# **Persistent Key Value store**

# **Files and ways to run**

### **driver\_code.py**

This file has the driver code. For testing purposes this alone will suffice to check the code. Run the the file as :

python3 driver\_code.py

IMPORTANT:if the script has crashed before, it might give file loading errors ontrying to rerun. Kindly delete /values folder and map\_key\_to\_file to solve it. The script makes those files again when they are deleted.

1.The script first runs the server on a particular port. 2.Then asks how many clients do you want to test the server with. The input for this can be a command line integer thats been tested on silo to support atleast 1000 concurrent clients (put 0 to skip). 3.The script runs the number of clients given input as seperate processes. 4.Each client does a set request of a random key and random value and then gets the the value for the same key. 5.The entire process is logged to show whats happening. If the key that was set and the key that was got back from server match, client logs "successfull key value retrieval." 6.The driver code then gives an option to fire queries through command line. It gives 5 commands:

* help:lists what commands client and server support
* exit: exits the server *note: the keys will be persistent on next reboot even if you exit*
* set: you input two more strings as key and value one by one after entering 'set' . The client formats the message required by the server and sends the set request.
* get: you input one more string after entering 'get'which will be the key client will ask the server for.
* delete:This will delete the entire server cache! You will loose all data (keys and values) upon this.

### **Server.py**

IMPORTANT: The server stores a dictionary of keys and hashs in map\_key\_to\_file. If you delete this you will loose all persistent data. Individual values are stored inside the value/ folder. Deleting this folder will also lead in loosing all persistent values.

The server maintains a persistent key value store on the disk. It does this by having a dictionary of keys thats stores file names. These file names are derived by hashing the key! The seperation of values into seperate file per key was to enable faster file dumps and acesses. This way the map\_file is only updated when a new key is added (the file name stays the same if the key's value is being updated!). Parallel clients only need exclusive access to files now if they are trying to contend for the same key. This removes the bottleneck that comes when the server maintains only one dictionary which stores the key and the value.

Has all server class and methods for a server. Can be run on its own as well . It expects a port number as an environment variable called 'SERVER\_PORT'. You can set the environment first by running (modify the varianbles as you want in this file before.)

. env.sh

And then running the server by with

python3 server.py

IMPORTANT: You dont need to run this if you are running driver\_code.py! This File should be run to test a stand alone server.

* the server responds with a 'INVALID REQUEST\r\n' if message is not as below:

server accepts two types of requests on its port number for now as per specifications: (i) get \r\n (ii) set \r\n

--> Set:

Specifically, the set command is whitespace delimited, and consists of two lines:  
(i) set <key> <value-size-bytes> \r\n

(ii) <value> \r\n

* Note that this is a simpler version of the Memcached protocol, in which the set command also accepts flags and expiry time, which we will ignore for this assignment. The server should respond with either "STORED\r\n", or "NOT-STORED\r\n".

---> Get:

* Retrieving data is simpler: get \r\n

The server should respond with two lines:  
 (i) VALUE <key> <bytes> \r\n

(ii) <data block>\r\n

* After all the items have been transmitted, the server sends the string "END\r\n".
* If there is no key the server should return a bytes of 0 and an empty data block

---> DELETE:

* Deleting the whole cache: delete
* This will delete the individual value files inside value folder
* This will also delete the map\_key\_to\_file that has key to file name mappings.

You can even use the server as a class with following methods available to its instance:

server = server()

Methods:

def port\_setup(self, port\_num):

Sets up the server on the given port number. Pass 0 as port number for the OS to assign an available port on its own. When finished, the method returns a port number the server is on. (this is incase you passes 0 and want to know which port was assigned!).

def server\_loop(self):

This starts the server loop where it will listen on the port number and initiate communications with clients. For every client that connects it will start a worker thread that will server the client parallely! You just need to run this method and its will run till you provide a keyboard interrupt using cntrl+C. It calls on worker threads methods which further call methods to handle get, set, delete.

def process(self, connection\_socket, addr):

This is the worker thread that is spawned by the server process to serve the client parallely.It takes a socket the client and server have already established connection on(this is different from the server port number, that is just there for server and client to establish secondary ports). This thread will work in parallel to support concurreny.It accepts messages from the client and deals with the request it gets on port defined by connection\_socket.

def get\_key\_value(self, message\_args):

