DECS Assignment: 4

Building a Key Value Store

Mitul Tyagi, Roll no. : 193079017
Tarun Saurabh, Roll no. : 203050009
Rajendra, Roll no. : 203050094

13 November 2020

1 Introduction

This assignment implements a performant, salable key-value server. The design implements a key-value server to which multiple clients can connect. Once connected, the clients continuously send requests for either getting the value associated with a key or putting in a new key-value pair. The server uses a cache (**KVCache**), which is backed by a persistent storage (**KVStore**) to store key value pairs since all keys cannot be stored in memory.

The diagram shown below describes the architecture of the server design.

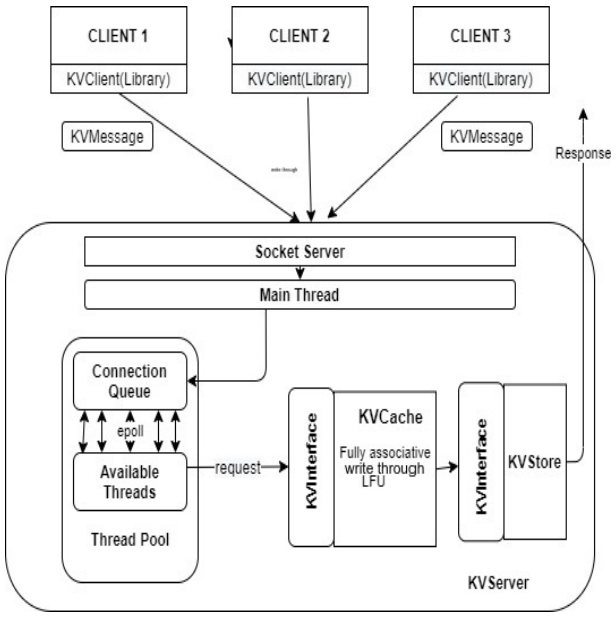

Figure 1: Architecture of KVServer

2 Architecture of Key Value-Store

2.1 Socket Server/Main thread

KVServer consists of a main thread that performs the following steps:

- It reads a config file (**server.conf**) and sets up a listening socket on a port specified in the config file.
- It creates **n-worker threads** at startup which are specified in the config file.
- It then performs **accept()** on the listening socket and pass each established connection to one of the n threads in a round robin fashion.

- Each worker thread sets up and monitor an **epoll context** for the set of connections it is responsible for.
- It runs an infinite loop that monitor its client connections for work (in addition, if there are new connections for the thread, it adds it to the **epoll context**). After reading requests from clients, it will process those requests, and write a response back to the client.

2.2 Thread Pool

2.3 KVCache

The server has limited memory (specified via the **server.conf file parameter** that lays out how many key value entries the server is allowed to hold in memory) and so this will serve as cache to the files that will hold the persistent representation of the data. The cache is a fully associative cache with **LFU** replacement/eviction policies. Multiple threads can access this cache using **Read/Write Locks**. Read locks are shareable which means multiple GET requests are served concurrently and Write locks are only used by one thread at a time for PUT/DEL requests. The locking granularity is on a single entry in case of KVCache.

2.4 KVStore

All keys and corresponding values are stored on disk in files. The meta data about key positions and file name mapping is stored in main memory - using a bitmaps kind of approach. After the PUT/DEL request is executed for the cache, the same is done for store's meta-data and the file where only one writer can access it, but while reading multiple readers' can read from the metadata. Rather than searching file on disk, it is searched on meta data, which in turns reduces the search time and when the file name or path is obtained, we get data directly from it.

3 Architecture of KVClient

KVClient uses following interfaces:

- GET (Key k): Retrieves the key-value pair corresponding to the provided key.
- PUT(Key k, Value v): Inserts/Overwrites the key-value pair into the store.
- DEL(Key k): Removes the key-value pair corresponding to the provided key from the store.
- 'Keys' and 'values' are always strings with non-zero lengths.
- Each key and value cannot be greater than 256 bytes.

- When **PUTing** a key-value pair, if the key already exists, then the value is overwritten.
- When **GETing** a value, if the key does not exist, return an error message.

4 KVClientLibrary

This is a library, which will encode and decode the request and response messages. It has three main APIs for GET, PUT and DEL.

5 KVMessageFormat

Each request and response is a 513 Byte messages. First byte of the message is the status code, and this status code will uniquely identify different requests and responses. Next 256 Bytes will be key. The last 256 Bytes of the message will be value. In case of response appropriate response messages are sent from the server.
Status Codes for request message
GET: 1
PUT: 2
DEL: 3

Status Codes for response message
Success: 200
Error: 240 (with the appropriate error message)

6 How to run the Program

A README file is provided which has the detailed instructions for running the server and the client. Images shown below give commands that are used most frequently

