CS744 Autumn 2025

Project: HTTP-based KV Server

40 Marks, 20% weightage

The goal of the project is to build a multi-tier system, and perform its load test to identify its capacity and bottleneck resource. You are free to choose any system of your choice, subject to the following constraints. You must design and build a multi-tier client-server system with at least two tiers in the server, e.g., frontend and backend. You can reuse existing components, e.g., web servers or database servers, as required. The system should serve at least two different types of requests, using which you must be able to generate different workloads that have different types of performance bottlenecks. Please ensure that there are at least two different workloads with different performance bottlenecks, e.g, one workload that is CPU bound and another that is I/O bound. After building the system, you must perform a load test and systematically identify the various performance bottlenecks for the different workloads.

Below, we provide an example system description that can serve as a default choice. Please read through this example to get an idea of what we expect from you in the project, even if you choose a different system to work with. If you work with existing open-source databases or web servers, you must ensure that you understand the internal system architecture, e.g., threading model, in enough detail that you can present the system design and interpret the performance results correctly.

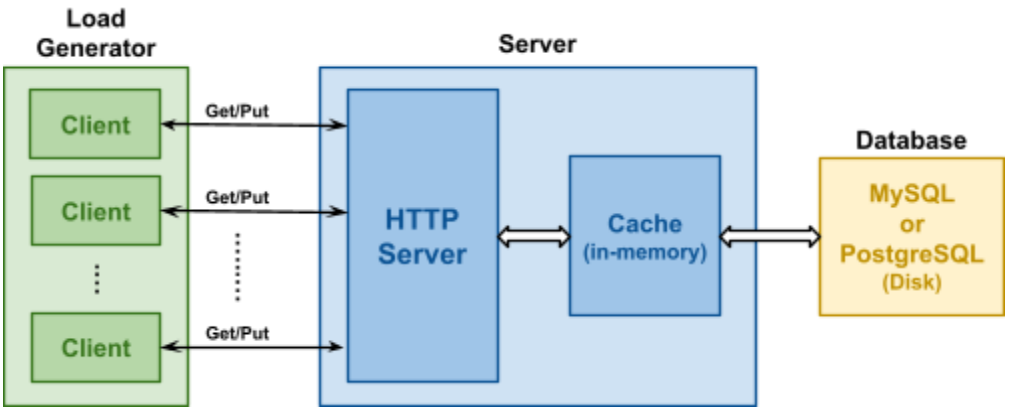
The project will be graded in 3 phases as described below.

Project timeline and grading

- **Phase 1 (10 marks):** By the end of the instruction period (Nov 7), you must submit a link to a GitHub repository which has all your code. You must then meet one of the TAs and show a demo of your working system. At this first stage, we will be expecting to see a functionally correct and working system.
- **Phase 2 (20 Marks):** After endsems, we will schedule a second demo with the TAs where you must demonstrate the complete working of your system, along with performance testing, and the bottleneck identification for different workloads. We expect you to have experimented with at least two workloads with different performance bottlenecks.
- **Final presentation (10 Marks):** The final evaluation in the project will be with a viva with the instructor, where you will present your final results and answer questions.

Example: HTTP-based key-value server with caching

One possible system to build and test is an HTTP server with a key-value (KV) storage system using C/C++/Go/Rust programming language. The system should support concurrent client connections, caching, and persistent database storage as shown in the figure below.



There are three main components in this system: a multi-threaded HTTP server with a KV cache (in-memory storage), a load generator (client side), and a database (disk storage). The load generator will emulate multiple clients, and generate client requests concurrently to the server. The server is built over HTTP, and uses a pool of worker threads to accept and process requests received from the clients. The clients generate requests to get and put key-value pairs at the server. The server stores all key-value pairs in a persistent database, and also caches the most frequently used key-value pairs in an in-memory cache (e.g., in-memory hash table).

Design of components

Below we describe the system components in more detail. Several HTTP libraries and database reference guides for different programming languages are provided at the end of the document.