This function is called by a worker thread if the server receives a get request as mentioned that meets the formatting. The message is passed in message args as an array. This method fetches the file in which the value for the key is stored. It sends back the message as specifications say, returning an empty value if the key is not present. eg:"VALUE <value\_size> \r\n<data\_block>\r\nEND\r\n"

```bash

def dump\_to\_file(self, file\_name, data\_block):```

helper function to create a file named file\_name and with the data\_block it needs to store. Uses pickle to dump the value in non human readable format. EAch of these files is independent to their key.

def set\_key\_value(self, message\_args):

This function is called by a worker thread if the server receives a set request as mentioned that meets the formatting.The message is passed in message args as an array. This method uses the key and a helper hash function (see below for more on the hashing) to generate a hash. The hash will be the file name inside which the value for the key will be stored. It uses the dump\_to\_file() to do this. Returns "STORED\r\n", or "NOT-STORED\r\n" as response.

def read\_map\_from\_disc(self):

Loads the map thats been stored before. Automatically called on boot up of the server.

def add\_key\_to\_map(self, file\_name, key):

Adds map[key] as file\_name in the run time map. Whenever the map is updated, it is dumped to the disc using pickle. A lock is used to avoid simulataneous data dumps(otherwise the file gets corrupted).

def delete\_full\_cache(self):

deletes the entire map(even the persistent version, along with individual value files)

### **client.py**

Each server instance will be tied to a socket which it connects to with the server. To initializa an abject of client object, you need to pass the port number the server is running on.

client\_instance = client(server\_port\_number)

the client class has three methods:

def set\_key(self, key, value):

takes a key and value as strings to send formatted message as per server specifications and sends a message to the individual port the client and server are talking on(this is different from the port number passed which is the one server just accepts connections on.) Gets either "STORED\r\n", or "NOT-STORED\r\n" from the server and logs that output.

def get(self, key):

takes a key as string and formats the message as server specification above and sends the message to the individual port the client and server are talking on. Logs the output of the server.

