DECS Report

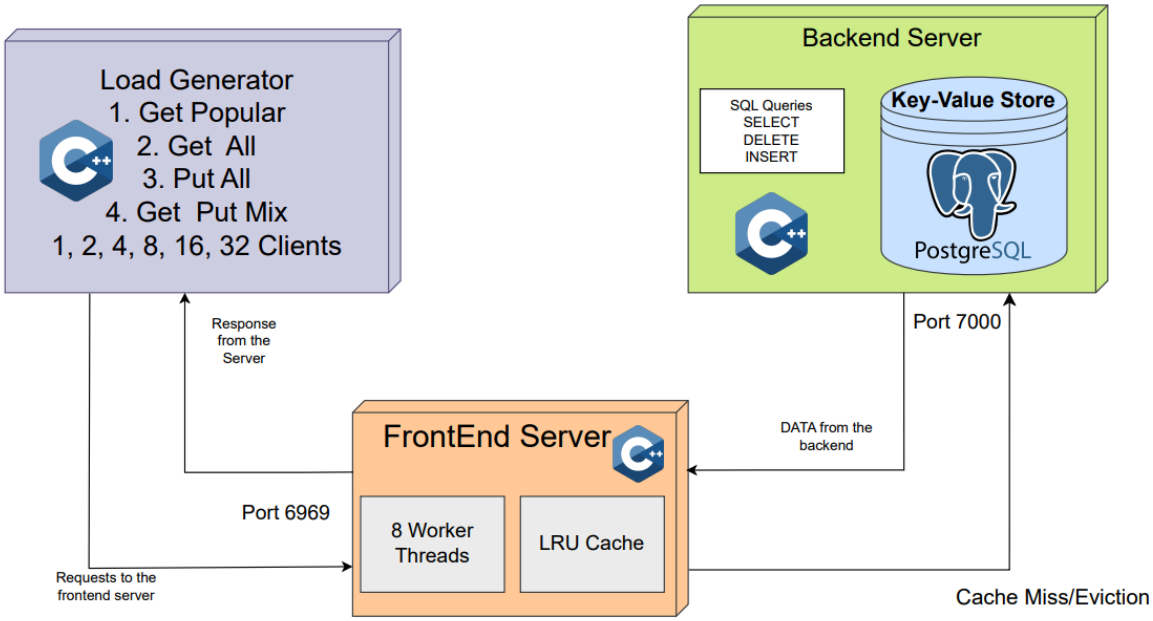
Performance Evaluation of a Distributed Key-Value Store

1. System Architecture

The system is designed as a high-performance, multi-threaded distributed key-value store consisting of three distinct components.

- **Load Generator (Client):** A custom multi-threaded cpp client that generates closed-loop HTTP traffic. It supports configurable workloads to stress-test specific system subsystems.
- **Frontend Server:** A C++ server utilizing a **Thread-Safe LRU Cache** and a **Worker Thread Pool** (8 threads). It employs a *Write-Back* caching policy, serving hits from RAM and flushing dirty nodes to the backend only upon eviction. Communication is handled via a custom HTTP parser.
- **Backend Server:** A persistent C++ layer that manages a single persistent TCP connection from the Frontend and executes queries against a **PostgreSQL** database.