1. **Server:** Implement an HTTP-based server that can accept and process concurrent HTTP client requests using a thread pool to handle concurrent client requests. The server should support *create, read,* and *delete* operations using RESTful APIs. You can decide the exact format of the request. The server must also have an in-memory cache to store frequently accessed key-value pairs in memory. The purpose of this cache is to enhance server performance by reducing repeated database access, much like real-world systems do. Below is a description of how you must handle the various types of requests.

- ***create:*** When a new key-value pair is created, store it both in the cache and in the database. If the cache is full, evict an existing key-value pair based on a chosen eviction policy (e.g., FCFS, LRU).
- ***read:*** When reading a key-value pair, first check the cache. If it exists, read it from the cache; otherwise, fetch it from the database and insert it into the cache, evicting an existing pair if necessary.
- ***delete:*** Perform all delete operations on the database. If the affected key-value pair also exists in the cache, delete it from the cache as well to synchronize it with the database and prevent inconsistent data. This approach ensures faster access for frequently used data while maintaining consistency with the underlying database.

2. **Database:** Connect a persistent KV store to the HTTP server, which stores data in the form of key-value pairs. Use any backend database tool, such as MySQL, PostgreSQL, etc., to maintain the data sent by the clients using create, update, and delete operations. Instructions required for the MySQL and PostgreSQL database integration are mentioned at the end of the document. Note that you should use a standalone database application and not a library database (e.g., SQLite).

3. **Load Generator:** Implement a closed-loop load generator that emulates client requests from a certain number of clients. You can implement the load generator as a multi-threaded program, where each thread emulates one client, i.e., sends a request to the server, waits for the response, and proceeds to send the next request after a response is received. You can use zero think time between requests to load the server with fewer clients. The number of threads and the time duration of the load test should be configurable, e.g., specified in the command line arguments. Each thread of the load generator must automatically generate its list of requests to the server, and must not get the requests from the user or from a file (as it slows down the request generation rate). The workload generated by the load generator can be any mix of *create*/*read*/*delete* requests. You can use different combinations of these requests to generate different workloads (which will expose different performance bottlenecks) during the load test. Below are some examples of workloads you can generate:
    a. **Put all:** Generate only *create/delete* requests for different keys from the clients, which will require access to the database for every request. This workload is likely to be disk-bound at the database.
    b. **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. This workload is also likely to be disk-bound at the database.
    c. **Get popular :** Generate only a small set of unique *read* requests that are repeatedly accessed by all clients. This workload will result in frequent cache hits at the in-memory cache of the server, and hence is likely to be CPU or memory bound at the server.

d. **Get+Put :** You can also generate a random mix of *create*, *delete*, and *read* requests. Some requests get hit on the cache, and some requests access the database after a cache miss. The performance bottleneck will depend on the specifications of your system and the type of requests you generate.

Your load generator should be able to gracefully handle all possible failure scenarios that may occur under high loads, e.g., you must think about how to handle socket read/write failures.

After all the load generator threads run for the specified duration, the load generator must compute (across all its threads) and display the following performance metrics before terminating.

- **Average throughput:** Average number of requests successfully completed per second by the server for the duration of the load test.
- **Average response time:** Average amount of time taken to get a response from the server for any request, measured at the load generator.

Note that the throughput and response time you measure will be a function of your workload (i.e., the mix of different request types you send to the server). So it only makes sense to compare throughput or response time values at different load levels if the measurements are all made at the same (or similar) workload. You must keep this point in mind if you are using any randomization when generating your workload.

Experiment setup

1. Use two separate (physical or virtual) machines to host the server with the database and the load generator, to easily separate their resource usages. If you cannot manage two separate machines, you must pin the server, database, and load generator processes to separate CPU cores, e.g., using the *taskset* command. Without a clear separation of load generator and server resources, your results of the load test will not make any sense.
2. You may assign as many hardware resources (CPU cores, memory, etc.) as required to your load generator in order to ensure that it is capable of generating enough load to saturate the server.
3. You must ensure that the system whose capacity is being measured (the server) is saturated by some hardware resource, while the system that is generating load (the load generator) is able to generate enough load.