def spawn\_random\_client(self, max\_key, max\_value)

This method is for testing purposes. The method generates a random key of random length upto max\_key and random value upto random length or max\_value. This method does a set of the random key set and does the get on the same key. It basically invokes set(key,value) and get (key). It then compares the response from the get(key) and logs as successful if the return value is same as that was set.

### **helper functions:**

hash\_fnv1a.py A simple fast hasher of noncryptoraphical type.Hashes a string into a value and returns the hash. Design choice on why this explained in problems faced. log\_helper.py A simple logger intiator for the other scripts to use. Returns a logger object.

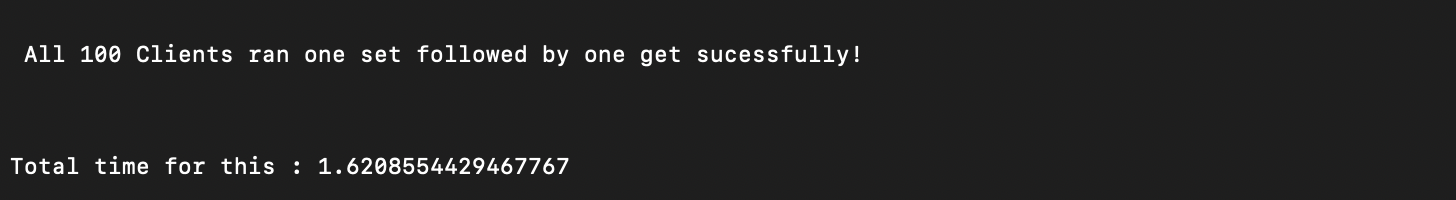
# **Testing and performance evaluation (done using burrow/silo):**

For each of the three tests, the client does a single set and a single get on the same key. It uses the spawn\_random\_client() method from client. Each of the keys is upto 30 bytes long and the values are upto 1000 bytes

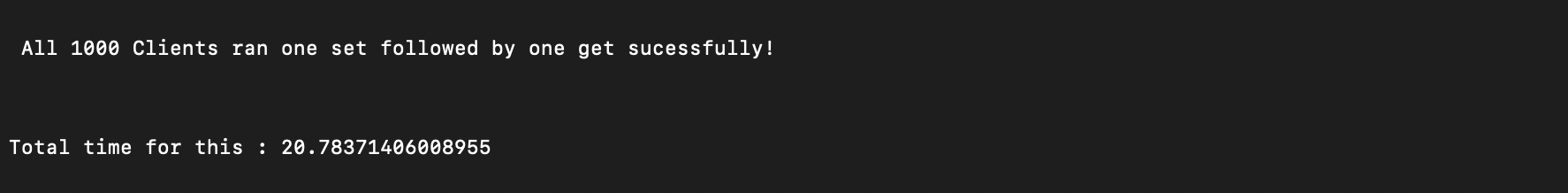
### **for 10 clients as concurrent processes:**

****

### **for 100 clients as concurrent processes:**

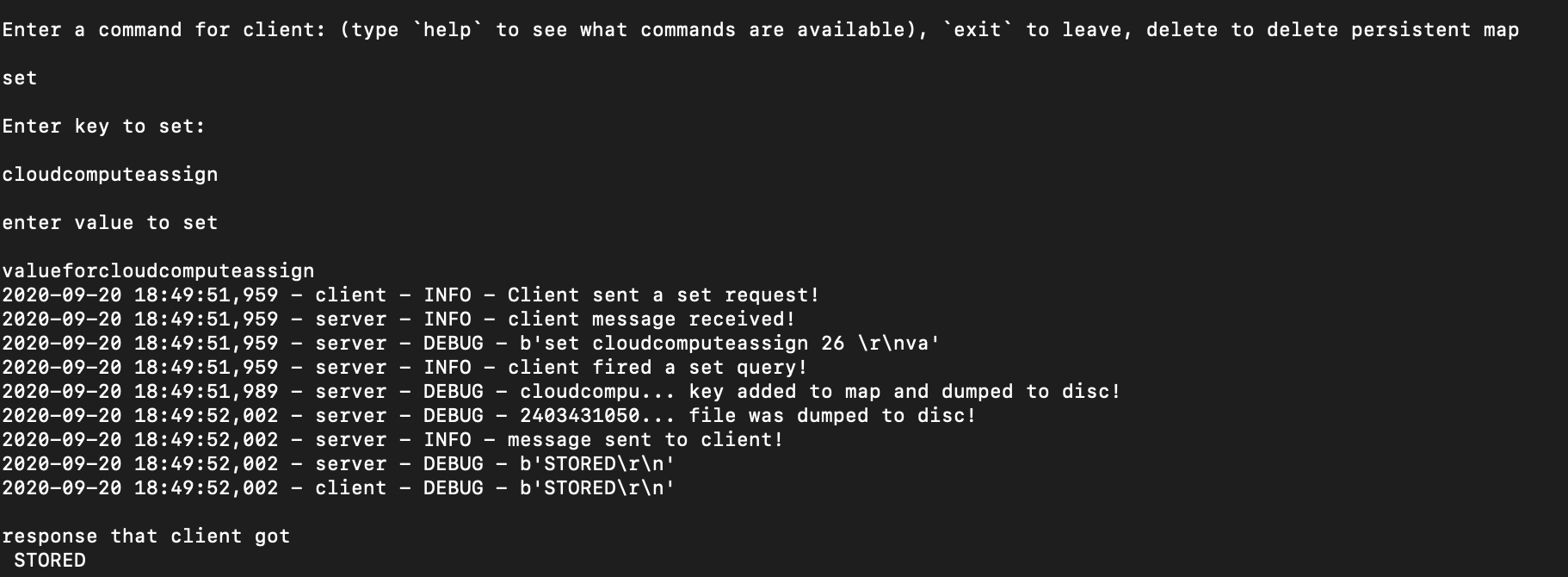
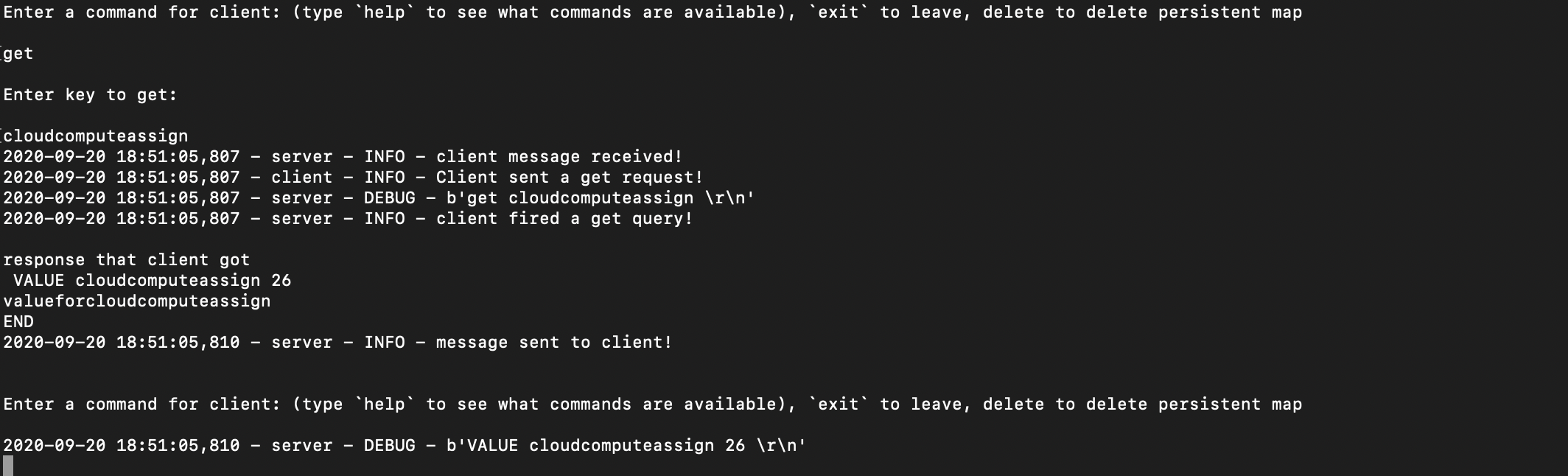
****

### **for 1000 clients as concurrent processes:**

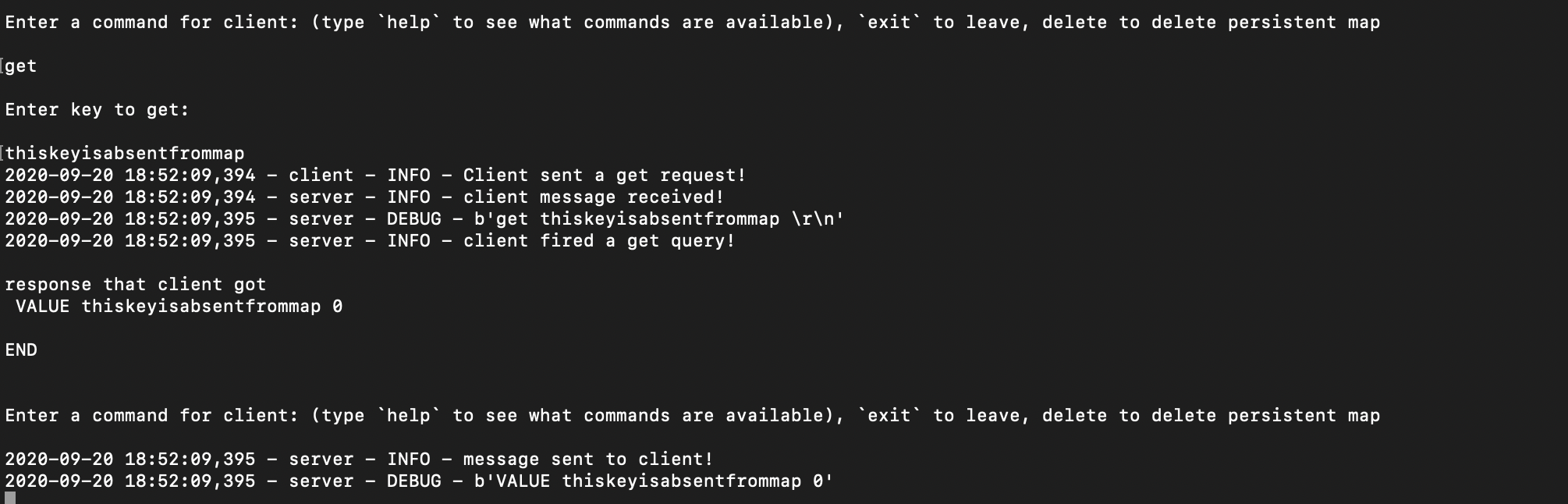
****

I noted that since i was logging extensively, there was considerable lag due to it. I suppose if i remove logging mode from debug(gives all logs) to just errors(only logs errors) the time will be much faster.

### **output for a simple set and get requests using the command line interface from driver:**

set  get 

### **Testing for a key thats not there, with driver code.**

****

### **an interesting test case:**

Initially I was using the os.open() function with O\_EXLOCK. This keeps a lock on each seperate file allowing the server to do miltiple file dumps in parallel. The operating system handles the singaling between the processes contending for same files(This will only happen if the the key is same).The issue? silo(and many old gen systems including windows) dont support this flag! I had to switch to some other mechanism. I had two choices: Either have a global lock or individual lock for each file. Global lock would remove all advantages the server got from seperating values into seperate files. I