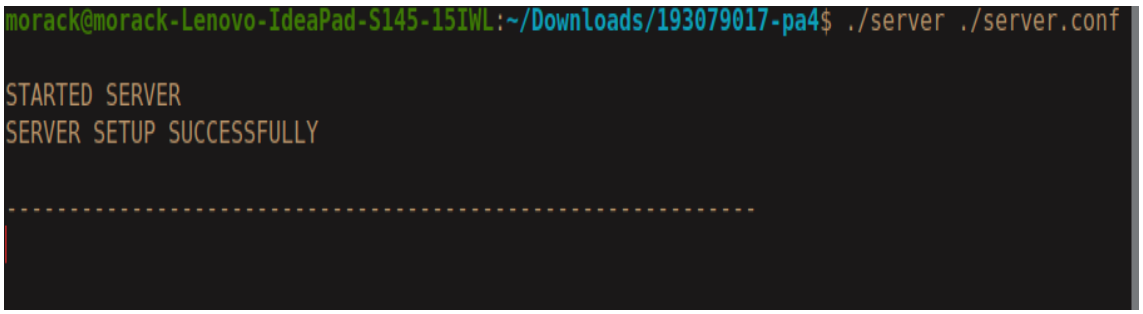

Figure 2: Method to run server

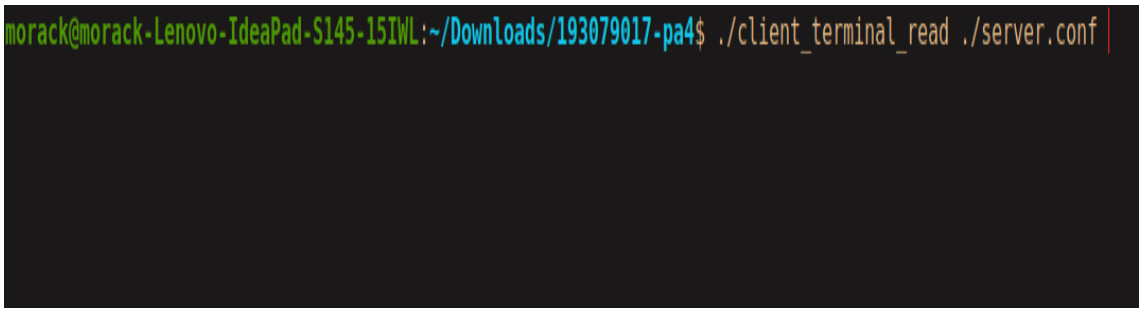

Figure 3: Run client_terminal_read

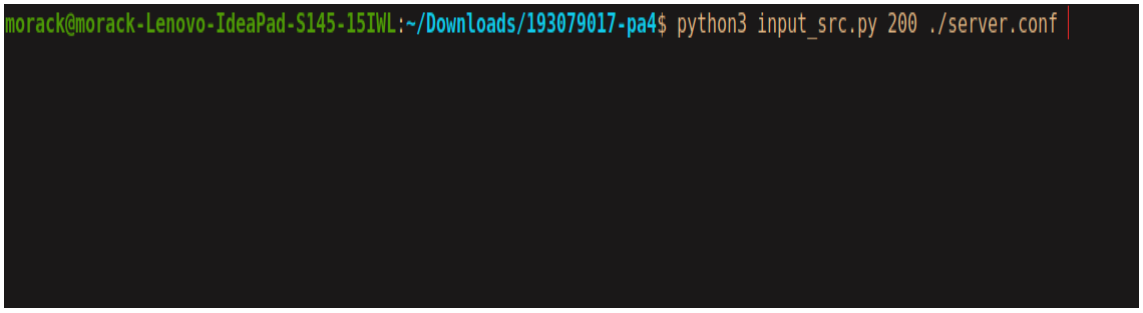

Figure 4: Python Script to run Multiple Clients in Parallel

Figure 5: Plotting the Graph

7 Design Principles

Some of the design principles used are as follows:

- The Server consists of a main thread which accepts the connection and puts it in an fd integer value for each thread. It then signals a particular thread to accept the connection. This is done in round robin fashion. Once all the threads have got at-least one connection, then the main thread puts the accepted connection's file descriptors in a queue and any one thread will accept it randomly.
- The cache implements only **LFU Algorithm** to insert new entries. The reader-writer locks are available for accessing the data from the cache.
- Two APIs, **show_cache_data(int sock_fd)** and **show_file_db(int sock_fd)** are provided to get the contents of the cache and file metadata for debugging purpose.

8 Performance analysis

The Program was first run for **150 clients** using the python script and the total number of requests made were **573,750** and the graph obtained is shown below(Figure 6).
There were 150 clients and each client was running a file containing some number of requests. First file had 100 requests, second had 150 requests and so on.
Also the graphs for 100 clients(Figure 7) and 50 clients(Figure 8) are shown below.