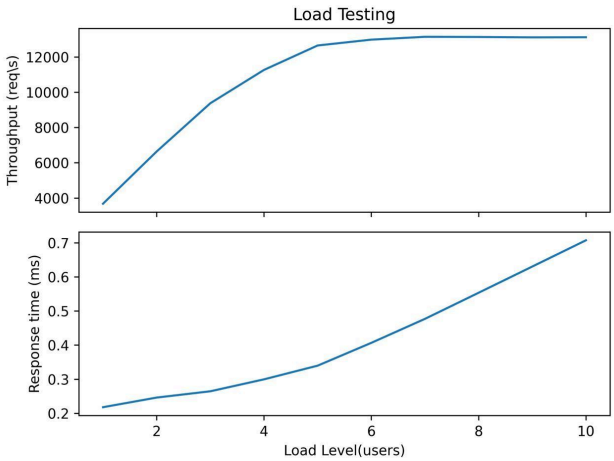
Instructions for load test

After completing the load generator and server setup, you must run multiple experiments to evaluate the server performance in the following manner.

- Run multiple experiments by varying the workload type (i.e., Get all/Put all/Get popular/Get+Put) and load level (i.e., number of concurrent load generating threads) at the load generator. Plot the average throughput and response time of the server with respect to the load level.
- In addition to the throughput and average response time on the client side, you must also calculate the resource utilizations of the various resources at all components, to help identify performance bottlenecks.
- Each experiment at a specific load level should run for at least 5 minutes to ensure that the throughput and response time reach steady-state values. The throughput and response time plots should include results from at least 5 experimental runs conducted at 5 different load levels.
- You will observe that the average server throughput increases with the load at first, but eventually levels off once the server reaches its maximum capacity. Meanwhile, the response time starts low and rises sharply as the server nears this limit. At the point of saturation, one or more hardware resources—such as the CPU or disk—typically reach close to 100% utilization.

Your final deliverables for the load test should be throughput and response time graphs of the system (example shown below) for different workloads, an estimate of system capacity, and an analysis of the performance bottleneck. You must provide detailed steps on how you performed the load test and how you identified the performance bottlenecks. You can also show other metrics (resource utilizations, cache hit rates etc.) to substantiate your claims of the identified performance bottleneck. ***We expect you to conduct a load test for at least two different workloads with two different performance bottlenecks.***

Finally, you must give a demo of the load test to the TAs, and also present your methodology and results in the final presentation.



Example results of load test

Reference Guides:

MySQL Installation:

- Following is a guide to set up your MySQL database: [Install mysql - DigitalOcean]
- Or, if you are familiar with Docker, use the following Docker image: [mysql - Docker Hub]

PostgreSQL Installation:

- Following is a guide to set up your PostgreSQL database: [Install PostgreSQL - DigitalOcean]
- Or, if you are familiar with Docker, use the following Docker image: [PostgreSQL - Docker Hub]

C/C++ Library:

- To make an HTTP server, you may use an HTTP library like civetweb or cpp-httplib.
- To make a KV cache, use the built-in C++ standard library, or you can integrate the KV store you made in C with the HTTP server.
- For database connectors, you can use the standard libraries provided by MySQL or PostgreSQL. Usage guide of libpq with PostgreSQL: [link]

Rust Library:

- To make an HTTP server, you can use an HTTP library like axum or actix. Example setup with axum: [link]
- To make a kv cache, use the built-in HashMap, or a concurrent HashMap like DashMap, or a HashMap-based caching library like moka
- For database connection, use a library like diesel or seaorm. Usage guide of diesel with PostgreSQL: [link]

Go Library:

- HTTP Server: You can use Go's built-in net/http package to create the HTTP server. [Usage guide for net/http]
- KV Cache: To make the KV cache, use a built-in map with a mutex for thread safety, or a concurrent library like Ristretto.
- Database Connection: For a simple connection, use the standard database/SQL package. For a full-featured ORM, use GORM. [Usage Guide for GORM]