could not figure out a way to maintain a list of active files and lock them. For now the only lock I have is on the main dictionary file that stores the key to file\_name dictionary. Whats so interesting here? As I was poking around in debugger, I realized that the only time this will create errors is when two seperate clients fire get for the same key and the same time, but even that wasnt creating proper errors. This was mostly becuase the pickle dump I did was really fast and got completed before the process switched mid way. The small chance it created errors was when pickle dump was mid way and the processes switched, the value file would go corrupt in this case. But if the pickle was not interrupted, the later call wins.

# **Key Challenges:**

### **why the split between keys and their values in seperate files of a key-filename pair and value-inside-filename?**

At first look the extra hashing and the seperate files may look like extra and useless. But in an example where you have 1000 keys of 20 bytes each with 1kb values, you are dumping a 20mb file for each set request! And you can not do that concurrently since that would corrupt the file if you seitch threads in between your dump. Compare the same case to my implementation: 1000 keys of 20 bytes each with a 32 byte hash key which is less than a 1 KB! This will also be the only file you are dumping with a lock among all your workers. After you update the dictionary file, you can use the file for saving value wihtout contention since its unique for your key. This also gives boost when you are updating keys a lot. My approach skips upting the map file since the key and the hash dont change in this case. You just read what file you need to update and change that.

### **The hashing:**

The hashing at first might look like a needless complexity. But it simplifies the file name generating and it gives both the uniqueness a UUID name generator would provide but also a logical link that you can generate it any time you get the key. This would also generalize file name lengths to a good level. So, Which hashing function and why? The famous hashing functions SHA, MD5 are too computation heavy and more valuable cryptographically so they are booted out from the start. While researching hashing functions I came across <https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed/145633#145633> It is a research and comparison on between hashing functions and it puts forward FNV-1a as superior than others: <http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-param> Since the code for the hasher was easy and with no dependancoes I chose FNV-1a

