COMP28112 Exercise 2 Report

Account: a65914zc
Date: 19th March 2023

Task 2.4

Protocol Description

To implement the traditional server-client system of centralized architecture, most functionalities would be satisfied in the server, making simple communications with clients.

In terms of principles of this system, **firstly**, for the naming scheme, the style of **Centralized Naming Approach** was followed. *For one thing*, the process of this system worked in different machines was simulated by several terminals locally with specific command parameters. *For another thing*, the name of users could be uniquely identified through the client (socket) object. Since in this coursework, the client object was allocated by the machine memory satisfying the uniqueness, the name entered in the input () function would be sent and compared with the dictionary or list in the server, to check whether this name was occupied. **However**, “the single point of failure” should also be noticed especially in the implement of python files. The whole system may return the error because of a small bug in the client or server, affecting the stability of this system. To solve this, “the single point of contact” was considered and some “try-except” pairs and if-else statements should be used in the code to catch exceptions during the system running, replying with suggestions for the inappropriate format of client inputs.

Secondly, for the main communication protocol, three commands were designed to satisfy the instruction requirement as the "technology specifications", following the style of the HTTP (Hypertext Transport Protocol).
The command list: **OPTION**, **GET**, **POST** and **DELETE**
For the **OPTION**, it would return operation numbers to users in the client, making it better for the selection. When the client started or began a new loop whatever the previous operation was (except: exit), the fixed words would be printed in the While loop in the client terminal. And simple checking system would be set and return the hint if a wrong operation number(string) was given.

For the **GET**, it would return current screen (client) names connected to the server. When the user input the correct operation number in the client terminal, which was also sent and received in the server, this server would return a dictionary or list preserving name information back to the client, then printed in the client terminal to the user.

For the **POST**, it would send the message for the public or private chat from one client to other clients (or just one private client). When the user input relational operation numbers for the public or private chat, the client terminal would encapsulate and create a piece of message under some rules, sent to the server message listening station. The communication content or private client name (if possible) would be interpreted out and sent to the target client through a dictionary or list preserving the client object and name data. **If** the operation was public one, this message would be shared with all registered users connected to the server. To be more specific, the message would be sent to all used socket (client) objects, saved in values of the dictionary of the server. **If** the operation was private one, this message would only be sent to the client of that private name. The server might check whether keys of the dictionary had this name: if true, the socket (client) object would be got through the input name for private chat and the

message was sent to it, if not, the hint would be returned in the client.

For the **DELETE**, it would disconnect the current client from the server and delete the information stored on the server such as name and socket(client) object. Therefore, when people disconnected the terminal, the resource of names would not be wasted and could be registered again.

Pseudo-Code

According to the coursework requirement, pseudo-code of the **server** was provided, showing basic principles of the protocol.

```
Server() {
    onStart()
        dict names = {}  // dictionary   [name]=socket
        int active_client_number <- 0

    onConnect(socket)
        active_client_number++
        socket.tag <- "default"
        socket.name <- "Anonymous"

    onDisconnect(socket)
        active_client_number--
        IF socket.name of the client terminal is not "Anonymous"
            DELETE a key-value in the dictionary (key <- socket.name)

    onMessage(socket, message)
        (name, operation, content, private) <- message.split("symbols")

        IF name not in names AND socket.tag == "default"  // CASE 1
            ADD "name:socket" to names  // [name]=socket in dictionary
            socket.name <- name
            GENERATE a piece of random strings  // random strings or numbers
            socket.tag <- random strings + socket.name // unique identifier
            // operation numbers
            IF operation == 1:
                print keys of names // keys of the dictionary
            ELIF operation == 2:
                for socket in names  // each value of the dictionary
                    send the content to every socket
            ELIF operation == 3:
                target_user <- get socket object from searching private in keys of
names // keys of the dict
                IF target_user not in values of names // target_user -> socket
                    print(This user has not been registered)
                ELSE
                    send the content to the target_user
            ELIF operation == 4:
                send message to the client, let it disconnect to the server
```

```
            ELSE
                print(Parsing Failure)

        ELIF name in names AND socket.tag == "default"  // CASE 2
            print(Note: This name has been registered)

        ELIF name in names AND socket.tag == random strings + socket.name // CASE 3
            // operation numbers
            IF operation == 1:
                print keys of names // keys of the dictionary
            ELIF operation == 2:
                for socket in names  // each value of the dictionary
                    send the content to every socket
            ELIF operation == 3:
                target_user <- get socket object from searching private in keys of
names // keys of the dict
                IF target_user not in values of names // target_user -> socket
                    print(This user has not been registered)
                ELSE
                    send the content to the target_user
            ELIF operation == 4:
                send message to the client, let it disconnect to the server
            ELSE
                print(Parsing Failure)

        // it could pass at initial time, put it at the bottom of the IF-ELIF
statements
        ELIF socket.tag != random strings + socket.name  // CASE 4
            print(This is not your name)
            print(You should input the same name as before)

        ELSE  // CASE 5
            print(Parsing Failure)
}
Server()  // run
```

