Final Project

SBChat

Submitted By:

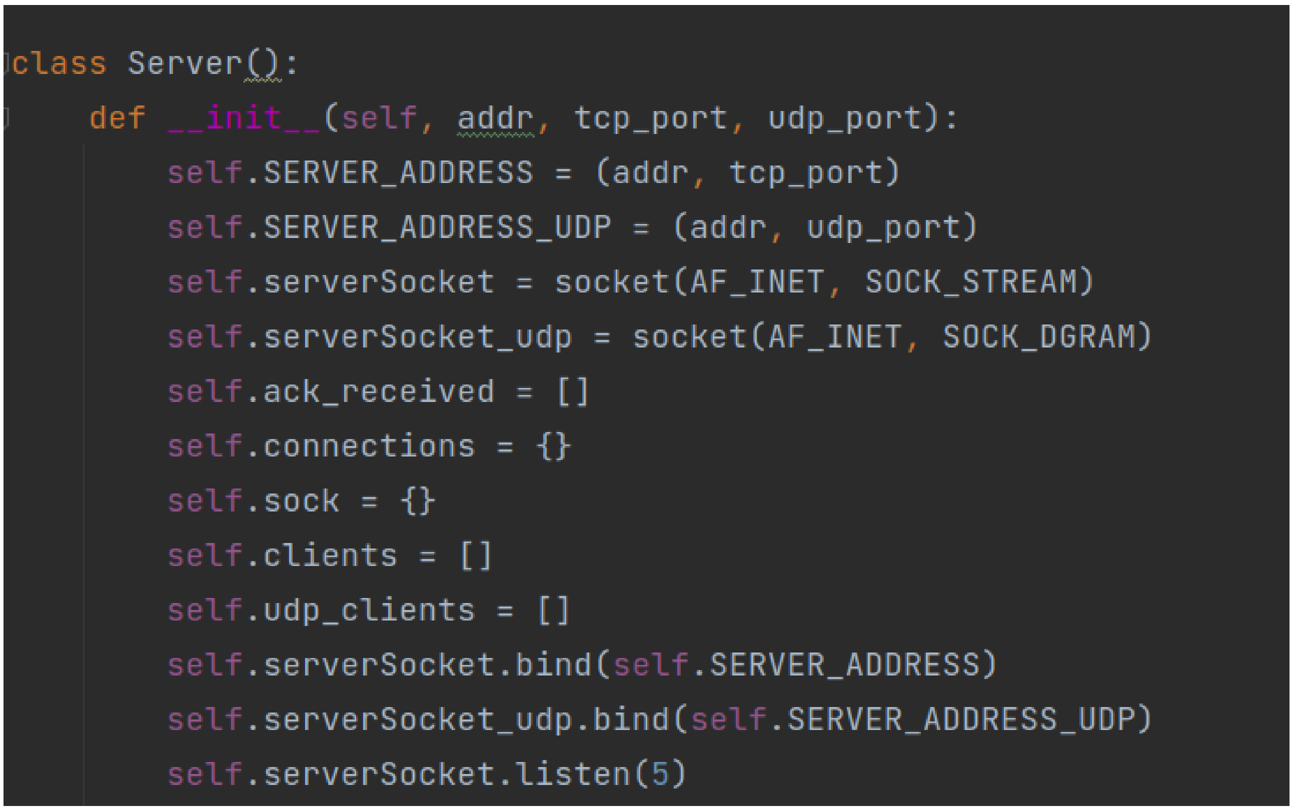
Bar Goldenberg 209894286
Sappir Bohbot 316416429

Part 1:

Primitive chat app over TCP:

SERVER:

Setting up the server:



```
class Server():
    def __init__(self, addr, tcp_port, udp_port):
        self.SERVER_ADDRESS = (addr, tcp_port)
        self.SERVER_ADDRESS_UDP = (addr, udp_port)
        self.serverSocket = socket(AF_INET, SOCK_STREAM)
        self.serverSocket_udp = socket(AF_INET, SOCK_DGRAM)
        self.ack_received = []
        self.connections = {}
        self.sock = {}
        self.clients = []
        self.udp_clients = []
        self.serverSocket.bind(self.SERVER_ADDRESS)
        self.serverSocket_udp.bind(self.SERVER_ADDRESS_UDP)
        self.serverSocket.listen(5)
```

- in this explanation we will only talk about the TCP setup (all UDP related code will be explained later.)

first we save the server address as a tuple, the input of addr and tcp_port is taken from the graphical interface (will be shown later).
Then we open a server socket and bind it to that adress.
We set serverSocket.listen(5), because in this project, we need to accept up to 5 users at the same time.

How the run() method works:

first we wait for a user to connect to the server
- conncetion_socket, addr_client = serverSocket.accept()
then if the address is new, (has never connected)
we add the user to the ‘connections’ dictionary which holds an address as a key and a user as a value, once a user has connected he immediatly sends his name to the server, so using that name we save him in the dictionary.

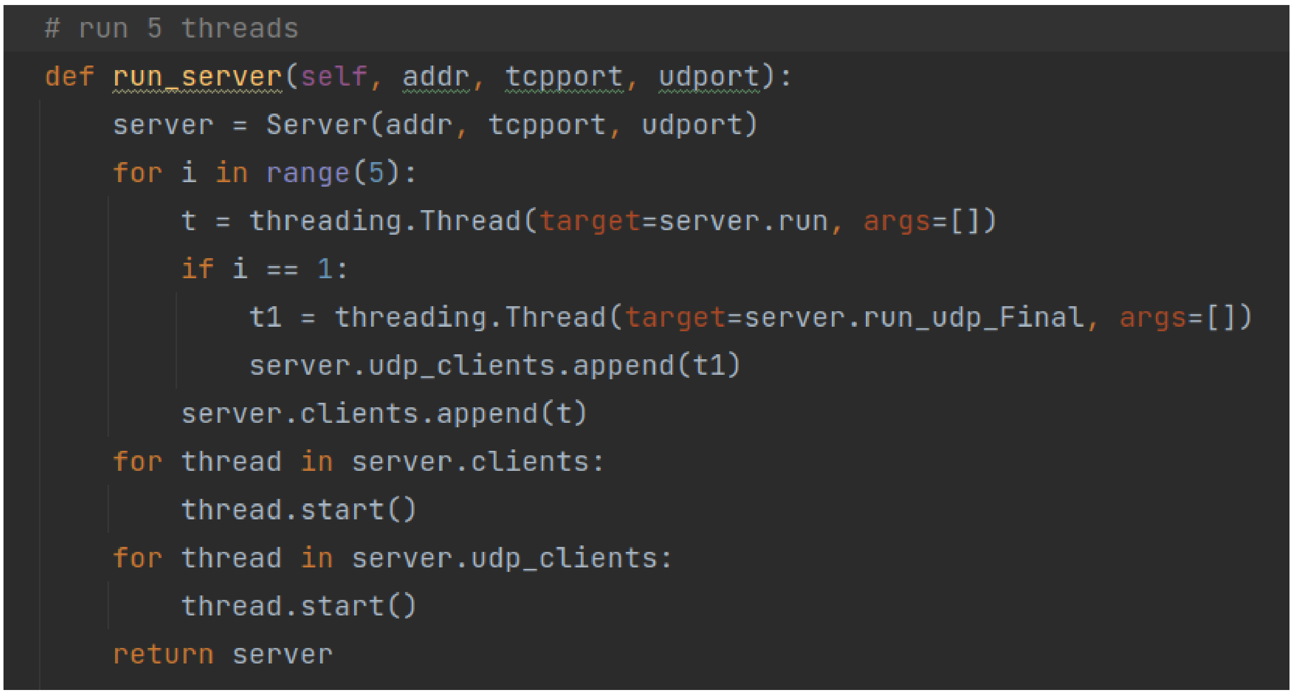
The second dictionary we use is the ‘sock’ dictionary which holds a user as a key and a connection_socket as a value.
By default each user is connected to them self in the begining.
If a user wants to connect to another user he writes the command <connect_username>, where username is the person he want to connect to, for example, lets say there are two users, Bar and Sappir.
If Bar want to connect to Sappir he will write <connect_Sappir>.
When that command is sent to the server, the server changes Bar’s ‘connected_user’ attribute to Sappir’s socket and vice versa, then when ever Bar sends a message the message is received by the server and sent to Sappir usings Sappir’s socket.

