CSCI-B 534 Distributed Systems

Assignment 1: Memcached Lite

Abstract:

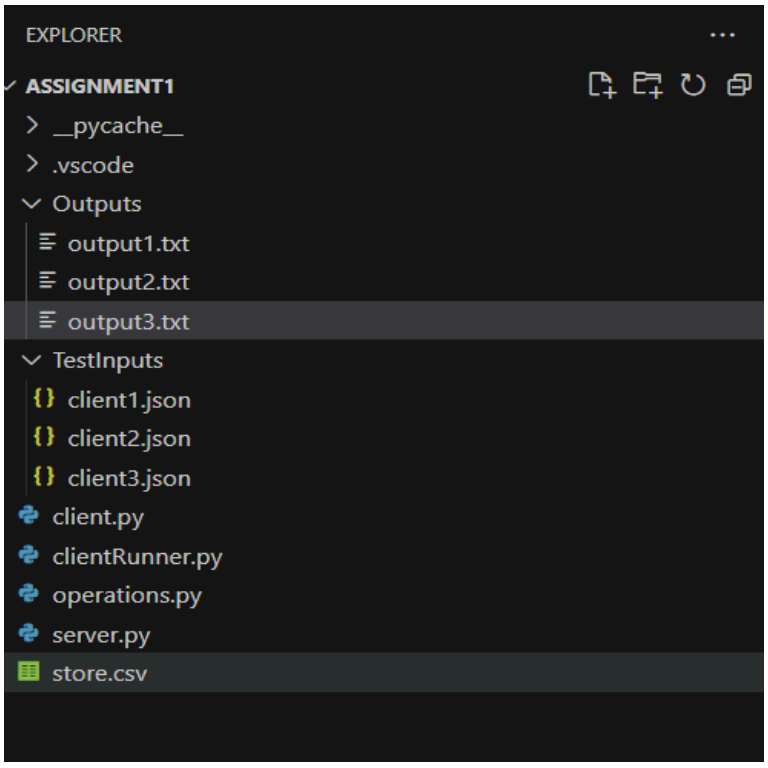
In this Assignment, we aim to build a simple key value store where a server stores the keys and the corresponding values in a file system and multiple clients can perform get and set requests concurrently, where the get request retrieves the key value pair and the set request stores a key value pair if it does not exist. We also need to ensure that our protocol is compatible with Memcached clients and that they can connect to our server and perform a series of get and set operations.

Architecture:

The system consists of one central server which receives get and set requests from multiple concurrent clients. Whenever a new client connects to the server machine, a new thread is spawned which handles the requests for that client. Each get and set request in turn, spawns a new thread which executes the request on a separate thread to handle concurrency. I have also maintained a requestQueue which stores the series of requests as they come. A thread is spawned from the main thread which handles this requestQueue and executes the requests, creating a new thread for each get or set request. There is a client runner file which takes all the input json files and spawns the number of clients which are specified as an argument while running the script. These clients then execute the series of requests specified in their corresponding input file.

File Structure:

The main server process writes the key value pairs to a file called store.csv which keeps the storage persistent, that is even after the server process has closed, the data is persisted in the file. I have created a folder called Clients which has all the client files used to send a series of get and set requests to the server. Each of the client json file generates an output file in the Outputs folder where it logs the response of the series of requests that it perfomed. There is operations.py file which actually operates on the store file and handles the get, set operations.



The clientRunner.py file runs all the client.py file takes the argument as the number of clients that we are testing on.

Design Specifications:

When the requestQueue spawns a new thread for each get and set request, I call the respective handler for that request which in turn calls the specific request method from the Operations file which then performs the required operation. To avoid multiple set requests on the same key and a set and a get request on the same key, I have used locks. I initially created a default dictionary on each key. When multiple set requests or a set and a get on the same key are received, the request Queue thread puts a lock on the first request received and blocks the other ones. When the locked request completes it's execution, it releases the lock and then the other requests in the queue can execute. In this way, I ensure that there are no concurrent set requests to the same key (Lost Update Issue) and no concurrent get set requests to the same key (Read-Modify-Write Issue). For testing purposes, I made the get method slower by introducing a sleep of 10 sec. This was a design choice and is not the ideal case. I also introduced a sleep of 3 sec in the set method. That is in general, I have made the get request slower than the set request.

