PyChat

Romano Francesco : P37000022

May 1 2020

Abstract

PyChat is a simple but effective Server-Client socket chat, written in Python Language. The purpose of this project is to show the functions of a socket chat written in python using socket and threading libraries. The general idea was to build an user friendly chat, easy to use and accessible to everyone. PyChat has an username feature, a break command that allows to instantly shut down the chat and a threading module that allows to send and receive multiple messages, without waiting for the other user's reply. These features will be explained more in the detail later. This project was made as assignment for the class of Computer Networks of the University of Naples Federico II. Thanks to Francesco Di Cicco, Italo Ferrante and Vincenzo Di Buono for helping testing the codes and the network.

1 Analyzing the code:

1.1 Server:

To get PyChat up and running the users have to first execute the Server code and then the Client one. For a more efficient step to step explanation we start with the server code.

First thing first we start by importing the core libraries for this chat: **socket** and **threading**. The socket library allows us to access the socket interface while using Python object-oriented style, *socket()* is our socket object, it has two variables: *AF_INET* which means that we are using the IPv4 address, so we need a server host and a port. The second variable is *SOCK_STREAM* which means we are using a TCP connection. TCP (or

Transmission Control Protocol) is a Transport Level net protocol which task is to establish a reliable connection for the exchange of informations. I chosen this protocol (instead of the UDP one) because, in order to establish a realible connection with no packet loss, TCP is the preferred one. Losing a packet in a chat application would result is a significant loss of information and this is something I wanted to avoid. In the "Tests and errors" section I show the handshake phase to further prove that a successful TCP connection has been established.

Threading instead allows us to build thread-based parallelism interface. This tool improves the "quality of life" of our chat, with this library we can create thread objects that allow us to send and receive multiple messages; without it we wouldn't be able to do so and instead we would have to wait for the reply of the other user. Threading is immediately initialized at the beginning with the lines *threading.Lock()* on two objects called **reply_lock** and **send_lock**, with this we create two Lock objects, a Lock is the lowest level of synchronization available and has two basic methods: *acquire()* and *release()*. With the first we block a thread until it's released by the last command.

We then start to create all the needed variables for our socket connection: **server_skt** is our socket that is using, as stated before, an IPv4 and a TCP connection. **server_host** is an object that uses a function named *gethostname()* this allows us to get the name registered in the host's computer, in my case it's simply named "DESKTOP". We also define **server_ip** to be later use as handy snippet to be sent quickly to the client user, without spending more time searching for the address; it is later printed so it can be quickly given. Lastly we define the server port, and we proceed with the *.bind()* method, leaving the first parameter empty (this is the part where the IPv4 address goes) and using our server_port as second parameter.

Later, the server asks for a name to be used as username, then we order our server to listen for exactly one connection, with the command *.listen()*, when a connection is established, two objects **conn_skt** and **addr** are created. The method .listen accepts a numeric variable, this is the amount of connection it will accepts, in our case only one. They will be used to communicate with the client user as well as also receive its username. Once established the connection, the server prints a message telling that the client user has connected to the server, showing also the chosen username.

After all this initialization process, we proceed by analyzing the reply and send functions called **thrd_send** and **thrd_reply** they are both very similar,

however one is used for receiving messages and one for sending them. They both consist of a While True cycle, with the first component being a feature that allows us to break the threads and thus the connection if the message "[bye]" is either sent or received. The reply function has a *.recv()* method that receives data from the socket. The number "1024" inside is the buffer size, which states the maximum limit of data that can be received. The if method has a *.release()* method, used to release the Lock object we previously created, it will be acquired in the main function. Both functions work with *encode()* and *decode()* for receiving and sending messages, this is required to make the message sent or received readable by the user.

We can now proceed to explain the main function: first of all we *acquire()* the lock objects so that we can start the threading process, then the code prints a message with the address and port of the client that connected with us. After that, it starts the threads for both the reply and the send function, the threads are created with the metohd *.Thread()* and started with *.start()*.

After starting the main function this is an example of a standard chat with this code, on server side:

```
DESKTOP ( 79.27.94.122 )
Enter your name: Francesco
The server is ready to receive
Trevor has connected to the chat room.
Type [bye] to exit the chat room.
Connected to : 192.168.56.1  :  57751
Me: Hi!
Me: :D
Trevor : Hello Frank! Been a while!
```

1.2 Client:

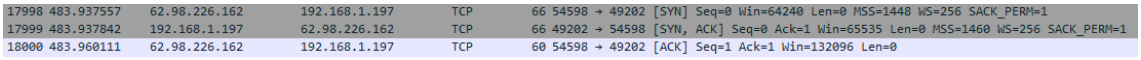
The client code is not much different from the server one, it has almost the same functions, both sending, receiving and main one. It has however a different approach to allow the client to connect to a server: it asks for address and port of the server the user wish to connect to, while also asking for the username to be used by the user, and to be received by the server. Then it tries to connect to the server, using the inputs given, using the *.connect()* method.

This is, instead, the same example of before but on client side:

```
Enter server address you want to connect to: 79.27.94.122
Enter the server port you want to connect to: 49202
Enter your name: Trevor
Successfully joined  Francesco 's chat room.
Type [bye] to exit the chat room.
Francesco : Hi!
Francesco : :D
Me: Hello Frank! Been a while!
```

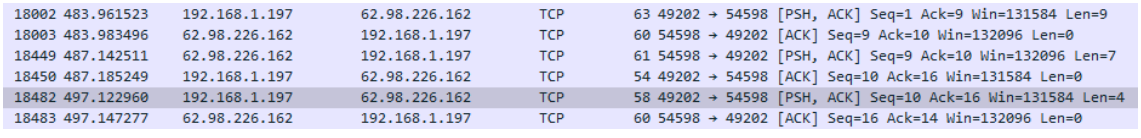
2 Tests and errors:

Three interoperability tests were conducted with six different client-server combinations, both helped showing issues on mine (and our) codes. The most particular error encountered were, first of all, port-forwarding my ip address. I solved it by entering my router settings and adding an exception for that address. During the testing phase I also encountered issues with having to wait for the other user's reply before writing (and sending) a new message. I solved this by adding the Threading library, at first it was a challenge to create and run the two threads, but then after some cleaning of my code and some debugging, I managed to solve the issue and it is possible to send more than one message at the time without waiting for the reply of the other user. These are the packets shown on WireShark for the handshake phase during the last interoperability test:



A three-way handshake phase is a method used by the TCP protocol to establish a Server-Client connection. It is done by exchanging SYN and ACK packets before the start of comunications.

This is an exchange of packets during the conversation:



3 References:

(Kurose, Ross) Computer Networking: A Top-Down Approach

4 Conclusion

For further questions and doubts, e-mail me on my personal mail:
francesco.romano.t@gmail.com
Or my professional mail:
francesco.romano31@studenti.unina.it