If the user wants to send a message to all connected users he will write <msg_all>msg.
For example, if Bar wants to say ‘Hello!’ to all connected users, he will write <msg_all>Hello!
When the server receives a message of this type, it itterates over the sock dictionary’s values, and send the message to all the connection sockets (except for the senders socket).
Overall steps in run():
before the while True loop, we wait for a connection, after we receive a connection we wait for incomming messages and send them acording to the explanation above.

How do we acheive 5 users connected at the same time if only one function is listening for a connection?

Threads. if we only have one function that does what we want, why not run it 5 times simultaniously?
So we use threads the function we call to start the threads is called ‘run_server()’.

run_server():



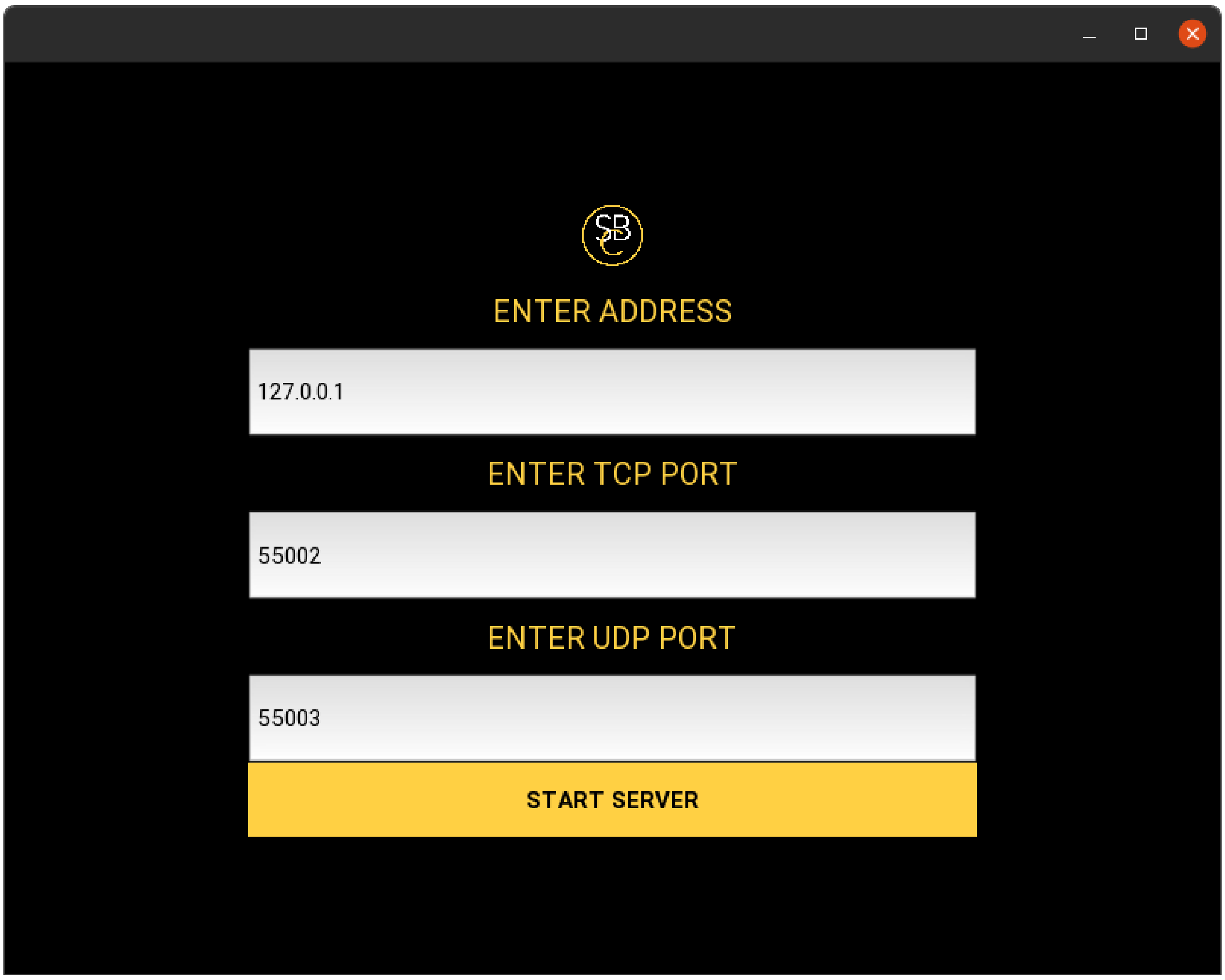
```python
# run 5 threads
def run_server(self, addr, tcpport, udport):
    server = Server(addr, tcpport, udport)
    for i in range(5):
        t = threading.Thread(target=server.run, args=[])
        if i == 1:
            t1 = threading.Thread(target=server.run_udp_Final, args=[])
            server.udp_clients.append(t1)
        server.clients.append(t)
    for thread in server.clients:
        thread.start()
    for thread in server.udp_clients:
        thread.start()
    return server
```

as shown above, we initialize a server instance and start 5 threads running the run() function.

Server Graphical Interface:

for this project we used the KIVY library to create our GUI.