Pseudo-Code Explanation

(**Suppose** the server code was designed to compromise the telnet terminal not myclient.py and the special situation for middle of "raw input" should be considered, therefore, the content would be entered once)

In terms of the code logic, when myserver.py was complied, **onStart()** would run firstly, setting a global server-level dictionary and a global integer to count how many client terminals were connected to the server and preserve their information. **onConnect()** would add the integer when a new client terminal was connected and two socket-level strings were assigned related to the protocol, which were created at the connection moment but freed automatically at the disconnection moment. Besides, **onDisconnect()** would do the subtraction and delete name information in the dictionary with the socket (if the name had been registered).

onMessage() would maintain the main protocol mechanism, when message sent from client terminals. Firstly, all the information (name, operation number, messsage, private chat target user) could be interpreted through the split () function. Then, the name registration system was started and there would be **five** CASEs as shown in the code.

The **first** CASE would check whether the input name was in the **server-level dictionary** and the **socket-level string** socket.tag was the default one. If they both satisfied, this name could be registered, since the socket-level socket.name could only be changed after the name registration in one client terminal, otherwise it was just "default". Then, **add** this 'name'-'socket object' combination to the dictionary, **update** the default socket.name as the input name and **change** the default socket.tag as the unique identifier of this client terminal through the unite of input name and random strings. Then, the server would check the **operation number**, if out of the range, the hint message would be printed. **If the number was 1**, the keys of the dictionary, which were names of the client, would be returned and printed on the client. **If the number was 2**, the FOR loop was used to get every socket object in the dictionary and sent the public content to all client terminals one by one. **If the number was 3**, the server would firstly check whether the target user for private chat was in the dictionary: if not, return the hint message; if the target was registered, the socket object could be got through the input target name, and message would only be sent to that people. **If the number was 4**, the server would make this client disconnected to it and return the relational message.

The **THIRD** CASE maintained the similar logic, this would happen if user used the terminal second time or later. The server would check the name situation, if the name was already registered, but the socket.tag was equal to the input name with the specific random strings, which meant the input name was just the registered one in this terminal, the operation number should also be checked sequentially.

The **second** CASE would happen if the input name had been registered in the dictionary and the socket.tag was still the default one, which meant this name was the first input in the client terminal but registered, therefore, provide the hint to the client.

The **fourth** CASE would happen if the socket.tag cannot equal the unique identifier, which meant the name was not same as the former input. However, when the people firstly registered the name successfully, this may still pass, so it would be set at the bottom of the CASE check system.

The **fifth** CASE would happen if other cases were skipped, though the situation was fully considered. To avoid the single point of failure, the real code may add the **try-except** to catch any error to strengthen the stability of this chat system.