### **the json vs pickledump argument.**

There seems to be a general misconception that json is faster for dictionaries than pickledump. This was infact valid when pickle dump was in its first protocol(which is still the defualt protocol). The protocol is human readable and very bloated. When used with the latest one I found this to support that Pickle dump is a lot faster. <https://stackoverflow.com/questions/2259270/pickle-or-json/39607169#39607169> In my personal testing, I saw the same results! 

### **when to close a connection in server**

The answer for this was infact very straight forward. The socket.recv() is a blocking call and when the value unblocks with a 0 value or empty string, the client has closed the connection on it's socket, so you can close it too.

# **Limits of the server and possible improvements**

* The linux and windows documentations say that at any time a process cant have more than 1024 file pointers. So if 1024 clients access do a get request at same time the process could break( not tested, it might be that the file pointers open as threads dont count)
* For now, the client and server only talk in buffer size of 4096 bytes. So the combined bit length for key and value should be just under 4kb. Future possible version can be where I check the bit Length from the set message and keep appending a set of messages till the value matches.
* When will the server be slowest? if All the clients keep firing set queries. Since there is a lock on the persistent dictionary file only one client can dump the key at a time. This is the biggest possible bottleneck for now. There can be a way to reduce this by dumping in groups of sets (every 10 set requests). Also I noted that if a thread ads a key to the map right before other thread dumps it to file the file already has this key! So I could skip another dump which the server does for now.
* Takes around 20 seconds for 1000 concurrent clients to do one set and one get request(this could be muxh faster if I was not logging heavily). The key was avg 10 bytes in length and value was avg 500 bytes in length.
* I handle cases when the files and folders required to dump data are not there. But if the file was created but curropted it creates an exception on opening. Future improved version can be a case where I deal with this.