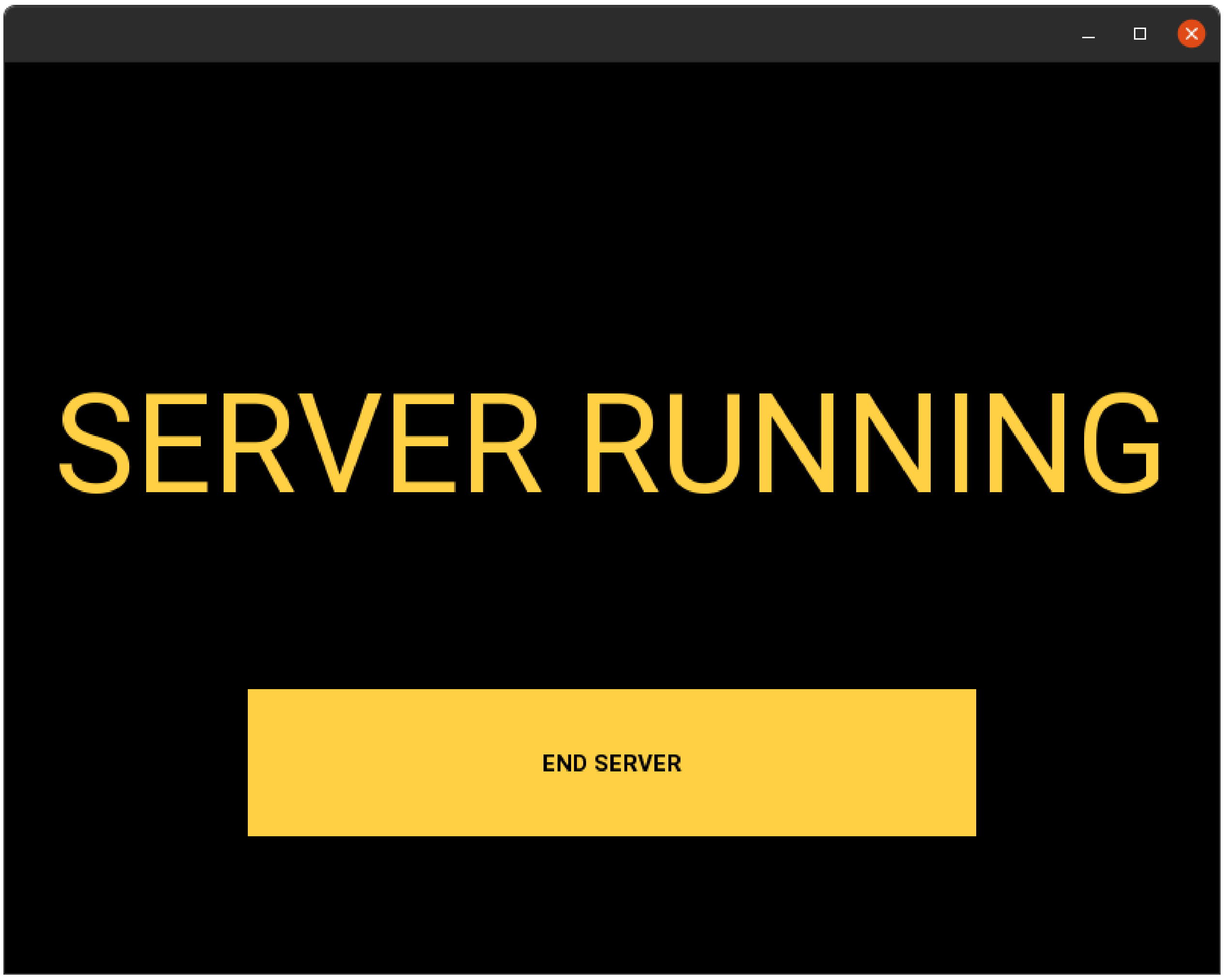
Start Screen:



as shown above, we have three text boxes. one for the servers adress, and two more for configuring ports.

To start the server we click on start server.

Running screen:



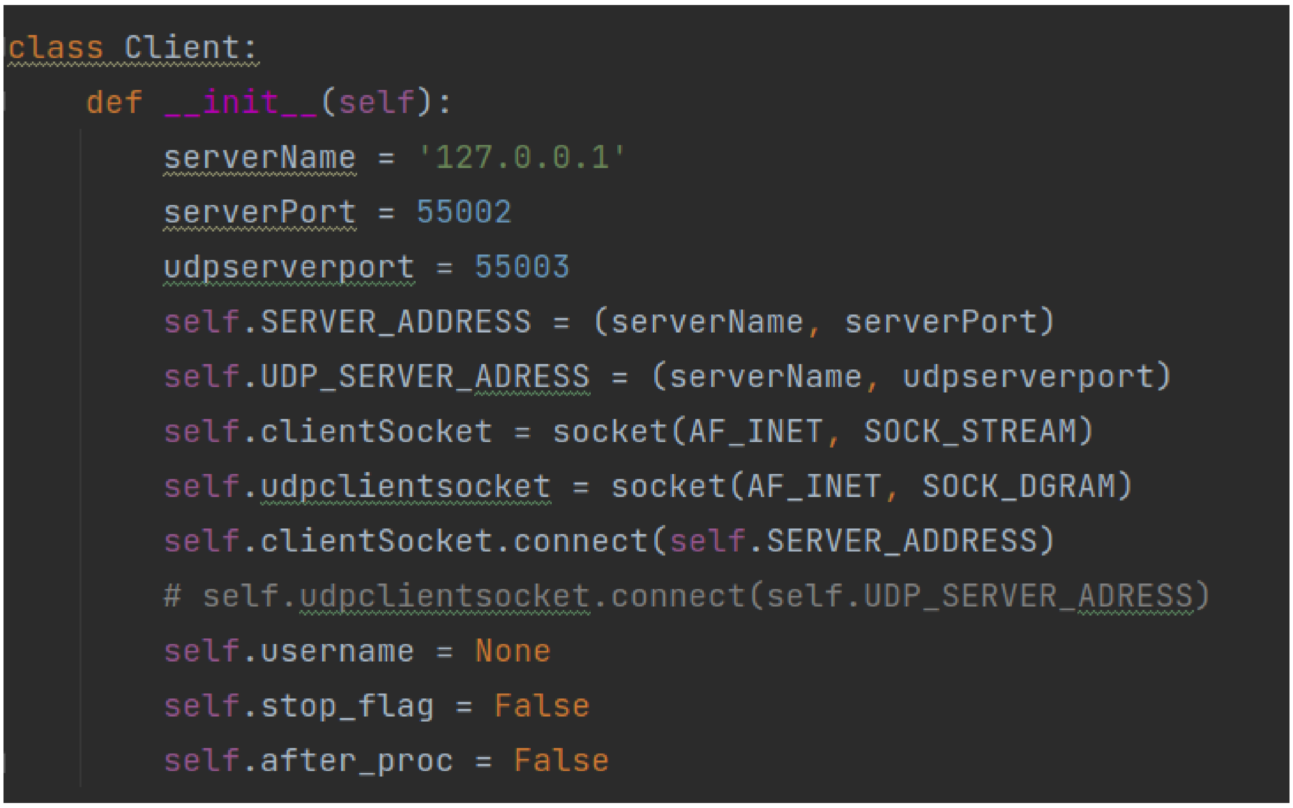
this screen just shows that the sever is running to close the server click on end server.
When end server is pressed, we close all threads and close all sockets.