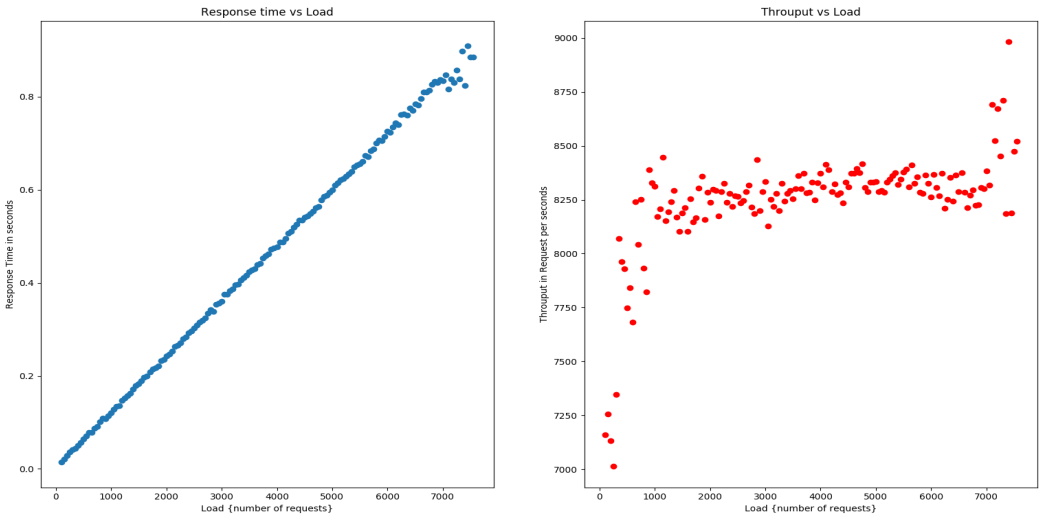

Figure 6: (Throughput vs Load) and (Response Time vs Load) For 150 Clients

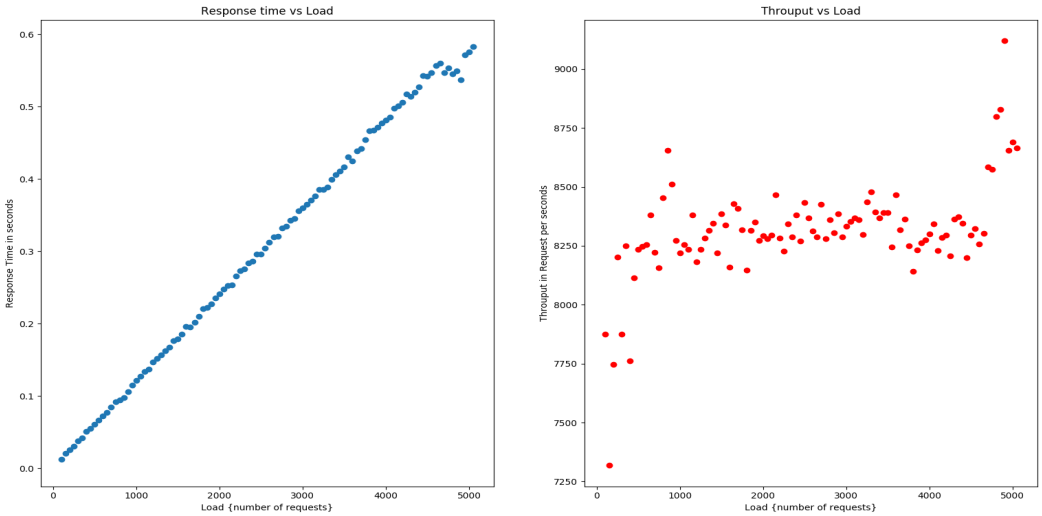

Figure 7: (Throughput vs Load) and (Response Time vs Load) For 100 Clients

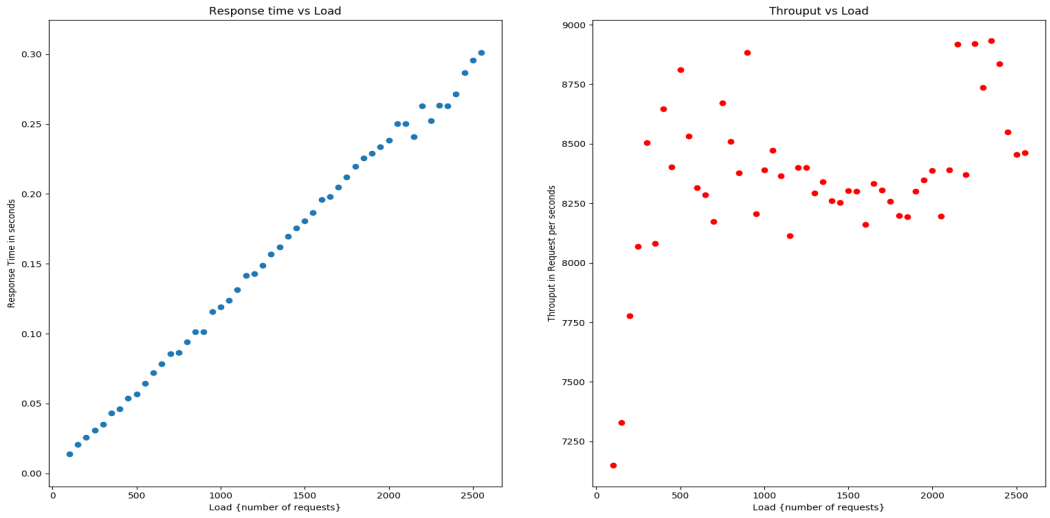

Figure 8: (Throughput vs Load) and (Response Time vs Load) For 50 Clients