Name :- Dibyendu Mukhopadhyay
Class :- BCSE-IV Group :- A3 **RollNo.** :- 001710501077
Subject - Internet Technology Lab

Assignment No. :- **01**

1. Problem statement

Implement a TCP-based key-value store. The server implements the key-value store and clients make use of it. The server must accept clients' connections and serve their requests for 'get' and 'put' key value pairs.

All key-value pairs should be stored by the server only in memory. Keys and values are strings. The client accepts a variable no of command line arguments where the first argument is the server hostname followed by port no. It should be followed by any sequence of "get <key>" and/or "put <key> <value>". ./client 192.168.124.5 5555 put city Kolkata put country India get country get city get Institute

India

Kolkata

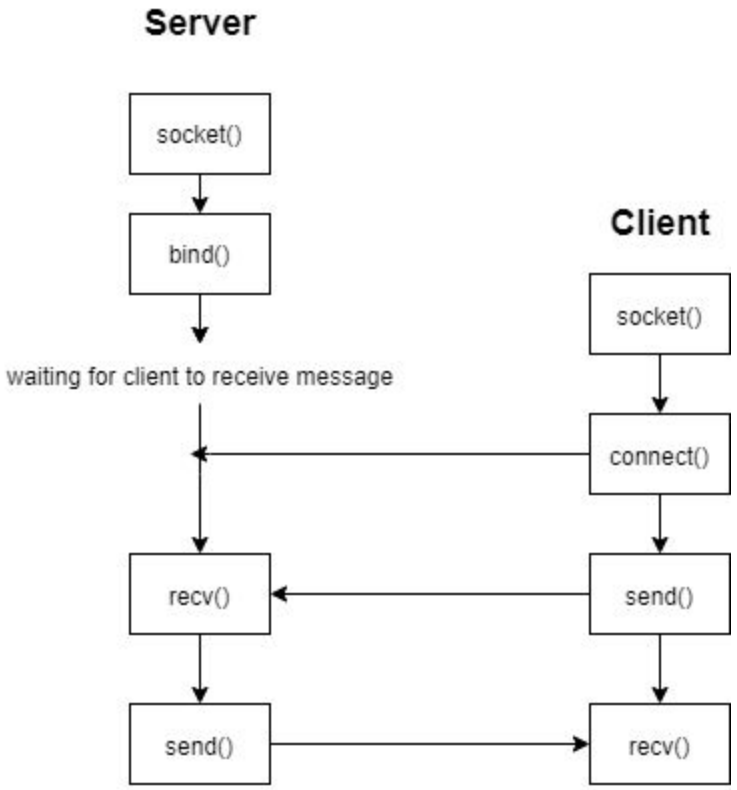
<blank>

The server should be running on a TCP port. The server should support multiple clients and maintain their key-value stores separately.

Implement authorization so that only few clients having the role "manager" can access other's key-value stores. A user is assigned the "guest" role by default. The server can upgrade a "guest" user to a "manager" user.

2. Solution Approach

The code is implemented in Python language in socket programming.



Model view of Data Structure

Server Side :-

A class list of data which contains username, key and value which performs initialization, adding the data and searching the data.

```
#list of the user details
class UserDetails:
    def __init__(self):
        self.data = {}
    #adding the data function
    def add_data(self, key,data):
        self.data[key]=data
    #searching data function
    def search_data(self, key):
        if self.data.get(key,-1)==-1:
            return str("404 :- Data not Found for this key")
```

```
        else:
            return self.data.get(key,-1)
```

This is the fundamental part to create a socket which helps to interact between server and client(s). In order to interact with the server, the client side must have the same address and port as the server. Notice, it is recommended to keep the port number above 1024 because the port number less than 1024 are reserved for standard internet protocol.

```
#Creates socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

#Binds the socket to the IP and port
server_address = ('', 5000) #127.0.0.1 5000
print('Starting up on host {} port {}'.format(*server_address))
sock.bind(server_address)
```

The server then waits for the client to receive the message whatever the client says. The server will then send the message in the form of pickle.

```
while True:
   packet, address = sock.recvfrom(4096) #waiting for receive the msg
   packet = pickle.loads(packet)
```

Now, the main role is given below

- **Connection** :- Once the client connects the server, then the server will consider the client as guest by default and will send the message to client so that client can confirm whether it is connected or not.

```
#connect to the client
    if packet["type"] == "connect":
        if users.get(packet["user_name"],-1)==-1:
            uname=packet["user_name"]
            users[uname]=UserDetails()
            users[uname].add_data("authority" , "guest") #by default

            #source,destination
            sock.sendto(pickle.dumps({"response":"connected"}), address)
#sends to the client
            time.sleep(11.6)
            sys.stdout.flush()
```

```
            print("User added {}".format(uname),"authority
mode(default):Guest",sep="<~~>")
            print("\n")
        else:
            sock.sendto(pickle.dumps({"response":"failed to connect"}),
address)
            print("\n")
```