CLIENT:

Setting up the client:

the client is split up into two seperate classes, Client.py and User.py.
The client class is used for setting up all network related things, and the user class is used for saving clients personal attributes.

Client.py:



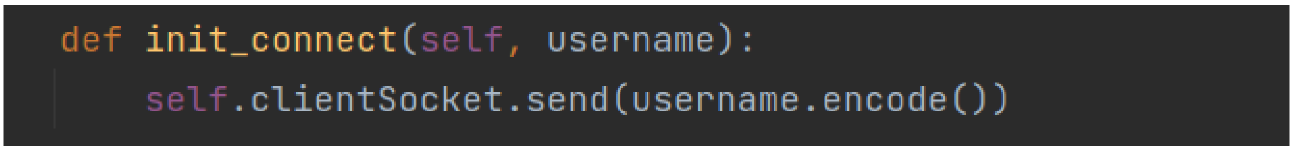
```python
class Client:
    def __init__(self):
        serverName = '127.0.0.1'
        serverPort = 55002
        udpserverport = 55003
        self.SERVER_ADDRESS = (serverName, serverPort)
        self.UDP_SERVER_ADRESS = (serverName, udpserverport)
        self.clientSocket = socket(AF_INET, SOCK_STREAM)
        self.udpclientsocket = socket(AF_INET, SOCK_DGRAM)
        self.clientSocket.connect(self.SERVER_ADDRESS)
        # self.udpclientsocket.connect(self.UDP_SERVER_ADRESS)
        self.username = None
        self.stop_flag = False
        self.after_proc = False
```

as shown above, we save the server’s address as a tuple (self.SERVER_ADDRESS).

We open up a client socket (AF_INET = IPv4, SOCK_STREAM = TCP, SOCK_DGRAM = UDP).

Then we connect the client socket to the tcp server.

init_connect():



```python
def init_connect(self, username):
    self.clientSocket.send(username.encode())
```

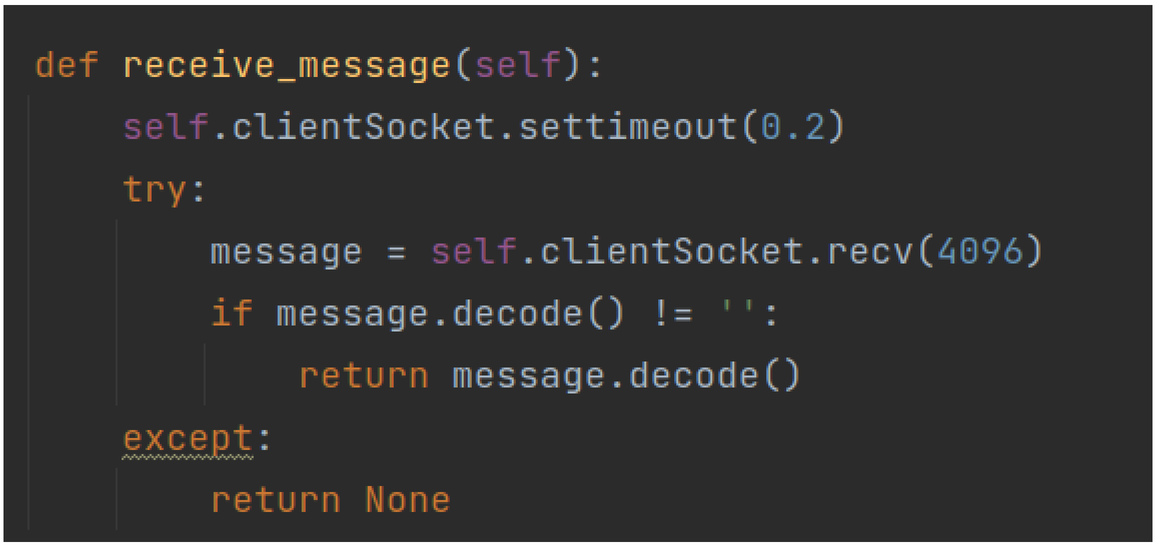
this function is called right after init, is sends the username to the server, so the server can add him to the connections and sock dictionaries.

send_message():

sends a message to the server, (code is long because of RDT over UDP file transfer, will be explaned later.)

receive_message():

calls clientsocket.receive(buffer) function.

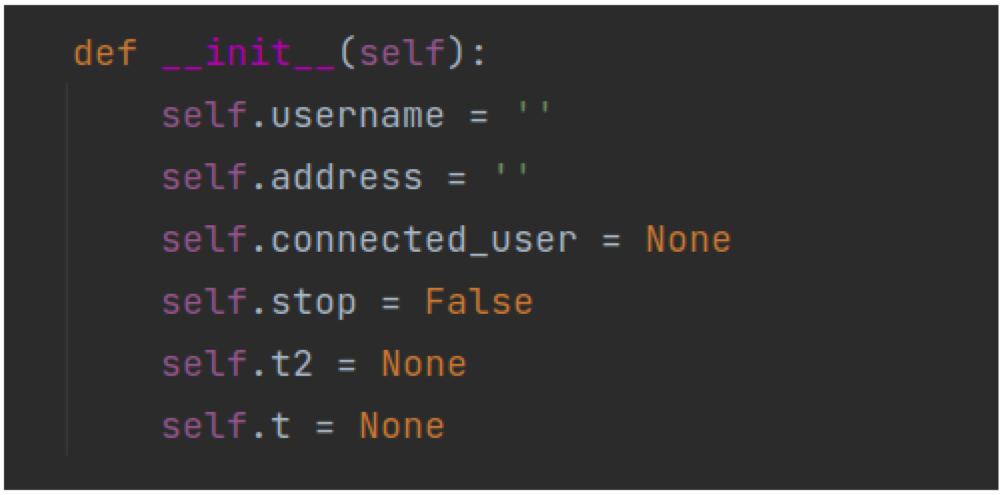


```python
def receive_message(self):
    self.clientSocket.settimeout(0.2)
    try:
        message = self.clientSocket.recv(4096)
        if message.decode() != '':
            return message.decode()
    except:
        return None
```

if the function times out return None.

User.py:

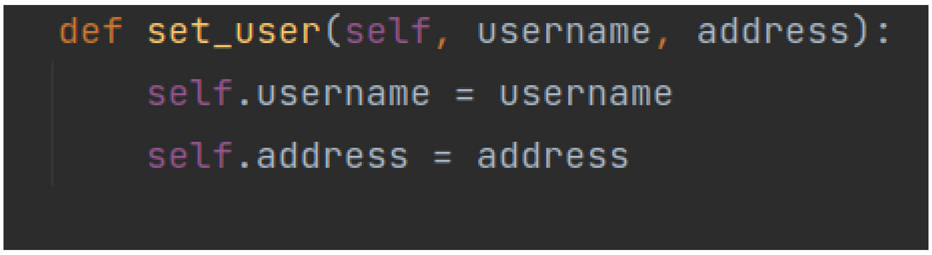
init():



```python
def __init__(self):
    self.username = ''
    self.address = ''
    self.connected_user = None
    self.stop = False
    self.t2 = None
    self.t = None
```

sets all attributes thing to “” or None.

