Lab: In-memory Key-value Store

Goal

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

Before you begin, familiarize yourself with the basics of socket programming. Simple client and server socket programs are provided with this lab for your reference.

Part A: Multi-threaded server

In this part, you will write two programs: a KV server `kv-server.c` and a KV client `kv-client.c`. The KV server maintains key-value pairs, while the KV client accepts user commands to manipulate the key-value pairs and executes these commands by communicating with the KV server over sockets. You are given that keys are integers, and values are strings of arbitrary length.

The KV server socket program should take two command line arguments: the IP address and port number on which to accept connections. It must then spawn a server socket to listen to new connections from clients on that port. There can be multiple KV servers in the system.

The KV client program can run in one of two modes: interactive and batch. In interactive mode, the client program must prompt the user for commands. Upon receiving a user command, the client must execute the command by communicating with the KV server as required, display the result of the execution to the user, and then prompt the user for the next input. In batch mode, the KV client should read a file with one user command per line and execute it before moving on to the next line. The output of the commands must be displayed to the terminal in both modes.

The KV client program can take one or two command line arguments. The first (mandatory) argument specifies the mode of operation: `interactive` or `batch`. If the first arguments indicates a batch mode of operation, the second command line argument must specify the filename corresponding to the command batch. There is no need for a second command line argument when operating in interactive mode.

Listed below are the user commands and their semantics that must be supported by the KV client.

- `connect <server-ip> <server-port>` should cause the KV client to open a socket and connect it to a remote KV server specified in the command. The client program should display
“OK” if the connection has been successfully established, and an error message otherwise. Note that a client can have an active connection to only one KV server at any point of time. All other commands below must only succeed only if there is an active connection to a server.

- **disconnect** should cause the client to close its active connection to the remote server. The client should display “OK” after disconnection completes. After disconnection, the user can ask the client to reconnect to a (possibly different) server again at a later time. However, if the user asks to connect to a second server while the first connection is still active, the KV client should print an error message.

- **create <key> <value-size> <value>** should cause a new key-value pair to be created at the server. If the creation succeeds, the client must display “OK”. If the key being requested already exists at the server, or if the creation fails for any other reason like the absence of an active server connection, a suitable error message must be displayed to the user.

- **read <key>** must display the value corresponding to the key if it exists at the connected server, and an error otherwise.

- **update <key> <value-size> <value>** must update the value of an existing key with the new value at the connected server. Note that updates can only be done to existing keys, and the client must return an error if the key does not already exist.

- **delete <key>** must delete an existing key-value pair, and must display an error if the key does not exist or the delete fails due to some other reason.

Note that the above messages specify the commands given by the user to the KV client. 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. Further note that one command can be split over multiple network packets between the client and the server, e.g., when the value is larger than the maximum packet size permissible by the network.

While a KV client can be connected to only one server at any point of time, a server can be concurrently connected to multiple KV clients. Further, the server should maintain a common KV store across all clients. That is, the key-value pairs created by one client should be visible to all other connected clients.

In order to correctly handle multiple concurrent clients, 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 data structure (e.g., hash table) that is shared across all server threads. Because the size of the values is not known a priori, the server must dynamically allocate memory to store values, and must free this memory when the value is no longer in use.
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 write a KV client \texttt{kv-multi-client.c} that acts as a closed loop load generator. You are not expected to modify your server in any significant manner from what you have in part A.

Your load generator will be a multi threaded program, with the number of threads and the duration of the load test specified as a command line arguments. Each thread of the load generator will send a command to the server, wait for a response from the server, and proceed to fire the next command without any think time between successive commands. Each thread of the load generator must automatically generate its list of commands to the server, and must not get the commands from the user or from a file (like in the case of part A). The load generator threads also need not display the result of every command to the terminal. The workload generated by the load generator can be any mix of connect/disconnect/create/read/update/delete commands that makes sense, and must span at least 10K unique keys.

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 of the server**, defined as the average number of requests per second successfully processed by the server for the duration of the load test. A request can be any of connect/disconnect/create/read/update/delete requests sent by the clients.

- **Average response time of the server**, defined as the average amount of time taken to get a response from the server for any request, as measured at the load generator.

After writing the code for your load generator, you must run a load test in the following manner. 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. Each experiment at a given load level must run for at least 5 minutes to ensure that the throughput and response time have converged to steady state values. Your plots for throughput and response time should include at least 5 experimental samples at 5 different load levels. 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. At the load when the server hits capacity, you will also notice that some hardware resource (e.g., CPU or network) has hit close to 100% utilization. Of course, the capacity you measure and the bottleneck resource you identify will depend on your chosen workload, and the configuration of your client and server machines.

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) and an identification of the bottleneck resource at saturation.

You may use a client and server running on one or more CPU cores for your capacity measurement. You may use as many worker threads as needed to saturate all the cores of 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 \texttt{taskset} command. Without a clear separation of client and server resources, your results of the load test will not make any sense.

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 (or similar) workload. You must keep this point in mind if you are using any randomization when generating your 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. You must ensure that the system whose capacity is being measured (the KV server) is saturated by some hardware resource, while the system that is generating load (the load generator) is not saturated and is able to generate enough load.

You may find that your server cannot process all client requests, especially at high loads. You must carefully write your code to gracefully handle all possible failure scenarios that may occur under high loads. For example, if a load generator thread fails to connect to the server, it must retry again before proceeding to send requests for creating key-value pairs.
Submission instructions

- You must submit the files `kv-server.c`, `kv-client.c` and `kv-multi-client.c`.
- You must also submit a PDF/text report with your code.
- Place these files and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Your report must contain the following:

- Instructions to compile and run your client, server, and load generator programs.
- A description of your test setup, and the relevant hardware and software configurations of your client and server systems, for the load test in part B.
- A description of your workload, i.e., the mix of various commands issued to the server by each load generator thread in part B.
- Plots of throughput and response time of the server with increasing load levels, as generated in part B.
- A short description of the bottleneck resource of your server at saturation, including how you identified the bottleneck (e.g., what was the CPU utilization you measured?) and why the bottleneck is justified (e.g., was your workload CPU-intensive?).

Grading

We will run your code to test the correctness of your client-server system, as well as reproduce some of your load generator results. We will also read your report and code to assess correctness.