Simple Key value Store.

Gowtham Chowta
chgowt@iu.edu

Introduction:

Build a simple key-value store where clients from a different host machine can connect to a server and Set, Update and Retrieve key value pairs.

Problem Requirements:

- On a set request, if a key does not exist then we need to create a new key-value pair. If a key exists, we need to update the key with the new value.
- Server can connect to multiple clients and process each client in a highly consistent manner.
- Data must be stored in files in the server local machine. If the server crashes, we still have access to the existing key-value pairs.

Implementation Details:

Language: Python3 socket programming.

Server implementation:

A TCP server is setup in my local machine where the clients can connect to this server socket and make GET and SET requests.

Once the client gets connected to the server. The client can make a GET or SET request to update/retrieve the Key-value pairs stored in the server.

GET:
Request: (from Client)

```
get <key>\r\n
```

Client asks for the value of a key. If the key is present in the Server key-value store it retrieves and returns that. For faster data retrievals/updates the data is stored in dictionary. The time taken to retrieve a key is O (1).

Response: (from Server)

```
VALUE <key> <bytes> \r\n

<data block>\r\n
```

```
END\r\n
```

Server returns the message in the above format, In the first line the key and the size of the value is sent and in the next line entire value is sent.
After the message is sent, server sends 'END' to indicate the entire message has been transmitted.
Server retrieves the key from the local key-value store and returns the value in the above format. If the key does not exist it returns a message 'Not-Exists'.

SET:
Request: (From client)

```
set <key> <value-size-bytes> \r\n

<value> \r\n
```

Client requests a message in the above format, In the first line the key and the size of the value is sent and in the next sent the entire value is sent.
Server now updates in the key-value in its local key-value store. If the key already exists it updates the 'value' with the 'value' received from the client. Else, it will create a new key-value pair. All the changes will be saved to local file system in a JSON format, so if the server restarts after some crash, it can use the existing key-value pairs making a **fault-tolerant** system.

Multi-Threading server:

For maintaining high consistency, concurrency throughout the clients an improved server is implemented(server.py). *When a client request is received at the server, a new thread is created for this client and the SET/GET is executed in this thread.* So, no request need not wait for other requests and can do its work in parallel.

Improvements I wish to make:

- In the implementation the key and value limits for the Memcached server-client is 2048 bytes. Make the Memcached client work for bigger key-value sizes.
- If two threads want to update the same key. There will be a conflict and the order of operation depends on the time the message received the server. We can have a lock mechanism per key level to avoid the conflicts.
- I would also like to add an upper limit on the number of keys I can store. Like LRU Cache, we can remove the least used keys to save space and remove unused keys from a very long time.
- A better mechanism to achieve concurrency. There will always be a limit on the number of threads we can create based on the hardware in hand.

Output:

SET request:

Client Sends a set <msg> <file_size> \r\n <data_value> \r\n
If the data is stored, client receives a message STORED.

Client console:

```
❯ python3 client.py 8080 tcp
tcp
[Client]: set test 9 0
gaduValue

[Server]: STORED
```

Server console:
Server receives a message updates the key-value pairs and sends the ACK back to the client.

```
❯ python3 server.py 8080
Server is listening on port 8080
Accepting connection from  ('127.0.0.1', 52099)
Spawned new thread
[Client]: b'set test 9 0 \r\ngaduValue\r\n'
Writing data to disk
Writing data to disk Complete
[Server]: b'STORED\r\n'
```

GET request:

Client Console:
Client sends a GET request to retrieve the value of a certain key.

```
❯ python3 client.py 8080 tcp
tcp
[Client]: get gauti

[Server]: VALUE gauti 0 4
gadu
END
```