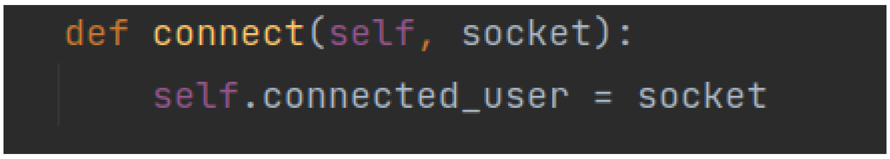
set_user():



```python
def set_user(self, username, address):
    self.username = username
    self.address = address
```

setter for user class.

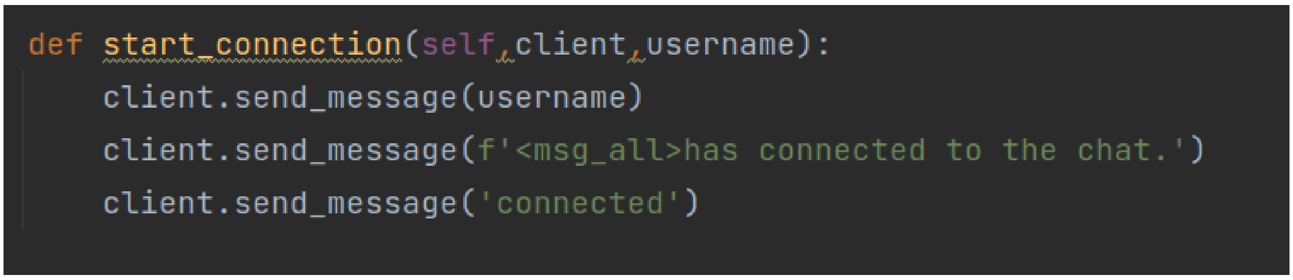
connect():



```python
def connect(self, socket):
    self.connected_user = socket
```

connects a user to a given client socket.

start_connection():



```python
def start_connection(self,client,username):
    client.send_message(username)
    client.send_message(f'<msg_all>has connected to the chat.')
    client.send_message('connected')
```

sends username to server using client, then send a message to all users that a new user has connected.

receive_message():

a loop waiting to receive messages from the server and update them on the GUI.

send_message():

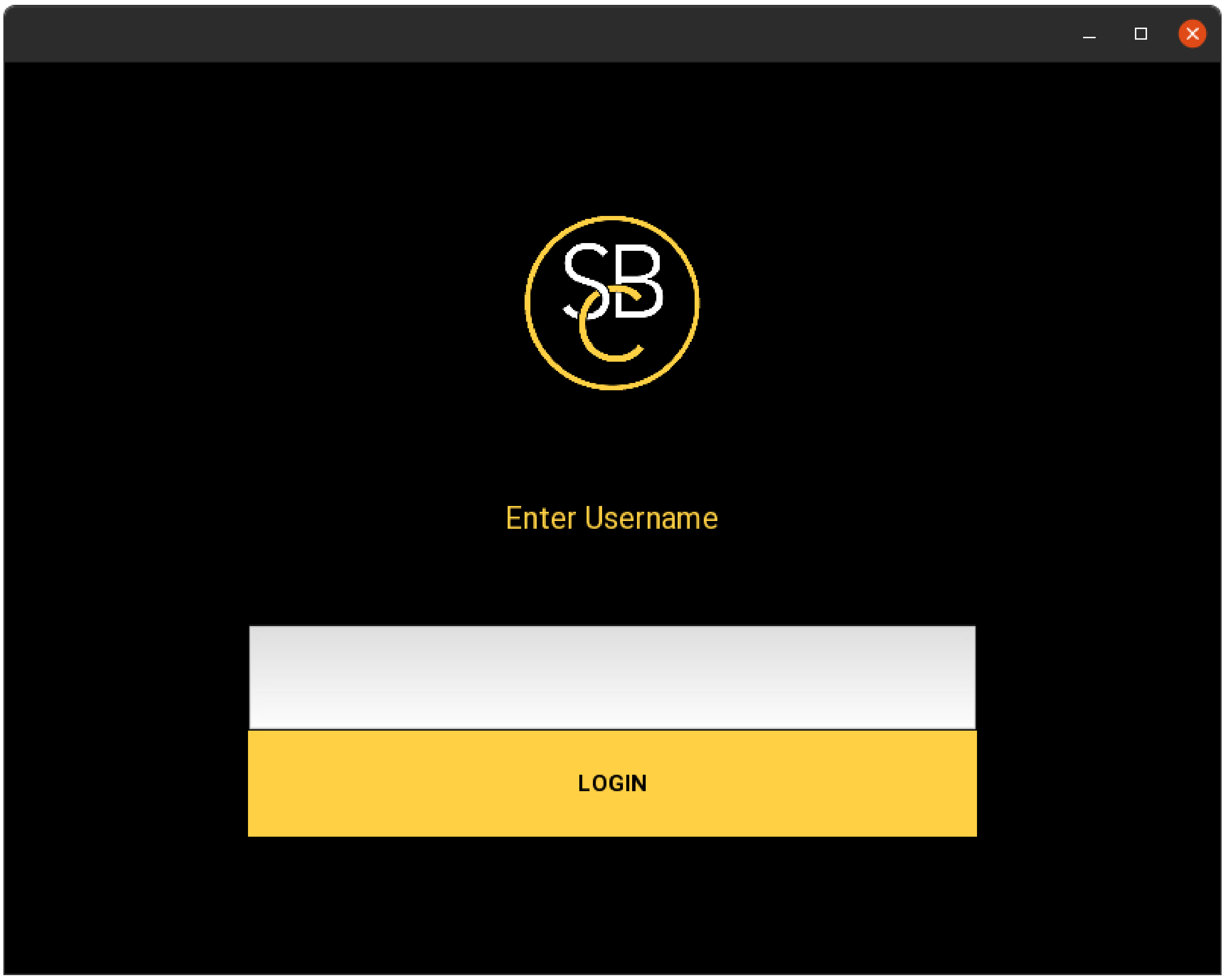
calls the send_message() function from client.

listen():

runs the receive_message() function using a thread.

