialized and subscriptions to DB1 and DB2 are created using PostgreSQL's `CREATE SUBSCRIPTION` commands.
- In **Direct Mode**, DB3 is not started, and only DB1 and DB2 containers are launched via the `docker-compose.yml` file.

This manual configuration approach ensures clear operational control and isolation between testing, performance evaluation, and production-ready replication setups.

5 Metrics Endpoint

The server also exposes a `/metrics` endpoint that reports:

- Cache hits and misses
- Cache size and hit ratio
- Total GET, POST, and DELETE operations

- Average database read/write latency

These metrics help in analyzing system efficiency, caching performance, and load distribution between the databases.

6 Dockerized Database Setup

Each PostgreSQL container is defined using Docker Compose and configured to:

- Run on isolated CPU cores using `cpuset`.
- Limit memory usage to 500 MB.
- Persist data on separate `/mnt/loop` volumes.

Example snippet for database configuration:

```
db1:
  image: postgres:16
  ports:
    - "5432:5432"
  volumes:
    - /mnt/loop1/pgdata1:/var/lib/postgresql/data
  command: postgres -c max_connections=500
```

7 Conclusion

This system integrates **in-memory caching, synchronous database writes, and logical replication** to form a fault-tolerant and high-throughput key-value store. The two distinct execution paths—**cache hits (memory)** and **database queries (disk)**—allow performance comparison between low-latency in-memory operations and persistent storage access. Further extensions also include: Asynchronous acknowledgment.

GitHub Repo: `https://github.com/monil2003/cs744-project.git`