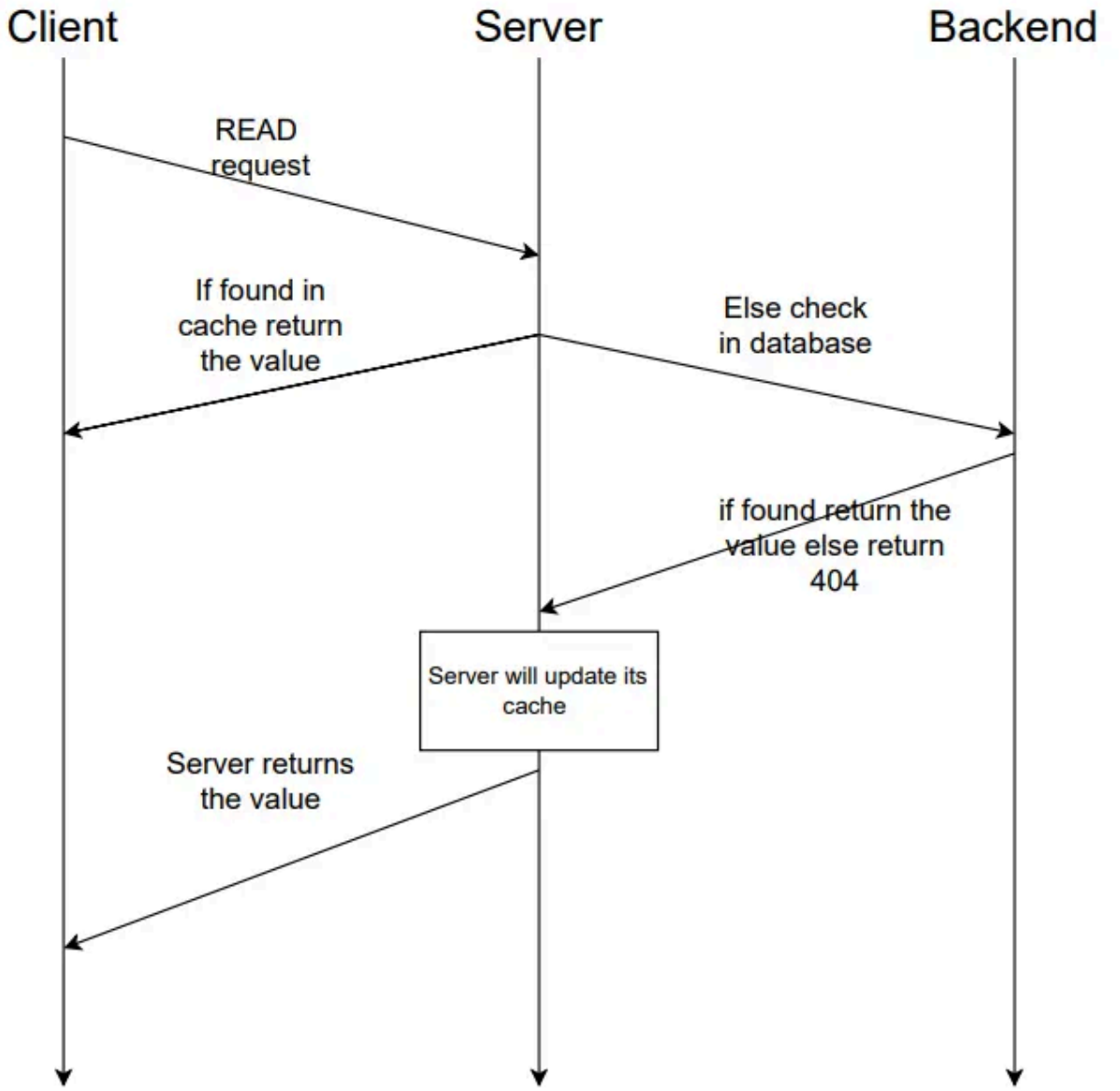
Architecture Diagram:



2. Load Generator Details

The load generator was implemented in C++ using standard socket libraries.

- **Model:** Closed-loop (Send Request → Wait for Response → Record Latency → Repeat).
- **Configuration:** Configurable number of concurrent threads (Clients) and test duration.
- **Workloads :**
  1. `GET_POPULAR` **:** Requests a small set of keys (1-50) that fit entirely in the LRU Cache. Expected behavior: 100% Cache Hits.
  2. `PUT_ALL` **:** Generates random writes with 512B payloads. This fills the cache and forces constant Evictions to the Database.
  3. `GET_ALL` **:** Generate only read requests with unique keys, ensuring that each request results in a cache miss and requires a read from the database.
  4. `GET + PUT` **:** Generates a random of mix of create, delete and read requests.



Data Flow for GET Request

1. A client sends a `GET /get?key=mykey` request to the **Front-End Server.**
2. The Front-End checks its in-memory LRU cache.
3. **Cache Hit:** : the key is found in the cache

- the corresponding node is moved to the front of the LRU Doubly Linked List.
- the value is returned to the client immediately. The request is complete.

4. **Cache Miss:** If the key is not in the cache:
    - The Front-End server sends an internal `GET /db_get?key=mykey` request to the **Back-End Server** (port 7000).
    - A Back-End worker thread queries the PostgreSQL database: SELECT value FROM KV_Store WHERE key = $1
    - The Back-End returns the value (or a 404) to the Front-End.
    - The Front-End **adds the new key-value pair to its LRU cache** or evicts an old item if full.
    - The Front-End returns the value to the original client.