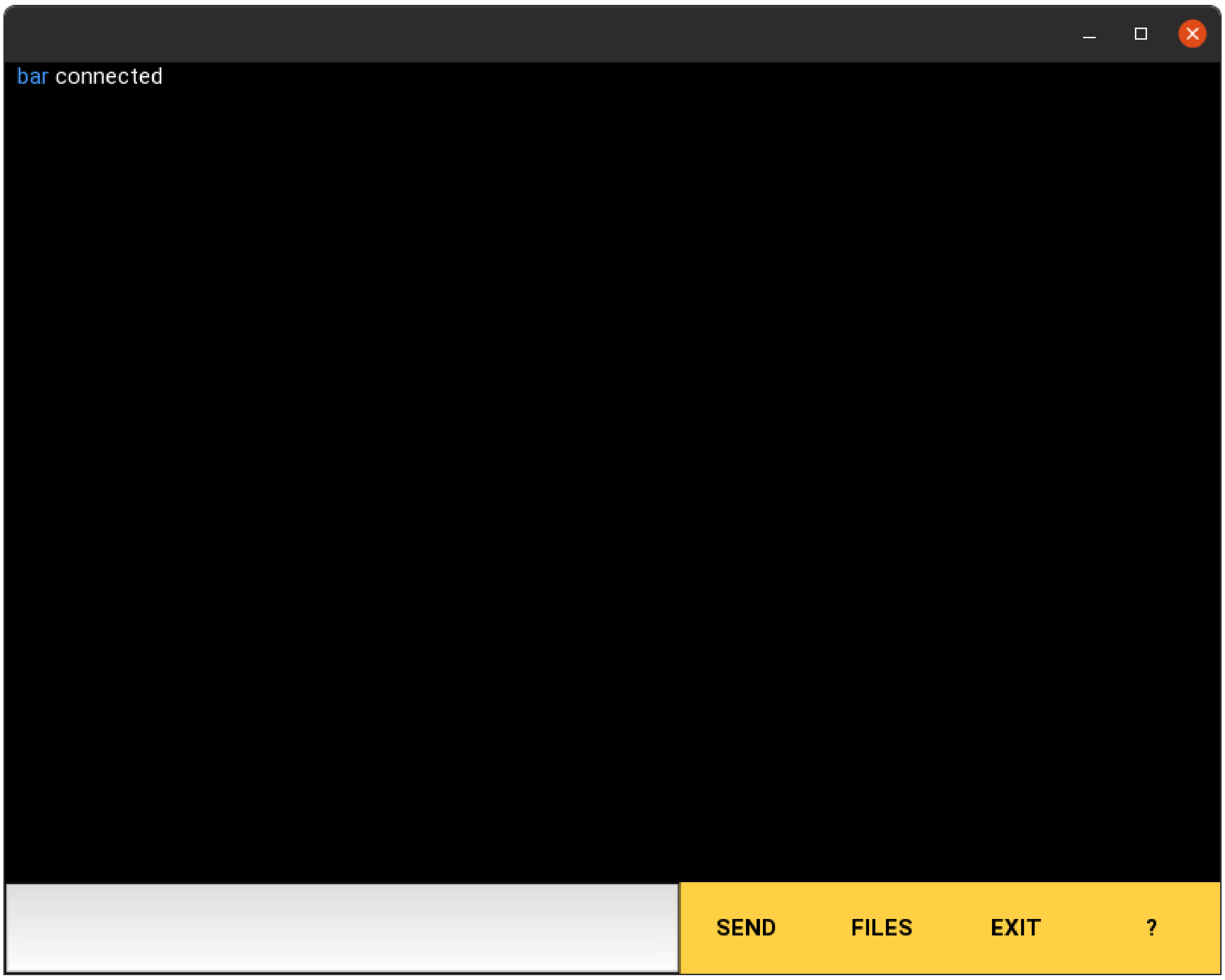
Client GUI:

Login Screen:



to login just enter username and click LOGIN.

Chat Screen:


bar connected

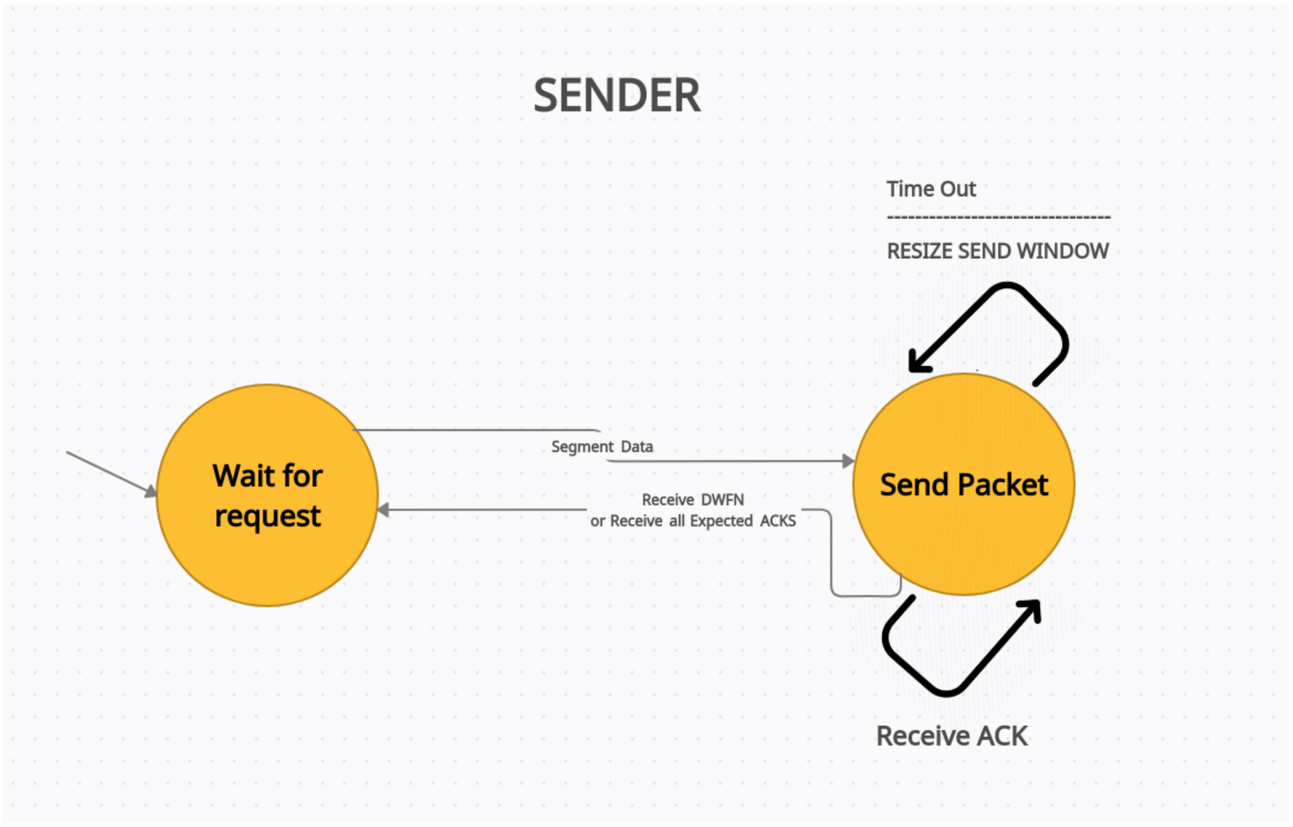
as shown, we have a screen that displays all outgoing and incoming messages, a text box to input a message you want to send, a send button to send (can also press enter), a files button to choose what files you want to download, a help button which shows all the commands that you can use, and an exit button to safley exit the program.