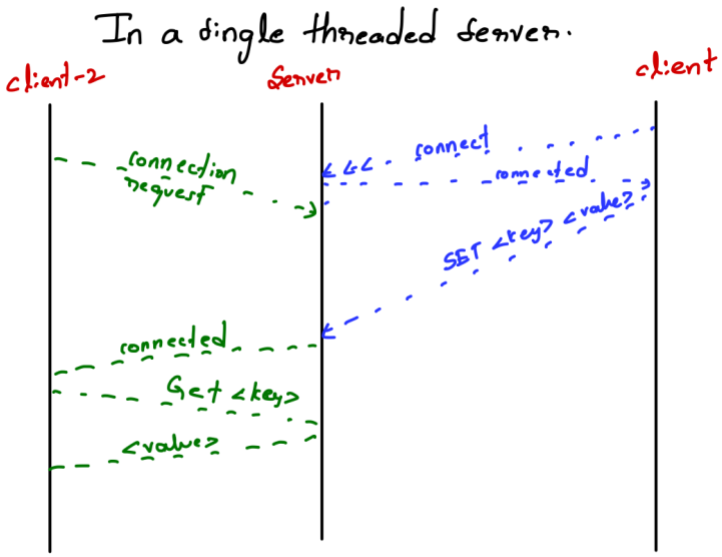
Server Console: Server gets a request and sends the value back to the client.

```
Accepting connection from  ('127.0.0.1', 52106)
Spawned new thread
[Client]: b'get gauti\r\n'
[Server]: b'VALUE gauti 0 4\r\ngadu\r\nEND\r\n'
```

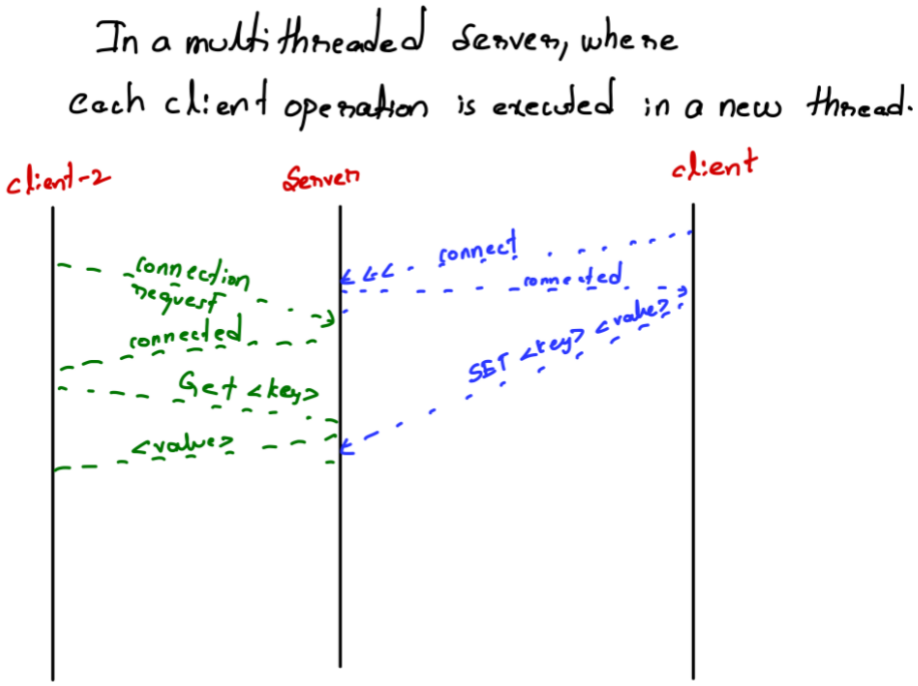
Multi-Threading

Let a client connect to a server. Client transmits a message SET request to the server and say for some reason the message got delayed and reached the server after 20 seconds.
Meanwhile another client gets connected to the server and is requesting for some key. In a single threaded server the first one has to be completed in order for the second one to start creating a lot of delay.

For e.g: As shown below, in a single threaded server only when a connection is closed a new connection is opened. So, client-2 has to wait till client-1 has to process.



In a multi-threaded server: Each operation is executed in a new thread created by the server. So, many clients can be connected to the server at once and each of them is executed separately.