Data Flow for Set Request

1. A client sends a `SET /set?key=mykey&value=myvalue` request to the **Front-End Server.**
2. The Front-End checks its in-memory LRU cache.
3. **Cache Hit:** : the key is found in the cache
    - its value is changed & boolean value of dirty is set to true
    - the corresponding node is moved to the front of the LRU Doubly Linked List.
    - The request is complete.
4. **Cache Miss:** If the key is not in the cache:
    - If number of pairs is already max(N) then the previous node of tail is selected to evict to database
    - A new node is created with key & value set to the values of specefied key & value
    - This new node is appended to the head of the Doubly Linked List.
    - This new key-value pair is added to the cache.

Data Flow for Delete Request

1. A client sends a `DELETE /delete?key=mykey` request to the **Front-End Server**
2. The Front-End checks its in-memory LRU cache.
3. **Cache Hit:** : the key is found in the cache
    - the node is detached from the Doubly Linked List
    - it is erased from the key-value store.
    - The request is complete.
4. **Cache Miss:** If the key is or not in the cache:
    - The Front-End server sends an internal `DELETE /db_delete?key=mykey` request to the **Back-End Server** (port 7000).
    - The Back-End server queries the PostgreSQL database for `mykey`.
    - The Back-End deletes or returns the value (or a 404) to the Front-End.
    - The request is complete

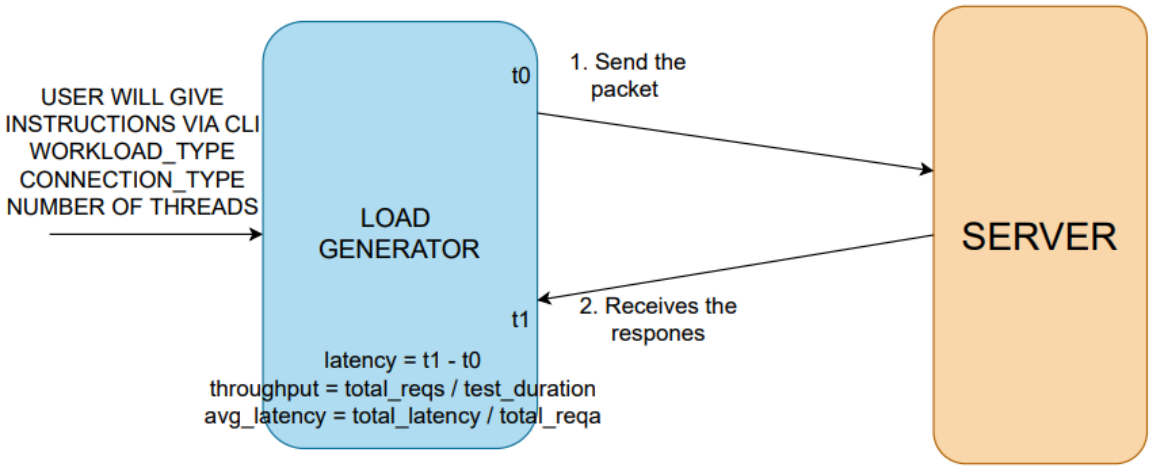
3. Experimental Setup

The system operates on a two-tier model:

- **Tier 1: Front-End Server (Caching Layer)**
    - This is the public-facing server that clients interact with.
    - It maintains a fixed-size **LRU (Least Recently Used) cache** in memory for extremely fast data access.
    - It uses a **thread pool** to handle many client connections concurrently.
    - It implements a **write-back** cache policy:
        - `SET` operations are written *only* to the in-memory cache and marked "dirty."
        - Data is only written to the Back-End database when an item is evicted from the cache or during a graceful shutdown.

- It communicates with the Back-End Server using an internal HTTP-like API (e.g., `/db_get`, `/db_set`).

- **Tier 2: Back-End Server (Persistence Layer)**
  - This is the internal server responsible for data persistence.
  - It translates internal API calls (e.g., `/db_set`) into parameterized **SQL queries** (`INSERT`, `SELECT`, `DELETE`) to safely interact with the database.