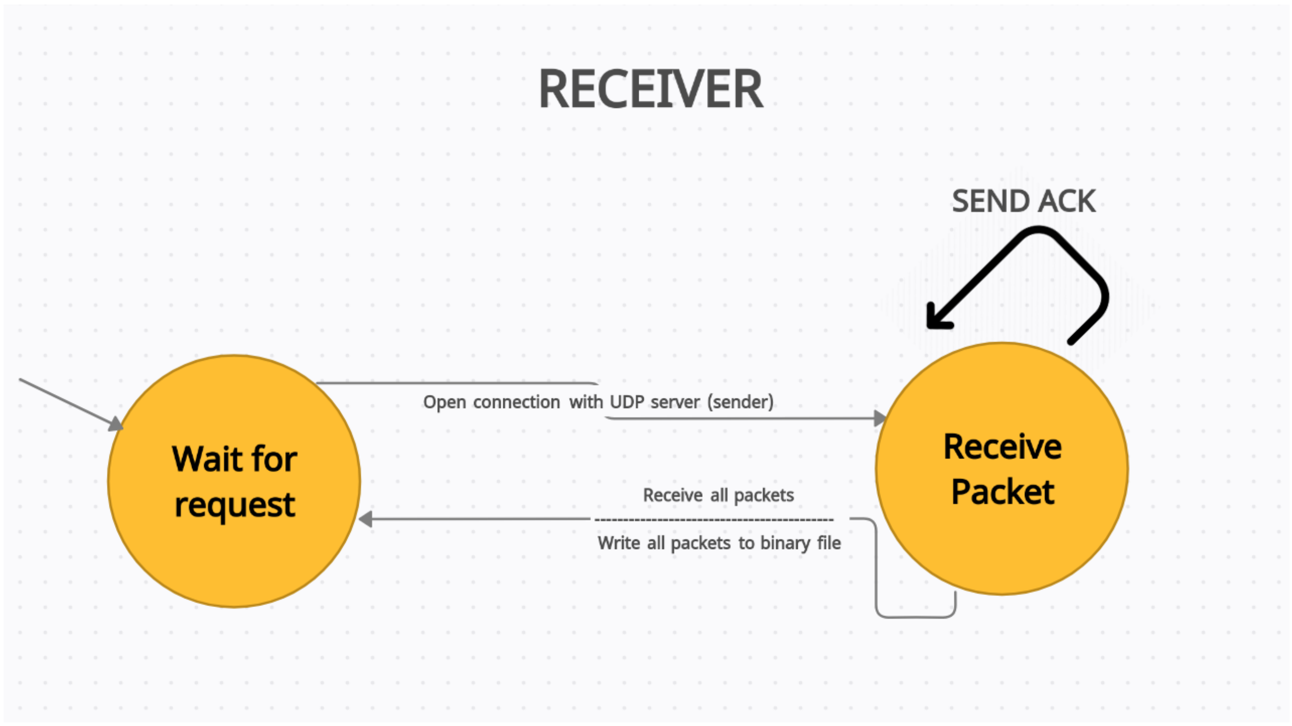
Part 2:

Fast Reliable UDP.

In this project we decided to implement the selctive repeat protocol for reliable data transfer. And reno for congestion control.

State machine for sender (server) and receiver (client).



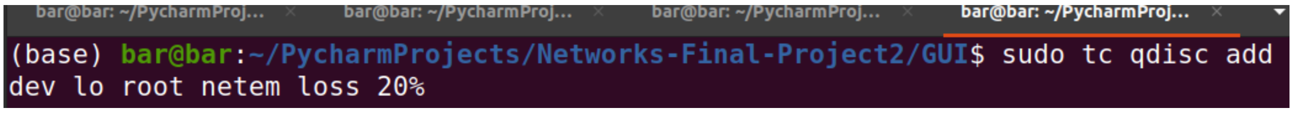

RECEIVER
SEND ACK
Wait for request
Open connection with UDP server (sender)
Receive Packet
Receive all packets
Write all packets to binary file

How do we make up for packet loss?

We use selective repeat.
Our program sends a number of packets at once.
After sending the sender opens a timer for an ACK to receive,
(the ack is a sequence number).
If the ack did not come in the window of our timeout, the server send that packet once more untill it recieves an ack.

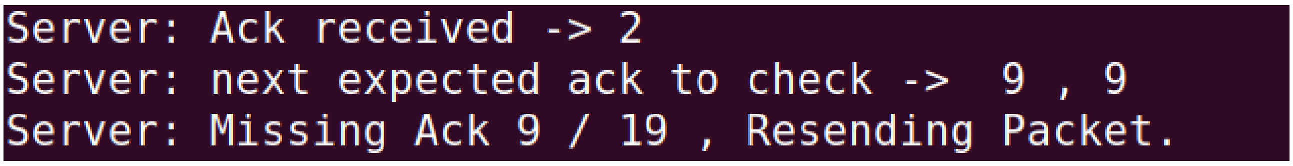
Now we shall demonsrate how it works:

First we will loose packets:



```
(base) bar@bar:~/PycharmProjects/Networks-Final-Project2/GUI$ sudo tc qdisc add
dev lo root netem loss 20%
```

now we will show a snippet from the log show the resending of packets:



```
Server: Ack received -> 2
Server: next expected ack to check ->  9 , 9
Server: Missing Ack 9 / 19 , Resending Packet.
```

the server noticed it had not received an ack for packet 9 and resent it.

Wireshark pcap file is included with this project.

How do we deal with latency?

We use congestion control in perticular we use the reno algorithm,
first we will define some terms:
window size := the amount of packets we send at once.
Threshhold := a varieable that defines the point of which we switch from exponential growth to linear growth.

We start with a window size of 2 packets and a threshhold of 1,000,000 each time we are able to send all packets without packet loss or latency we multiply the window size by 2, untill we reach or pass the threshhold, were from there on we increment the window size by one.
If there is any packet loss we decrease the window size back to 2 and set the threshold to the last window size before we started loosing packets.

How to run the chat app:

open the GUI folder from the terminal.

-To open the server type in the command:
python3 ServerGUI.py

-To open the client and connect type the command:
python3 SBChatApp.py

Part 3:

Question 1:

We will describe the steps from connecting the computer to the switch for the first time, till the other side receives the message.
When we first connect the host (our computer) to the switch, we will need the configuration information (such as are IP address, the DHCP server IP, the DNS server IP, the Subnet Mask and basically all the info are computer needs in order to function on the network). We will get this service from the DHCP server (Dynamic Host Configuration Protocol):
1. Discovery – The host will send discovery message as broadcast to all the computers on the LAN (Local Area Network), in order to find the DHCP server (this packet wont contain any destination IP, so it will stay in the LAN).
2. Offer – If there are DHCP servers in the LAN with a available IP address, our host (computer) will get offer packet witch contain all of these IP addresses.
3. Request – the host sends a request packet with all the information he chooses (IP addresses and more...) as a broadcast to all other computers in the LAN, so it will let all the other servers know the chosen address (and all the DHCP servers whose offer wasn't accepted will assign back the IP addresses that where offered as available).
4. Acknowledge – the host will get confirmation for the request, after the acknowledge the computer will start to use the assigned configuration info (this packets contain more details that is essential such as subnet mask, DNS servers IP and more…).