Sample output for Single threaded server:

Client-1:

Connect to 10.0.0.151:8080

sleeping for 20 seconds

b'set vjcolldyqk 50 \r\n cnbhdftlpheatjmksujulozpyjmomyjrqjkveuatiazvcxylhs\r\n'

Current Time: Tue Feb **15 00:45:06 2022**

Sent message b'set vjcolldyqk 50 \r\n
cnbhdftlpheatjmksujulozpyjmomyjrqjkveuatiazvcxylhs\r\n'

Current Time: Tue Feb **15 00:45:26 2022**

ACK received from client is: STORED

Request sent at 00:45:06 and request received at 00:45:26

Client-2:

Sent message

Current Time: Tue Feb 15 00:45:08 2022

Sent request to get the key wctfpwloxe

MSG received from client is: VALUE wctfpwloxe 51
ygxajwerfvlzkwygfumnqmqzsmoewydpwctmbyzrrwievnvrnt

Value received from client is: ygxajwerfvlzkwygfumnqmqzsmoewydpwctmbyzrrwievnvrnt

Current Time: Tue Feb 15 00:45:26 2022

ACK received from client is: END

Request sent at 00:45:08 and request received at 00:45:26

Client-3:
Sent message
Current Time: Tue Feb 15 00:45:10 2022
Sent request to get the key wctfpwloxe
ClientMSG received from client is: VALUE wctfpwloxe 51
ygxajwerfvlzkwygfumnqmqzsmoewydpwctmbyzrrwievnvrnt
Value received from client is: ygxajwerfvlzkwygfumnqmqzsmoewydpwctmbyzrrwievnvrnt
Current Time: Tue Feb 15 00:45:26 2022
ACK received from client is: END

Request sent at 00:45:10 and request received at 00:45:26

Notice: Response for Client-2 and client-3 is received only after client-1 message has been processed.

Output for Multi-threaded server:

Client-1:
Connect to 127.0.0.1:8080
sleeping for 20 seconds
b'set vwyikmuoml 50 \r\n wzsipovvtbrpshlnqpfhwhpknoqxdyribhvvrtmxvziwaubgng\r\n'
Current Time: **Tue Feb 15 00:26:51 2022**
Sent message b'set vwyikmuoml 50 \r\n
wzsipovvtbrpshlnqpfhwhpknoqxdyribhvvrtmxvziwaubgng\r\n'
Current Time: **Tue Feb 15 00:27:11 2022**
ACK received from client is: STORED

MSG sent at 00:26:51 and response received at 00:27:11

Client 2:
Current Time: **Tue Feb 15 00:26:53 2022**
Sent request to get the key wctfpwloxe
MSG received from client is: VALUE wctfpwloxe 51
ygxajwerfvlzkwygfumnqmqzsmoewydpwctmbyzrrwievnvrnt
Value received from client is: ygxajwerfvlzkwygfumnqmqzsmoewydpwctmbyzrrwievnvrnt
Current Time: **Tue Feb 15 00:26:53 2022**
ACK received from client is: END

MSG sent and received at 00:26:53 almost instantaneously.

Client 3:

Current Time: Tue Feb 15 00:26:54 2022
Sent request to get the key wctfpwloxe
MSG received from client is: VALUE wctfpwloxe 51
ygxajwerfvlzkwygfumnqmqzsmoewydpwctmbyzrrwievnvrnt
Value received from client is: ygxajwerfvlzkwygfumnqmqzsmoewydpwctmbyzrrwievnvrnt
Current Time: Tue Feb 15 00:26:54 2022
ACK received from client is: END

MSG sent and received at 00:26:54 almost instantaneously.

As shown above, the multi-threaded server executes the client requests on the go on a separate thread. So, one client cannot slow down other client.

Bonus:

Support for Memcached client.

Client.py supports both TCP and Memcached as client. An argument type must be passed to notify the client what protocol to use. (memcached for MemcachedClient)

```
Fall 22/Cloud Computing/Key value store via 🐍 v3.8.8
❯ python3 client.py 8080 memcached
memcached
[Client]: Memcached set(key,value): test gaduValue
[Server]: True
[Client]: Memcached get(key): gauti
[Server]: b'gadu'
```

Server:

```
Accepting connection from  ('127.0.0.1', 52215)
Spawned new thread
[Client]: b'set test 0 0 9\r\ngaduValue\r\n'
Writing data to disk
Writing data to disk Complete
[Server]: b'STORED\r\n'
Accepting connection from  ('127.0.0.1', 52216)
Spawned new thread
[Client]: b'get gauti\r\n'
[Server]: b'VALUE gauti 0 4\r\ngadu\r\nEND\r\n'
```

Note: There is no change needed in the server side. The server works out of the box. The server gets the same message from the Memcached client/TCP client.