4. Load Generator

The load test was conducted on an **8-Core Intel i5-10300H** environment using **taskset command.**

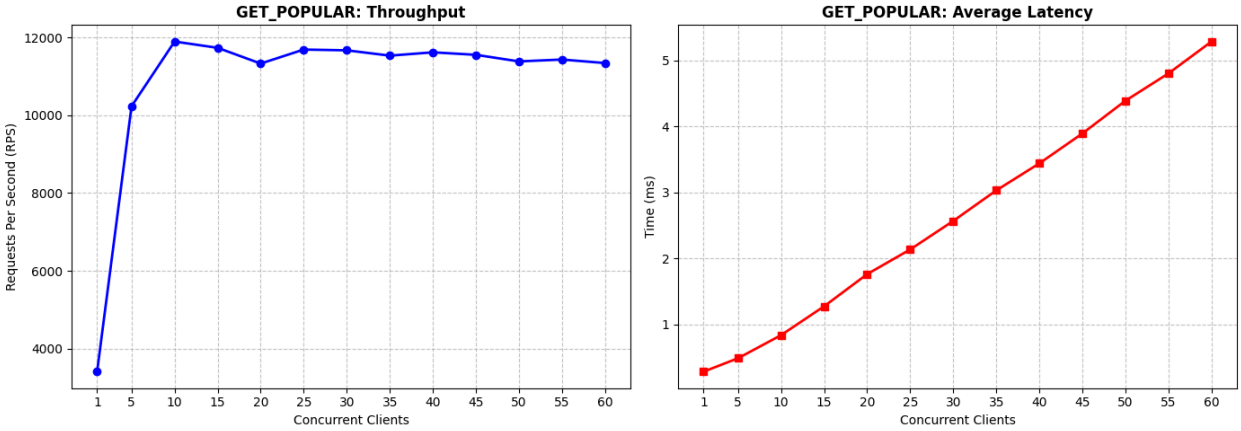
- **Cores 1-2:** Dedicated to the Load Generator (to prevent client-side bottlenecks).
- **Core 3:** Dedicated to the Backend Server & Postgres.
- **Cores 4-7:** Dedicated to the Frontend Server (4 cores for the 8-thread pool).

Metrics Collection:

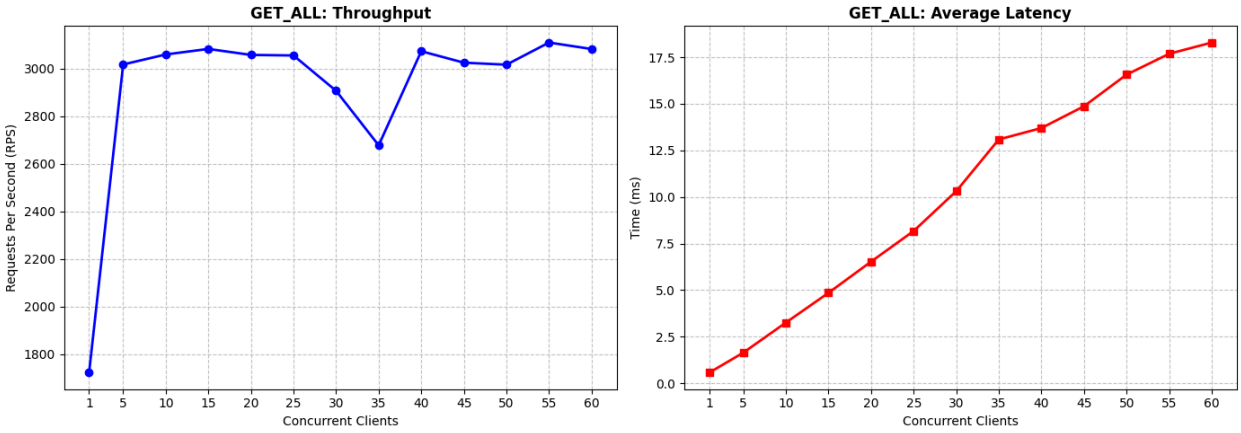
- **Throughput/Latency:** Measured directly by the Load Generator.
- **Resource Utilization:** Monitored using `htop` (CPU) and `iostat` (Disk I/O).

- **Cache Efficiency:** Instrumented using counters inside the C++ Frontend.

