Name: Aditya Arun Rudrawar

Instructions: Install the required modules from requirements.txt

Run server command: python server.py

Run client command: python client.py

Run custom client command: python client\_two.py

# 1. Description and Details

## Design

Diagram

Description automatically generated

The .env file has the host, port for TCP connection.

Implemented Memcached-lite. The server supports multi-threaded operation. One or more clients can be connected to the server where each client does a single request and closes the connection after the request is completed. The Basic server and client uses the socket module to connect to the host and listen to the specific port number. Based on the memcache protocol of set and get; we send the key with its flag and expiry and the value while set operation and the operation is defined by ‘SET’ keyword, similarly for ‘GET’ operation as well.

The server is using JSON for persistent storage, I tried with pickle file, but the performance was better in the former storage option. Plus, JSON is universally supported.

Custom client: The inhouse developed client that is not pymemcache which has SET and GET operations.

## A. Advanced: Concurrency

The server handles concurrency by using the python multithreading module, where the server waits for a connection and for each request spawns a new non-blocking thread. This way each request from each client will run on their respective threads. The above figure explains the structure of concurrency. That thread will respond to the clients after processing ends. The write requests acquire a global lock on the file and does not allow any other read/write until the lock is release. This mimics the concept of critical sections and allows us to be consistent. While reading we just need to make sure that other thread is not writing to it, so we check if the lock is locked or not.

## B. Advanced: Memcache compatibility

To make the server compatible with memcache client, I had to format the set and get request that are in line with the specification. For the set request I had to pass the expiry, flags, no\_reply parameters as well, these parameters are persisted, and the use case of these are depending on the server. For now, there is no use case. In the future we can use them for our purposes.

There are two clients: one custom client, one pymemcache client. Both have the same test cases and functionality.

## C. Limits

In the code there is not specific limit set for the key or value. But the server/custom client fails to respond when the length of the value becomes greater than or equal to 64000 characters (64KB), they probable cause might be the packet limit set by TCP connection.

# 2. Sample test cases

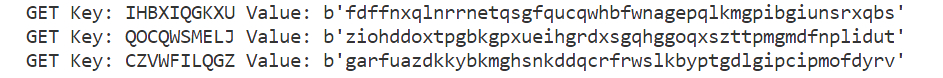
Basic test cases are handled on the custom client like Invalid Key, Key not present in KV Store, etc

SET requests with three keys:

Graphical user interface, text

Description automatically generated

GET requests with three keys:



Simultaneous GET and SET with three + three keys:

Text

Description automatically generated

# 3. Performance and Testing

The performance of the server is tested against different types of test cases. Certain constraints to length of key and value are specified:

Key: a random string of length 10

Value: a random string of length 50

Each test case is divided into three sections:

1. N number of concurrent SET requests.
2. N number of concurrent GET requests.
3. then N number of SET and N number of GET concurrent request, a total of 2\* N

where N is the number of clients making the requests.

The load to the server starts with 50 clients and goes to 400 taking a step of 50 for each test scenario.

[50, 100, 150, 200, 250, 300, 350, 400]

From the graphs below we can deduce response time is directly proportional to the number of concurrent requests. Each section of a test case is explained below:

## A. Concurrent SET requests.

The client generates random key and value pair based on the above constraints and sets either True or False to no\_reply flag randomly. The time is recorded for each request and a plot is drawn against the avg time(seconds) it takes for N clients where N goes from 50 to 400.

Chart, line chart

Description automatically generated

## B. Concurrent GET requests.

The keys generated in the previous are then used to test the GET request. Since we know the keys are stored on the server, we can rest assured that the data object is accessed atleast once. The avg time of GET requests against N clients is shown below.

Chart, line chart

Description automatically generated

## C. SET and 200 GET requests together.

For better real-world simulation I simultaneously send GET and SET request in a random order by creating a thread for each client and starting them. The performance of them is shown through the graph below.

Chart, line chart

Description automatically generated

## D. GET request for random keys

Random keys that are not present in memcache are requested, the server just responds with END\r\n

# 4. Interesting test case

I have supplied various parameters such as expiry, flags.

Also, before every read or write to file I am sleeping for a random time which are less than 1 second.

# 5. Errors (not handled)

When the key has any special characters, the memcache client raises an exception before sending the request to the server, but not all special characters are considered for example @ is passed to the server even though it is a special character.

# 6. Limitations

A. For now, the server only supports GET/SET operation with only one key. It does not support multiple keys for those operation in a single request. In addition to which it also does not support all the Retrieval/Storage/Error commands.

# 6. Future Improvements

A. We can use better file system as just one source of file will soon become bottleneck as the number of GET/SET request start increasing exponentially.

B. We can add support for the full suite of memcache commands to make it completely operable.