Cole Tistan
Data Communications and Networking
Semester Report
Dr. Wang

Introduction

For this semester project, you realistically can use any language to implement a UDP chat room like C language, Python, Java, PHP, and many more; now, for what language I decided to use, I ended up using Python to implement the TCP chat room project. This is because Python can be used to write scripts with complex configurations quickly; also, Python is very english-friendly for people new to coding and those that want to learn it for professional use for not just socket programming, but for other purposes too. For what Integrated Data Environment (IDE) I decided to use, I used Visual Studio Code to build my chatroom because Visual Studio Code comes packaged with many extensions and packages that can be installed to make development using any language, including Python, easier. Thus, making the development cycle move smoothly. We will discuss what the major differences between UDP and TCP sockets are, how they work, and what scenarios you will use one of these protocols.

According to this source, guru99.com, it describes a UDP socket as a "datagram oriented protocol," which is "a transfer unit associated with a packet-switched network" and UDP "works almost similar to TCP, but it throws all the error-checking stuff out, all the back-and-forth communication and deliverability." For the major differences between UDP and TCP, UDP is considered relatively faster than a TCP socket and doesn't use any connection-oriented protocols; thus, it is a connection less protocol. In addition, handshake protocols like ACK, SYN, and many more are not used by UDP whereas TCP utilizes handshake protocols while running.

For use cases, TCP and UDP can be used with video games with multiplayer where the servers for a match can have ten players total or a game where two people are versus each other. For instance, a game like Counter Strike: Global Offensive (CSGO) when playing competitive matches or any of the Call of Duty games would use UDP protocols in their servers because they know ten or more clients will be playing in one server at a time, so they are concerned about the speed of each game match running without any packet loss. In addition, UDP ensures that the game doesn't have to wait for every players' packet. For TCP, this protocol would be used for game matches involving two to four players only or games played locally with games like Super Smash Bros or Mortal Kombat. These games use TCP since they run each frame at a time to run properly without any packet loss.

During development, it was significant to be sure I understood which use cases are appropriate for either TCP or UDP. With that in mind, the project that I am working on is a simple chat room using TCP instead of UDP. I went with TCP because I am only going to have two to three clients talking to each other instead of more than four clients. Some of the biggest challenges I will encounter is how to allow the client to leave the chat using commands like '!quit' or '!join' to allow them to do so. I decided to not worry about this because the client can always leave and join by closing their session and entering their username respectively.

Design and Implementation

At the beginning of developing the TCP chatroom, I knew two different files were to be created, which are the server and client files. I originally was going to try an object-oriented design approach with this project where I have two classes, Client and Server with the appropriate methods they would use, then run create objects to run in the main method. However, I realized that the project would become very bulky and disorganized so I decided to create two separate files that contain the functions needed to run both files simultaneously in their appropriate scopes.

As I have said before, I was going to implement the TCP chat room using Object-Oriented Design Principles, but instead went with a procedural design approach. With this design approach, I will describe the pseudocode for the program made. To start, I placed all of my libraries and global variables, like the port number and IP address on the first twenty lines with the latter two global variables being placed inside a tuple; the libraries I decided to import were socket and threading, which I will go into detail later. These were all needed in both files during the development cycle because they will be used to help us connect to a server. For instance, the socket library contains constants like AF_INET, SOCK_STREAM, and SOCK_DGRAM that can be used to create sockets with the protocols needed. AF_INET refers to the ipv4 family address for connection to the internet, SOCK_STREAM is a TCP protocol and SOCK_DGRAM is a UDP protocol for connectionless use. The socket will then bind and listen for any incoming connections from multiple clients.

Now that we know what common assets the chatroom utilizes, both the server and client files have different tasks that rely on one another to function properly. Starting with the server file, every new client that arrives into the server will be stored in an array to keep track of which clients and users are still active in the server, which will be managed by a function called clientHandler(), which uses a *try-except block* to present all messages to the chatroom like indicating who has left when they close their chat window and who joined the chatroom. When a user would close out, the server would indicate who left and remove that client as well as their username associated with that client. For clients joining, the receive function will append a new client to the client array in addition to their name in the username array, which will then start a new thread for this client; the server will then print format the client's username that joined the chat. Originally, I wanted to add server commands to allow client(s) to be able to join a chatroom or quit by typing '!quit' to leave or '!join' to join the chat on their keyboard; Unfortunately, I was not able to implement a quit command because the clientHandler function is not able to recognize it as a command to close a socket for a particular client. I initially added a conditional statement that says `if message == '!quit', then close the client socket and remove their client information from the arrays.` For joining the chat, the way a user can join is by entering their username and that will allow the server to indicate that they have joined successfully.

