Project Assignment

Big Data

Michael Mitsios

cs2200011

For all the exercises I used python 3.8 and it wasn’t necessary the installation of any new library.

The imports that we are going to need are:

- socket
- argparse
- json
- os
- random
- string

1. Data Creation

For the first exercise I created the main.py program.

The command to create our data is:

python3.8 main.py -k keyFile.txt -n 100 -d 2 -l 4 -m 5

The created data is going to be saved on a file named **my_output.txt.** The data creation follows the logic below:

1. For each line I have (-n 100) I create a *key_i* where *i* is the number of the line-1 starting from 0.
2. For each of these keys I picked a random number to get the maximum length and the maximum depth that they can go.
3. Then by calling the function my_directory it creates max_length features for our keys.

4. The features will be having as a value a random one, that agrees with their type which is mentioned on the keyFile.txt. If the value is a string then the random string will have a random length between 1 and max_length (-l).
5. For each of the features we flip a coin to see if we go deeper (create a nested dictionary) or not. If the coin shows that this feature is a nested dictionary then we also check if the max_depth is exceeded. If not we create the nested dictionary or else we just add a random value on our feature. The features from the nested dictionaries come from the keyFile that contains all the different types of keys that we can have (ignoring their type int, string etc.). If the coin is false then the feature just gets a random value.
6. Finally for each key and value on *my_dict* we write it on the file my_output.txt in the form that described on the question.

A simple example with (-d 1)

Another more complicated example(-d 2):


```
14 key_13 : {"name":{"name":{"height":5.585249520420716; "age":28}}; "level":{"level":4;
   "name":"lg"; "height":{"height":74.7499321897854; "level":9; "age":16; "name":"b"}}; "age":-
   {"height":{}; "age":{}; "level":{"street":"xqyi"}; "street":{}}; "street":{"street":"ed";
   "height":82.28810175017404; "age":{"street":"ff"}}; "height":2.0496041238956555}
```

2. Key Value Store

2-a KV Broker

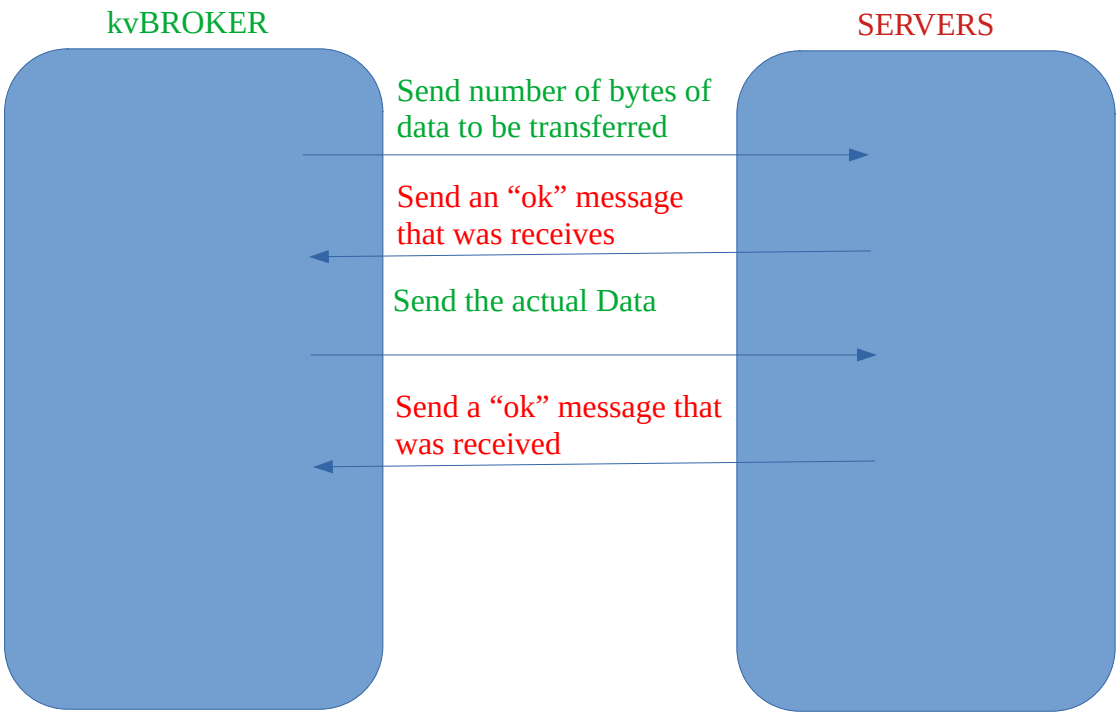
This part of the exercise is implemented in the kvBroker.py and can be executed with the following command:

python3.8 kvBroker.py -s serverFile.txt -i my_output.txt -k 2

First the broker has to find the servers and make as many connections as the servers are. For this it reads the serverFile and gets all the IPs and ports. It ends up with a list named *cons* that contains all the connections that he has established.

After that it follows the steps below:

1. Before it starts with the commands it starts to pass the data among all the servers. For each line in my_output.txt file it gets a random sample of the total connections equal with -k. By doing that it will send the key into a subset of the total connections. This subset is -k size.
2. Now that the connection subset is defined the kvBroker starts to send data through all the chosen servers. All the communication between the kvBroker and the servers is done by using this model:



In a few words each time something is sent then the other side response with an answer. This functionality was done more for the synchronization and avoid receiving many sends with one receive.

3. If all the lines were send to one server then the word “end” is send to indicate the finish of the process.
4. Completing that part, its time for a loop that will get an input command each time until “EXIT” is given. The commands are with capital and has the same syntax as the exercise describes.
5. A very important part is that before a command occur the kvBroker checks if the connections are still up by sending a test message. If it does not get a response or if the send throw an exception then we know that the server is not still up and the connection is lost. The kvBroker manage this by removing the broken connection from the list of the available connections. Then next time it will send a message to all server it will ignore the one with the missing connection.
6. Finally the kvBroker sends the command and based on the command that was typed it sends the corresponding parameters. In our case a high level key is given for commands GET and DELETE, and a set of keys (with format key_1.level.name) for command QUERY.
7. When the command EXIT happens the kvBroker close the connections and the server starts listening again.

A quick example for the commands:

GET key_0

DELETE key_0

QUERY key_0

```
##################### MENU #####################
Write one of the following commands:
GET key: get informations about a high level key
DELETE key: Delete a high level key
QUERY key: Similar to get but it can return value of a subkey
EXIT: close the program
GET key_0
######### 9090:The key->key_0 NOT FOUND
######### 9091:key_0->{"name":kfec}
######### 9092:key_0->{"name":kfec}
##################### MENU #####################
Write one of the following commands:
GET key: get informations about a high level key
DELETE key: Delete a high level key
QUERY key: Similar to get but it can return value of a subkey
EXIT: close the program
DELETE key_0
######### 9090:The key->key_0 NOT FOUND
######### 9091:key_0->DELETED
######### 9092:key_0->DELETED
##################### MENU #####################
Write one of the following commands:
GET key: get informations about a high level key
DELETE key: Delete a high level key
QUERY key: Similar to get but it can return value of a subkey
EXIT: close the program
QUERY key_0
######### 9090:The key->key_0 NOT FOUND
######### 9091:The key->key_0 NOT FOUND
######### 9092:The key->key_0 NOT FOUND
```

The port 9091 does not contain the key_0 form the start because the -k parameter is 2 and the total servers we have are 3. When we delete the key_0 from the 9090 we get a message that it didn't found the specific high-key.

Another example using the **QUERY** command:

```
##################### MENU #####################
Write one of the following commands:
GET key: get informations about a high level key
DELETE key: Delete a high level key
QUERY key: Similar to get but it can return value of a subkey
EXIT: close the program
QUERY key_4
######### 9090:The key->key_4 NOT FOUND
######### 9091:key_4->{"height":89.63993373891715; "age":7; "name":{"height":90.63532229506605; "name
":a}; "street":{"level":54; "height":40.94303676754581; "street":my}; "level":97}
######### 9092:key_4->{"height":89.63993373891715; "age":7; "name":{"height":90.63532229506605; "name
":a}; "street":{"level":54; "height":40.94303676754581; "street":my}; "level":97}
##################### MENU #####################
Write one of the following commands:
GET key: get informations about a high level key
DELETE key: Delete a high level key
QUERY key: Similar to get but it can return value of a subkey
EXIT: close the program
QUERY key_4.name.height
######### 9090:The key->key_4.name.height NOT FOUND
######### 9091:key_4.name.height->{90.63532229506605}
######### 9092:key_4.name.height->{90.63532229506605}
```