I tried running upto 9 pymemcache clients concurrently, which make a series of get and set requests by connecting to our server. As I increased the number of clients, the runtime slowly started increasing. The first 3 clients form a test case together, similarly the next 3 clients form a test case together and so do the last 3 clients.

I am using the client.raw_command() method of the pymemcache Client API, because I was facing issues while receiving responses when I used the explicit get and the set methods of the Client class.

Running Instructions:

1. Run the server process in a command line by running the following command in the root directory of the project. <python ./server.py>
2. Run the ClientRunner process in another terminal process by running python ./clientRunner.py <number_of_clients> where the argument Number_of_clients is a command line argument specifying the

number of clients that we are testing on. In the present scenario it is 3

Test Case:

Client 1:

```
[
    {
        "type": "set",
        "key": "name",
        "chunkSize":"1024",
        "value": "Jenil",
        "expected": "STORED\r\n"
    },
    {"type": "get", "key": "name", "expected": "END\r\n"},
    {"type": "get", "key": "school", "expected": "END\r\n"}
]
```

Client 2:

```
[
    {
        "type": "set",
        "key": "name",
        "value": "Aditya",
        "chunkSize":"1024",
        "expected": "STORED\r\n"
    },
    {"type": "get", "key": "name", "expected": "END\r\n"},
    {"type": "get", "key": "lastName", "expected": "END\r\n"},
    {
        "type": "set",
        "key": "lastName",
        "chunkSize":"1024",
        "value": "c",
        "expected": "NOT-STORED\r\n"
    },
    {"type": "get", "key": "class", "expected": "END\r\n"}
]
```

Client 3:

```
[
    {
        "type": "set",
        "key": "school",
        "value": "IUB",
        "chunkSize":"1024",
        "expected": "STORED\r\n"
    },
    {"type": "get", "key": "lastName", "expected": "END\r\n"},
    {"type": "get", "key": "school", "expected": "END\r\n"},
    {
        "type": "set",
        "key": "class",
        "chunkSize":"1024",
        "value": "DistributedSystems",
        "expected": "NOT-STORED\r\n"
    }
]
```

The Corresponding Outputs are:

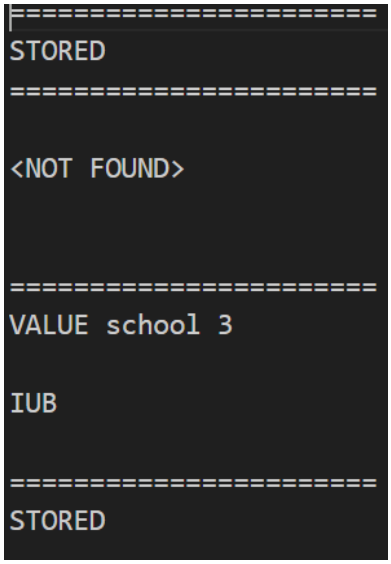
Output 1:

```
========================
STORED
========================
VALUE name 5

Jenil

========================
VALUE school 3

IUB
```

Output 2:

```
========================
NOT-STORED
========================
VALUE name 5

Jenil

========================

<NOT FOUND>


========================
STORED
========================
VALUE class 18

DistributedSystems
```

Output 3:



The clients are run sequentially from 1 to n.
As it evident from the output, we do not allow 2 concurrent set requests on the same key and the request which comes first, acquires the lock and sets the key to it's value.
Also, a get and a set request on the same key is not allowed concurrently and the request which comes first acquires the lock and performs it's execution.

Limitations:

1. As we are using a single file as a storage medium, as the number of key value pairs grow, it will take significant amount of time to read the entire file during a set command to check if the file already exists. This may lead to scalability issues.
2. If the file storage is corrupted due to some reason, there is no other way to retrieve key value pair data.
3. It is harder to store complex data structure due to serialization issues.
4. As the number of clients increases, the performance of the server may be affected and it may become slow due to memory limitations.

Potential Improvements:

1. We can make the storage distributed, by replicating the store.csv file across multiple locations or servers, so that if one file get corrupted, it's replicas would still have the storage and hence we can achieve fault tolerance.
2. The server process can also be created in a distributed manner by running the get and the set requests across different server processes, so that no one server is heavily loaded (load balancing)
3. We can use caching to store frequently accessed or recently modified key-value pairs in memory.

References:

1. Getting started! — pymemcache 3.5.2 documentation
2. socket — Low-level networking interface — Python 3.12.1 ...
3. https://docs.python.org/3/library/threading.html