For the client file, the user will be given an input prompt to enter their username that the server will store in two arrays to know which clients are active. It uses two functions, which are receive() and write() respectively. The receive function is similar to the server's receive function except that it makes sure the client has entered a username in order for it to be sent to the server it can be stored in the username array; if they do not, they would close out of their thread. For the write function, it is used to specify which username sent a message through the chatroom. For instance, when a client user named

Brett enters their username and types a message like ‘Hi’ in the chat, everybody active in the server would be able to see that Brett said Hi like this `Brett: Hi.`

Testing

When it comes to testing, I tested this program by creating a server threads and three clients to see what occurs in the program. I entered a username for each client thread created to be added to the server arrays to be stored until the client leaves. The server then read this:

```
Server listening...
connected with ('127.0.0.1', 59001)
username of the client is Turtle
connected with ('127.0.0.1', 59002)
username of the client is Pistachio
connected with ('127.0.0.1', 59008)
username of the client is Gammy
```

As you can see, the output in the server thread is what we exactly wanted, which is an indication of who joined the server and what port they are using to connect. Next, I tested what messages are being sent by each user. Here is the output in the client terminals:

```
Choose a username: Turtle
Turtle joined the chat!
Connected to the server!
Pistachio joined the chat!
Gammy joined the chat!
Hello
Turtle: Hello
Pistachio: Hi Turtle
Gammy: My name is Gammy.
Run along Gammy.
Turtle: Run along Gammy.
Gammy: No Turtle.
Pistachio: Come on Turtle.
```

```
n tcpClient.py
Choose a username: Pistachio
Pistachio joined the chat!
Connected to the server!
Gammy joined the chat!
Turtle: Hello
Hi Turtle
Pistachio: Hi Turtle
Gammy: My name is Gammy.
Turtle: Run along Gammy.
Gammy: No Turtle.
Come on Turtle.
Pistachio: Come on Turtle.
```

```
on tcpClient.py
Choose a username: Gammy
Gammy joined the chat!
Connected to the server!
Turtle: Hello
Pistachio: Hi Turtle
My name is Gammy.
Gammy: My name is Gammy.
Turtle: Run along Gammy.
No Turtle.
Gammy: No Turtle.
Pistachio: Come on Turtle.
```

This output shows which users have joined the server and which message corresponds to each user. Now, let's say a user wants to quit the chatroom, they would need to exit out of their terminal and this would broadcast to all users that this particular user has left the chat. Here is the output of when a user leaves a chat:

```
Pistachio joined the chat!
Connected to the server!
Gammy joined the chat!
Turtle: Hello
Hi Turtle
Pistachio: Hi Turtle
Gammy: My name is Gammy.
Turtle: Run along Gammy.
Gammy: No Turtle.
Come on Turtle.
Pistachio: Come on Turtle.
Turtle has left the chat
```

```
Gammy joined the chat!
Connected to the server!
Turtle: Hello
Pistachio: Hi Turtle
My name is Gammy.
Gammy: My name is Gammy.
Turtle: Run along Gammy.
No Turtle.
Gammy: No Turtle.
Pistachio: Come on Turtle.
Turtle has left the chat
```

From this output, we see that the user, Turtle has left the chat and this will allow the server to remove that user and client thread from the arrays in the server.

Conclusion

To conclude, I think I have fulfilled most of the requirements needed to complete this project; specifically, I believe I have met the following requirements: the server sends a welcome message to a newly joined user, the server indicates which user(s) have left the chat, the existing messages are broadcasted to all existing users in the chatroom, a unique port number is indicated to the server when a new client has joined, and multi-threading is utilized to wait for a user's input; thus, then wait for the retrieval of messages to print. The only requirements I believe I have not met were allowing users to type in join and exit commands to allow the user to have the ability to leave or join gracefully. For what I have learned, I learned how to write sockets and how to utilize them with multi-threading in Python. Overall, this was a very challenging project, but it has helped me learn new technologies and languages to give me a head start in when I write future programs that use Python.

Sources

https://www.geeksforgeeks.org/socket-programming-python/?ref=lbp

https://www.guru99.com/tcp-vs-udp-understanding-the-difference.html#:~:text=The%20full%20form%20of%20UDP,a%20packet%2Dswitched%20network