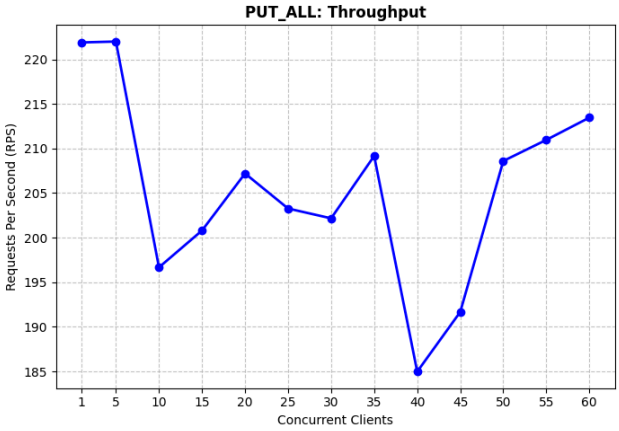
GET_POPULAR

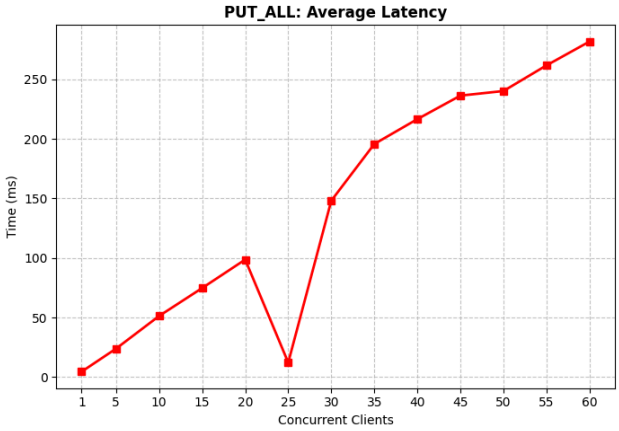


GET_ALL

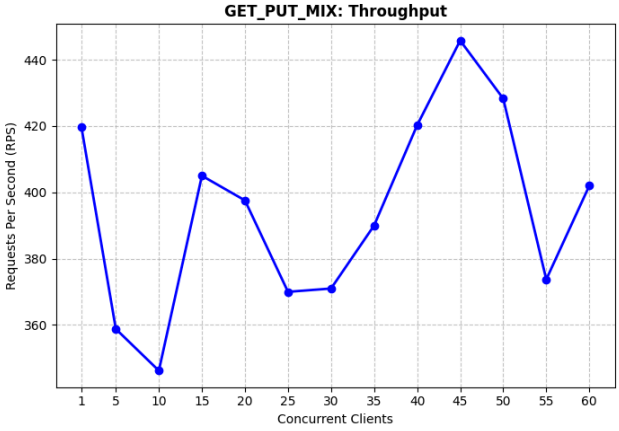


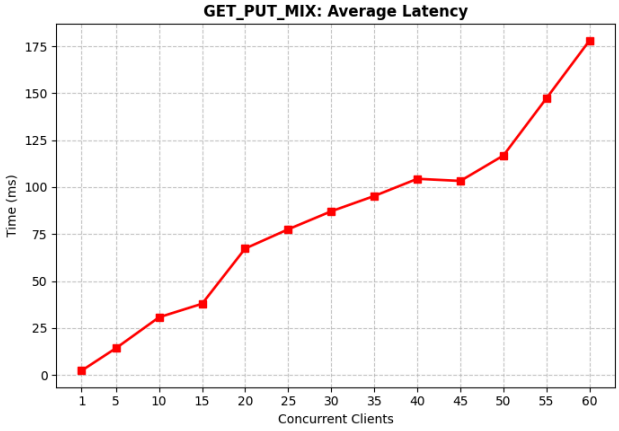
PUT_ALL





GET+PUT





Features

- **Multi-Threaded Server:** Server uses a thread-safe task queue and a pool of worker threads to handle high concurrency.
- **LRU Cache:** The Front-End implements a high-performance, fixed-size LRU cache to minimize database load.
- **Write-Back Caching:** `SET` operations are confirmed to the client instantly, with database writes happening asynchronously in the background.
- **Persistent Storage:** The Back-End uses PostgreSQL for robust and durable data storage.

- **Graceful Shutdown:** Both servers catch `SIGINT` (Ctrl+C). The Front-End will **flush all "dirty" data** from its cache to the database before shutting down, ensuring no data loss.
- **Safe Database Queries:** The Back-End uses parameterized SQL queries (`PQexecParams`) to prevent SQL injection vulnerabilities.