In this lab, you will implement a simple in-memory key-value (KV) store as a client-server application running over TCP sockets. You will also learn the basics of performance testing by building a load generator to measure the capacity of your KV server. Before you begin, familiarize yourself with the basics of socket programming.

Part A: Multi-threaded server

The KV server socket program should take the port number on which to accept connections as a commandline argument and listen to new connections from clients on that port. The KV client socket takes the server IP address and port number as commandline arguments, and can issue requests to create, read, update, and delete key-value pairs to the server. Assume that keys are integers and values are arbitrary strings of size 1–64 bytes.

Your client must run in an infinite loop. In every iteration of the loop, it must prompt the user for a command to read/write key-value pairs. Upon receiving user input, the client must open a new TCP connection to the server, send the user command to the server, read the reply and display it to the user, close the connection to the server, and then prompt the user for the next input.

The user inputs and their semantics that must be implemented at the server are as described below.

- **create key value** should cause a new key-value pair to be created at the server. If the creation succeeds, the server must return a positive response to the client. If the key already exists, the server must return an error.

- **read key** must return the value corresponding to the key if it exists, and an error otherwise.

- **update key value** must update the value of an existing key with the new value. The server must return an error if the key does not exist.

- **delete key** must delete an existing key-value pair, and must return an error if the key does not exist.

Your server should have a multi-threaded architecture, with a fixed (but configurable) number of worker threads to service client requests. The main server process accepts new client requests and delegates the actual processing of the request to a worker thread. The key-value pairs are stored in an in-memory datastructure (e.g., hash table) that is shared across all server threads. You are free to choose the communication protocol between the client and server, i.e., the exact format of the messages sent by the client and the responses returned by the server.

Before you proceed to the next part, you must test the correctness of your KV store by having multiple clients connect to the server and successfully read and write key-value pairs.
Part B: Load generator

You will now measure the capacity of your server by building a load generator to rapidly fire requests at the server. You must modify your client written in part A above to act as a closed loop load generator. Your load generator will be a multithreaded program, with the number of threads and the duration of the load test specified as a command line argument. Each thread of the load generator will continuously generate requests to the server, wait for a response from the server, and proceed to fire the next request without any think time between successive requests. The request workload can be any mix of create/read/update/delete commands that makes sense, and must span at least 10K unique keys. Note that every request must be sent on a new TCP connection, i.e., the client must close the connection to the server and reconnect before sending the next request. Note that your load generator should not pause for user input—it must automatically generate requests according to your chosen workload. After all the load generator threads run for the specified duration, the load generator must compute and display the following performance metrics before terminating:

- Average throughput of the server, defined as the average number of requests per second processed by the server for the duration of the load test.
- Average response time of the server, defined as the average amount of time taken to get a response from the server, as measured at the client.

To measure the capacity of your server, you must run multiple experiments by varying the load level (i.e., number of concurrent load generating threads) at the load generator. Plot the average throughput and response time of the server as a function of the load level. The throughput and response time values must be averaged over a long enough duration (e.g., 5 to 10 minutes) to ensure that you are measuring steady state values. If all goes well, you will notice that the average server throughput initially increases with increasing load, but eventually flattens out at the server’s capacity. The response time of the server starts small, but rapidly grows as the server approaches its maximum capacity.

The end result of your load test must be an estimate of your server’s capacity, obtained from the plots of the throughput and response time of the server at varying levels of load. You may use a server running on one CPU core for your capacity measurement. You may use as many worker threads as needed to saturate the server.

A few things to keep in mind when running this load test:

- When you find that the server’s throughput has flattened out, you must verify that the server’s capacity is limited by some hardware bottleneck (e.g., CPU or network). If you find that the throughput of your server is flattening out even with no apparent hardware bottleneck, you must investigate why your server is not able to handle more requests. Perhaps your load generator is not generating enough load, or your server does not have enough worker threads to handle all the requests coming in. Or perhaps you are printing out too much debug output to the screen, causing the server to wait for I/O most of the time. You must carefully debug your experiments until you are convinced that you have really saturated some hardware resource at your server.

- Note that the throughput and response time you measure will be a function of your workload (i.e., the mix of read and write requests 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 workload.
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.

It is recommended that you use two separate (physical or virtual) machines to host the server and the load generator, to easily separate their resource usages. If you cannot manage two separate machines, you must at least pin the server and client processes to separate CPU cores, e.g., using the `taskset` command.

**Part C: Multicore scalability**

We will now understand how the capacity of your server increases with increasing number of CPU cores. Ideally, the performance of a CPU-bound application should increase linearly with increasing CPU cores. Now, measure the saturation throughput of your server as you increase the number of CPU cores from 1 to, say, 4, and understand how its capacity scales with CPU cores. If you find that your server performance is not scaling well, investigate the reason for this poor scaling (e.g., by profiling your code) and try to fix it to improve scalability.

In this part, you must configure the worker threads of your process suitably to ensure that there are enough threads to saturate the server CPU as you increase the number of cores. Also verify that your load generator is able to saturate the server at all times.