- **put <key><value>** :- The client has put the key and value and the server will receive the message and will put in **add_data()** function. The server will send the message to the client that the data has been successfully added.

```
#put key :- Adding data
    elif packet["type"] == "put":
        users[packet["user_name"]].add_data(packet["key"],packet["data"])
        sock.sendto(pickle.dumps({"response":"data added successfully"}),
address)
```

- **get <key>** :- When the client wants to get the data by writing a “key” and the server will then search the data which was previously stored. Finally, the server will send the appropriate message to the client.

```
#get key :- search data
    elif packet["type"] == "get":

sock.sendto(pickle.dumps({"response":users[packet["user_name"]].search_dat
a(packet["key"])}), address)
```

- **Manager mode** :- The client requests the server to upgrade it to manager level. The server can either approve the authority or reject the same. As a result, the client will receive the message whether the server has considered the authority or not.

```
#Request server to approve manager
    elif packet["type"] == "manager_mode":
        print("{} wants to be the manager :".format(uname))
        accept = str(input("Accept/Reject : "))
        if accept=="Accept" or accept=="accept":
            users[packet["user_name"]].add_data("authority","manager")
```

```
            sock.sendto(pickle.dumps({"response":"authority update
successful"}), address)
            print("User approved {}'s authority mode:
Manager".format(uname))
        else:
            sock.sendto(pickle.dumps({"response":"discarded"}), address)
            print("User rejected {}'s authority: Manager".format(uname))
```

- **get_other<key><username>** :- Only manager can perform that in order to get the data from the other client.

```
#Only manager can do that...
    elif packet["type"] == "get_other":
        if users[packet["user_name"]].search_data("authority") ==
"manager":

sock.sendto(pickle.dumps({"response":users[packet["user_name2"]].search_da
ta(packet["key"])}), address)
        else:
            sock.sendto(pickle.dumps({"response":"not authorized"}),
address)
```

- **Close** :- When the client wants to close the server. The server then removes everything that particular client's details and removes it from the server. Once done, the client will receive the appropriate message that it has been disconnected. As a result, the client cannot perform any task since it has been disconnected.In other words, the client side will be terminated.

```
#close server
    elif packet["type"] == "close":
        if users.get(packet["user_name"],-1)!=-1:
            sock.sendto(pickle.dumps({"response":"disconnected"}),
address)
            print("User removed {}".format(uname))
            del users[packet["user_name"]]
```

Client Side :-

The client command must have three arguments as given in the first line of the comment section. The client will have to write the user name. If the socket is connected, then it will receive that it has been connected which is approved by the server since having the same hostname and port number.

```
if len(sys.argv) == 3: #python client.py 127.0.0.1 5000
    portname = int(sys.argv[2]) #5000
    hostname = str(sys.argv[1]) #127.0.0.1

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_address = (hostname, portname)
    uname = str(input("Username : "))
    try:
        while True:

sock.sendto(pickle.dumps({"type":"connect","user_name":uname}),
server_address)
            response, sender_address = sock.recvfrom(4096)
            if pickle.loads(response)["response"] == "connected":
                print("Connecting to server network",end='')
                for i in range(5):
                    time.sleep(2.4)
                    print(".",end='')
                    sys.stdout.flush()
                print("Now Connected!")
print("-------------------------------------------------------\n")
```

The client will perform their role as get the key value, put the key and value, upgrade to manager and get other client's data. Everything will be sent to the server and the server will do their own task. It is more like communicating between client and server.

```
while True:
                    commandline_args = str(input("cmd:>> "))
                    command_array = cmdfun(uname, commandline_args.split('
'))
                    for command in command_array:
                        if command["type"] == "error":
```

```
                            print("Invalid Input Command ... ")
                            print("\n")
                            break
                        elif command["type"] == "close":
                            sock.sendto(pickle.dumps({"type":"close" ,
"user_name":uname}), server_address)

                            response, sender_address = sock.recvfrom(4096)
                            if pickle.loads(response)["response"] ==
"disconnected":
                                print("{} quits!".format(uname))
                                print("Disconnecting from channel
network...disconnected")
                                sys.exit(0)
                                break
                        else:
                            sock.sendto(pickle.dumps(command),
server_address)
                            print("Requesting to server for
{}".format(command))
                            response, server = sock.recvfrom(4096)
                            response = pickle.loads(response)
                            print("{}".format(response))
                            print("\n")
```

This is the **else** part of the code. If the user name already exists or if, in case, there is an infinite loop, then press *control+c* to stop the loop.

```
else:
                print("Invalid username !\n")
                print("Or, Username already in-use")
                break
    except KeyboardInterrupt:
        print("KeyboardInterrupt has been taken...")
    finally:
        print('Closing socket')
        sock.close()
```

```
else:
    print('Two arguments are required...')
```

