GUIÃO 6

Ex 1.2.

On the server's source code, the server is always wating for a message from a user, on the IPv4 address `0.0.0.0` , on the UDP port 5005.

On the client's source code, there are two functions for user interaction, and its IPv4 address is `127.0.0.1` , using the same UDP port as the server. Then, it waits for a message to then send to the server using that IP address and port, so that the server receives the message (which is constantly listening).

On Wireshark, per message, we can capture a UDP packet, which contains the information of the message (length, content), and it also has the port information.

The client uses the UDP port defined by the server to send the message. The client's UDP port is defined automatically on the socket creation.

Ex 2.2.

After analyzing the server's source code, we define the IP and port of the server (same as before). After that, the socket is created, and starts listening for incoming clients (maximum is 5). Then, after a client connects, it stores the address and socket of the connected client. It now creates a thread to listen for messages, while still being able to accept more clients. When receiving a message from a client, echoes the message back to it.

On the client, it defines its IP and port (same as before), and creates a socket, which then connects to the server. Then, it waits for a message to then send to the server using that IP address and port, so that the server receives the message (which is constantly listening). Finally before allowing the user to send another message, awaits for the server to echo the sent message.

On both ends, there is a exception handler, in case of one of them disconnects, it clears the used socket, and allows the server to listen to a new client.

On Wireshark, we first capture 3 TCP packets when the client connects to the server:

- First there's a Synchronization from the client to the server asking to connect.
- After that, the server sends a Synchronization/Acknowledge packet to the client to confirm the connection and to acknowledge its information and send it to the client.
- Lastly, the client sends back an Acknowledge packet to confirm it has received the information.

We then capture 4 more TCP packets per message exchange in the following order:

- A Push/Acknowledge packet from the client to the server which includes the message information as well as asking the server to confirm the message was correctly received.
- An Acknowledge packet from the server back to the client to inform the client the sent message was correctly received.
- A Push/Acknowledge packet from the server to the client echoing back the previously sent message and asking the client to confirm the echoed message was correctly received.
- An Acknowledge packet from the client to the server confirming the echoed message was correctly received.

Ex 3.2.

The difference between this and the previous server, is that now all clients are handled by a single thread, using the `selectors` module.

After registering a client, it will look for activity/events (which will be in this case messages), so it can then trigger the `handle_data` function.

The `handle_data` function works similarly as the function `handle_client_connection` in the previous file. It waits for a message and sends it back, and if the connection closes or loses connection, it closes the socket.

Every single client has a key, provided by the selector, which is printed everytime it establishes a connection, it receives data, or disconnects.

Ex 4.2.

On both the server the client, there are a few changes. Now, instead of sending a simple message, the client sends a structured message using the `struct` python module. This is still a normal message, but we define how it should later be decoded.

If we take the example of this code, the struct `'!BLL20s'` represents a struct that contains the following:

- !: Represents the Byte Order (the protocol), in this case Network (which is the same as big-endian)
- B: Represents an unsigned char
    - Size: 1 byte
- L: Represents an unsigned long (64-bit integer)
    - Size: 8 bytes
- 20s: Represents a character array with 20 spaces
    - Size: 20 bytes

This way we are able to send not only the message, but we are also sending the version of the message, the order number of the client and the message size, this one being important for the server to unpack the struct since the message is a character (byte) array of variable size and the server needs to know how many characters to unpack.

On the TCP packet, we still see all the data compacted into its Data field.

Ex 4.3.

To adapt the code, we first edit the server. When receiving a message, it then sends back the struct to the connected client.

On the client, after we send the packet, we then await the response from the server and once the replies comes, we unpack it and print it.

Ex 5.2.

In this case, the packet is still similar to the previous one.

The difference now is, instead of sending the struct at once, we first send the struct header. This header contains the version, order and size (the first elements of the structs used before). The server then requests the number of bytes passed as the size. This way, we can now send a message with whatever size we want.

Ex 5.3.

For this, we adapt the code the same way we did in 4.3.