After we connected to the switch, In order to send an HTTP request, we will use the DNS server (Domain Name System) to get the destination IP (we need to know where to send the HTTP request). The DNS Protocol is an application layer

protocol that takes an URL address (that easy for humans to remember) and converts it to a IP address. DNS query encapsulated in UDP, encapsulated in IP, encapsulated in Ethernet and it also need the Mac address.
In order to get the Destination MAC address we will use the ARP Protocol in the link layer, witch works as follows:

1. ARP query broadcast to all the computers on the subnet.
2. Router receive the ARP query and sends an ARP Reply given MAC address of the router interface.

Now the host knows the MAC address of the first hop router, so it can now send frame containing the DNS query. The IP Datagram containing DNS query is forwarded in the LAN till it gets to the first router, it keeps moving forward from one subnet to anther (with tables like OSPF…) till it gets to the DNS server. The DNS server reply to the host with the IP address of the requested Web Server.
Now the host have almost everything he need to send the HTTP request, it only misses the actual connection to the Web Server – the TCP Socket.

The Host will open an TCP Socket, but in order to complete the connection, it will do the three way hand shake:
- First, the host will send a SYN message to the Web Server, that means the host would like to connect with the Web Server.
- Then, the Web Server will get the SYN message, and reply with a SYN-ACK message, that means that the Web Server is accepting the connection query.
- Finely, The Host received the SYN-ACK message from the web Server, and reply with an ACK witch representing the end of creating the new connection between the two Hosts.

After we finished to establish the connection between the host and the Web Server, we can send massages, HTTP requests and more, or we can end the connection if we like to.

Question 2:

What is CRC?
CRC or cyclic redundency check is a error detecting algorithm used in networking protocols.
How it works:

let x be the message we want to send, x is a stream of bits.
We add to x, y 0 bits to the end.
y- the number of bits we add to x.
Then we take a number p, usualy a prime number so that the probability of error detection is higher p has y bits.
We calculate r = x%p
then we subtract it from p
num = p-r
then we construct the final message:
msg = x + num
now if we calculate msg%p we get 0.
if the message does not devide p when it reaches the destination we know we have an error.
We can also represent the bits as polyomials and do polynomial devision on them to receive the same thing.

Question 3:

What is the difference between HTTP 1.0, HTTP 1.1, HTTP 2.0 and Quic?

First the HTTP 1.0 was published in 1996, it was made in a way that for each packet you want to request, you open TCP socket, sends your request, the server would reply the requested data and the connection were closed after each packet! (in that time there wasn’t big data to send – mainly texts, the RAM memoy was very limited, and the TCP socket took lots of storage so they tried to reduce the connection time as much as they can).

So, HTTP 1.0 was 1. Slow, 2. New TCP connection for each request.

HTTP 1.1 was published in 1999, this was the dominant protocol for 16 years (until 2015). It had some very strong improvements from HTTP 1.0, a 'Keep Alive' header was added so the server wont close the TCP connection so fast, but keep it connected for faster data transfer. Also it added: 1. Persisted TCP Connection 2. Low latency. 3. Streaming with chunked transfer. 4. Pipeline.

HTTP 2.0 introduced in 2015, it showed some revolutionary updates such as:

1. Multiplexing – instand of single connection as HTTP1.1, the new version could take a lot of requests that are the same but from different source, and get the requested data once from the server, and send it back to all of the ssources.

2. Comprssion – the HTTP 2.0 new how to compress and decompress the header and the data, such that the sending can be done faster and safer.

3. Secure By Default. 4.Server Push. 5. Protocol Negotiation During TLS

QUIC is a new transport layer protocol designed by google (since 2018 it was renamed as HTTP 3.0). it has all of HTTP 2.0 features, but the main difference is QUIC replaces TCP with UDP that has congestion control (So it’s even faster!).

Question 4:

Why we need port numbers?

Port numbers are important because they allos the host that received traffic to know where to deliver the data (to which application). Without port numbers, each server could get only request to one application (and same for sending, the server wont be able to send packets to several application on other server).

Question 5:

What is subnet? Why we need it?

Subnet is basically a segmented piece of a larger network (network inside a network). More specifically, subnets are a logical partition of an IP network into multiple, smaller network segments.
The main benefit of subnet the fact it makes the traffic go much faster, since there could be millions of connected devices, and it could take a lot of time for the data to find the right Device / Server. This is why subnetting comes in handy: subnetting narrows down the IP address to usage within a range of devices…

Question 6:

Why we need MAC address? Isn't the IP good enough?

The IP is a dynamic address, it assigned to the host only when he connect to the switch, but as soon as he disconnect the same IP can be assigned to different host.

This dynamic situation is problematic because a host can send a message to a server, the server will reply but un that time the host will disconnect and another server (that just connected to the same LAN as the host) can received the message reply that was never meant for him.

MAC address is a unique address on the network card. So one can uniquely identify your device from it. It cannot be changed but nowadays also can change it. These features make the MAC an essential key to proper delivery of messages.

Question 7:

What is the difference between NAT, Router, Switch?

First, NAT is a method, while Router and Switch are devices. One is for exchanging data inside the LAN and the other is routes data from one network to another.
The NAT is a method in which one or more local IP addresses are translated into one ore more global IP address (where the goal is to reduce the global IP usage).
Switch is a device that is used for exchanging data inside the LAN, it's a device that has multiple ports that accepts Ethernet connections from network devices. It also learns and stores the MAC addresses of all the connected devices so when a packet is sent to a Switch, it only directed to its destination port.
And last, the Router is a device that exchange data from one network (LAN) to another, when a packet get to the Router, according to its IP address, it decides if the packet sent to his network or another, if it wasn't meant to its own network, it will the send the packet forward to another network. So, a Router is essentially a Gateway of a network.

Question 8

what are the methods to overcome the lack of IPV4?
When invented in 1980, the IPV4 engineers was diseing it such that there will be about 4 billion (2^32) optional addresses, they never thought that one day we will reach this number. Today we suffer from lack of IPV4 addresses, we fight this problem in some methods:

- Classless Inter-Domain Routing (CIDR).
- Network Address Translation (NAT).
- Internet Protocol version 6 (IPv6), as a long-term solution.

CDIR – In the CIDR notation, addresses are written with a suffix, introduced by a slash " / ", that indicates the number of bits of the prefix.

NAT - The NAT is a method in which one or more local IP addresses are translated into one or more global IP address (where the goal is to reduce the global IP usage). This way a several hosts can be represented as just one host as a single global IP.

IPV6 - IPv6 is the next generation Internet Protocol (IP) address standard intended to supplement and eventually replace IPv4. The IPv6 has a lot of benefits such as: bigger addresses option (2^128 optional addresses) no more NAT, Auto Configuration, simpler header format and more…

Question 9:

BGP and RIP.

BGP RIP and OSPF.

BGP RIP and OSPF.

BGP RIP and OSPF.