Function part :- In order to make something like this, “put city Kolkata put country India get country get city get Institute” in client side command in a single line. This **cmdfun()** function is here to solve this where the command list of array will be appended and then sent to the server to do their task.

```
def cmdfun(uname, args):
    i=0
    command_array=[]
    while i<len(args):
        command={}
        if args[i]=="get":   #get <key>
            command["user_name"]=uname
            command["type"]="get"
            command["key"]=args[i+1]
            i+=2
            command_array.append(command)
        elif args[i]=="put": #put <key><value>
            command["user_name"]=uname
            command["type"]="put"
            command["key"]=args[i+1]
            command["data"]=args[i+2]
            i+=3
            command_array.append(command)
        elif args[i]=="get_other": #get_other <username><key>
            command["user_name"]=uname
            command["type"]="get_other"
            command["key"]=args[i+1]
            command["user_name2"]=args[i+2]
            i+=3
            command_array.append(command)
        elif args[i]=="close": # close
            command["user_name"]=uname
            command["type"]="close"
            i+=1
            command_array.append(command)
```

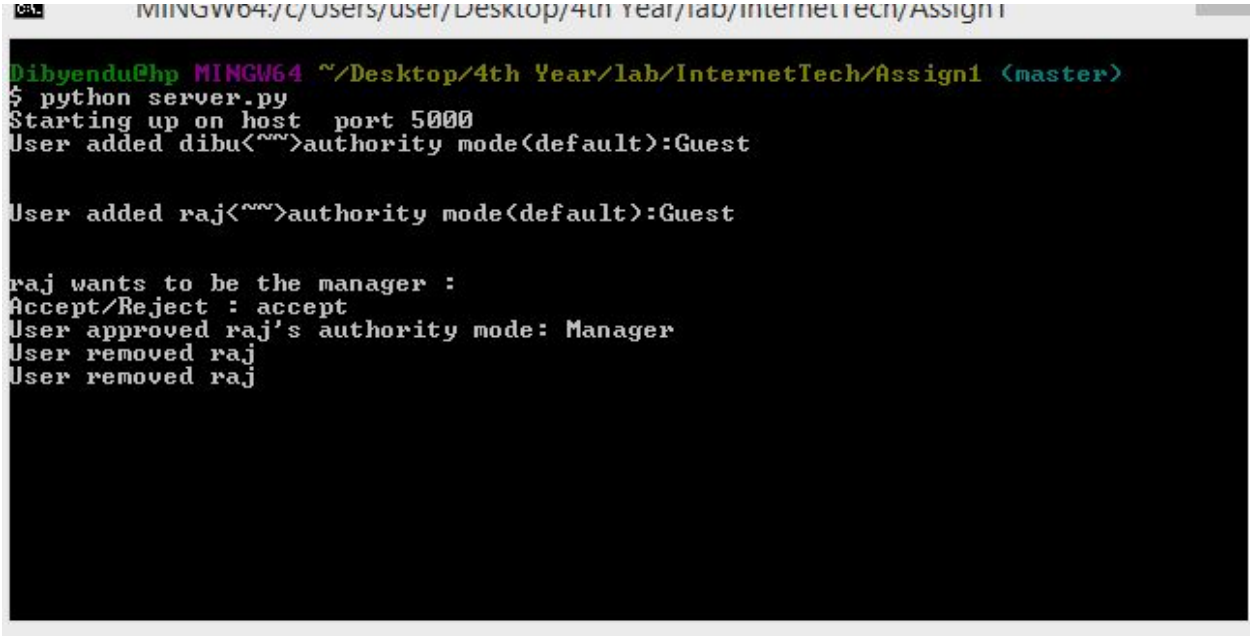
```
        elif args[i]=="manager_mode": #manager_mode
            command["user_name"]=uname
            command["type"]="manager_mode"
            i+=1
            command_array.append(command)
        else:
            command["type"]="error"
            i+=1
            command_array.append(command)
    return command_array
```

3. Salient features

- Server code executes at first and waits for client message/request.
- Client code starts sending the message first to the server.
- Server response in accordance to the client's request.
- **Non-Stop Server :-** Server program can keep running indefinitely. Once, the server stops the client will be terminated at once.
- **Non-interference :-** Client programs can disconnect itself from the server without interference from other clients communicating.
- **Multi-client Support :-** Widely supports the multi-client as long as their host number and port number are same as the server.
- **Manager Support :-** The manager level is supported when the server considers the authority and then the manager can search the other client's details. It can be done ***get_other <key><username>*** where the manager has to write the client's name and ask for what key he/she wants.
- **Data Structure :-** For every client, their details such as ***<key><value>*** are stored in the form of hashing and then stored in the arraylist. Whenever the user requests the *key*, the server then linearly searches for the key from the arraylist.

4. Sample Input/Output :-

Server Side :-



Client Side :-

a) Guest Mode (Default)

b) Manager mode