In case we delete at least one server (Ctrl+C) then the DELETE command should not be working.

```
##################### MENU #####################
Write one of the following commands:
GET key: get informations about a high level key
DELETE key: Delete a high level key
QUERY key: Similar to get but it can return value of a subkey
EXIT: close the program
DELETE key_4
######### 9090:Do NOTHING Cannot Delete if at least one server is disconnected.
######### The Number of servers that has been disconnected: 1
######### 9092:Do NOTHING Cannot Delete if at least one server is disconnected.
######### The Number of servers that has been disconnected: 1
```

And finally when the servers that are left are less than -k then the kvBroker stops.

```
##################### MENU #####################
Write one of the following commands:
GET key: get informations about a high level key
DELETE key: Delete a high level key
QUERY key: Similar to get but it can return value of a subkey
EXIT: close the program
GET key_4
9092:TOOK COMMAND EXIT
the number of remain servers(1) is lower than k(2)
user-sl@usersl-H61M-D2-B3:~/Documents/metaptyxiako/Big Data$
```

2-b KV Server

KV Server has 2 loops. One that always make him listen and another that starts when he finds a connection (becomes ready to receive commands from the kvBroker).

When a server make a connection it follows the steps below:

1. First he starts to reply to the test message that kvBroker sends to check if the server is still up or not.
2. Then he starts to wait for a command to come. The first command he is going to deal with is the “DATA”, a custom keyword that establish the beginning of the data transferring from the kvBroker to all kvServers. It receives the data and adds it to a dictionary. Each server has its own dictionary that keeps the data alive. When all the data is transferred and the server receives the word “end” then it converts the data into the TRIE structure that it has.

3. Having done that he waits for the user to type a command. When a command is typed the kvBroker send it to the server and based on the command the server make actions.
4. If the command is GET then the server receives the key he wants to find and then search it on the dictionary data structure that he built. When it finds it he return the results or else “NOT FOUND”.
5. If the server receives the command DELETE then he gets the key and deletes it from both the dictionary and trie structure. If the key deleted successfully it returns the message “DELETED” otherwise returns “NOT FOUND”. As the exercise says we can delete only high-level keys.
6. Last command, that kvServer has to manage, is QUERY. This command uses the TRIE data structure in order to find and search the given set of keys. If the value of the asked keys is a dictionary (has many key inside) then the whole dictionary is returned, otherwise the value of the key. In case where the asked combination of keys does not exists, then the server sends “NOT FOUND”.
7. If the server receives the EXIT message, then it breaks the nested loop and starts to listen again for a new connection. Also it deletes all the data that has collected (both trie and dictionary).

2-c TRIE

The Trie data structure is used for the Query command where someone can search both high-level keys and nested ones. To manage the Trie structure the KV Server has several functions to insert-delete nodes and search the structure.

First, a class was defined for the nodes of the trie structure. Each node has these characteristics:

- **char**: the character that defines the node
- **children**: The children of each node
- **finished**: A true-false flag that shows whether the Node is an and or not (simply if the Node contains a value).

- **has_nested_trie**: In case that a finished node has as value a dictionary then for this dictionary we have to define a nested trie. This variable is also a flag that show if there is a nested trie or not.
- **nested_trie**: Is the nested trie of the Node (if exists)
- **value**: Is the value of the Node (if exists). In care where the value is a dictionary then the value has inside the string format of this dictionary.

The str is a function that helped me to print the trie as I worked with it.

Having this class let's take a look on our functions.

add_to_trie: Its a function that given a root of a trie, a key and a value it will insert the key's Node to Trie and then add the value on the last Node.

delete_from_trie: This function takes the deleted key and erase its value and nested trie if exist.

search_trie: Search a trie and finds the value on the finished Node of the key.(if exists)

find_keys: Find keys is a function that searches a set of given keys going each time deeper into the nested tries if needed.

dict_to_trie: This last function given a dictionary builds the corresponding trie structure.

All the code has comments, hope to make it as clear as possible. In case of any misunderstanding please contact with me.

Thank you!