AMERICAN UNIVERSITY OF BEIRUT

EECE 350: Computer Networks

Chatting App Project

Group 1:

Elias El Khoury

Lynn Olleik

Zeinab Mazraani

Prof. Ayman Tajeddine
Spring 23-24

Table of Contents

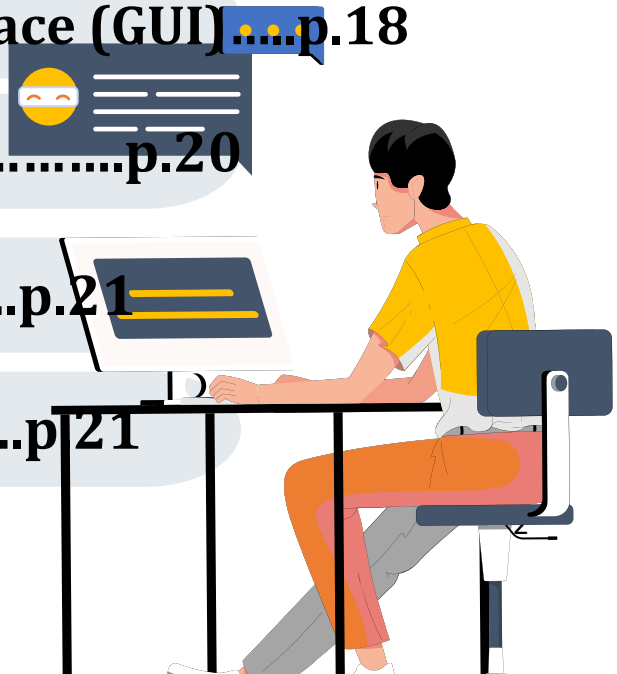


Table of illustrations

01 Introduction

1.1 Description:

- The objective of our project is to create a chatting application using Python. It focuses on ensuring reliable data transfer under unreliable network conditions while using UDP.

  While reliable local area networks can test our application, we also used ***Ubuntu*** virtual machine equipped with a built-in network emulator ***Netem*** to implement unreliable network conditions.

- We made sure to tackle several unreliable conditions such as: random packet drops, damaged packets, duplicated packets, and delayed packets, as well as a variety of different link bitrates and latencies (propagation delays) using ***Netem***.

- The chatting application operates on a peer-to-peer basis, where two peers establish a direct connection and relay messages between each other as required.

1.2 Phases and Requirements:

The project is divided into 4 consecutive phases:

02 Phase 1: Preparation Phase

- Installing Virtual Machine on MacOS and Windows: VMware or Virtual Box
    - Link for VMware: https://www.vmware.com
    - Link for Virtual Box: https://www.virtualbox.org/wiki/Downloads
- Installing Linux OS with Ubuntu 23.10
    - Link for Ubuntu: https://ubuntu.com/download/desktop
- Installing VSCode on Linux
    - Link for VSCode: https://code.visualstudio.com/download
- Setting up python3 environment
- Setting up netem packages
    - Link for Netem: https://manpages.ubuntu.com/manpages/bionic/man8/tc-netem.8.html

⇨ How to use NETEM on Linux to test unreliability of transmission?

1- Select the test we wish to do:
    a. Delay: sudo tc qdisc add dev lo root netem delay 6000ms
    b. Packet loss: sudo tc qdisc add dev lo root netem loss 50%
    c. Duplication: sudo tc qdisc add dev lo root netem duplicate 50%
    d. Reordering: sudo tc qdisc add dev lo root netem delay 100ms reorder 25% 50%\
2- Enter the corresponding command in the terminal.
3- Reset before changing tests using the following command:
    a. sudo tc qdisc del dev lo root netem

For our reliability tests, we ran our codes in phases 2 and 3 under all unreliable condition: delays, packet losses, duplication, and reordering.

03 Phase 2: Sending Messages

3.1 Establishing UDP Connectivity

- Import socket and threading python libraries.
- Create addresses for each of peer 1 and peer 2, each with the local host IP address (“127.0.0.1”) and unique port number (50010 & 50000 respectively).

```
# Save user/host's IP address
host_name = socket.gethostname() # Save hostname
host_ip = socket.gethostbyname(host_name) # Save host IP
# Save addresses for peer 1 & peer 2
user_1_address = (host_ip, 50000)
user_2_address = (host_ip, 50010)

# Create UDP socket for peer 1 & bind it to its address
user_1_UDP_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
user_1_UDP_socket.bind(user_1_address)
# Create TCP socket for peer 1 & bind it to its address
user_1_TCP_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
user_1_TCP_socket.bind(user_1_address)

user_1_TCP_socket.listen() # listen for TCP connection
print("I am ready to receive files!")

# Start Chatting UDP
start_chatting(user_1_UDP_socket, user_2_address, user_1_TCP_socket)
```

Figure 1: Code Sample from Peer 1 for UDP Connectivity

- Bind the addresses for each of peer 1 and peer 2 to their respective sockets.
- Use the concept of Threading because we need to run concurrent simultaneous processes and connections between peer 1 and peer 2.

```
# Function that starts the chat application
def start_chatting(user_UDP_socket, client_address, user_TCP_socket):
    print("Start up your conversation!")
    print("Note: To send a file, input the message '<I WANT TO SEND A FILE>' and follow the instructions!")
    threading.Thread(target=send_message, args=(user_UDP_socket, client_address, user_TCP_socket)).start() # Thread for sending messages
    threading.Thread(target=receive_packet, args=(user_UDP_socket,)).start() # Thread for receiving packets
    threading.Thread(target=process_packet, args=(user_UDP_socket, client_address)).start() # Thread for processing packets
    threading.Thread(target=handle_missing_packets, args=(user_UDP_socket, client_address)).start() # Thread for checking for missing packets periodically
    threading.Thread(target=receive_TCP_file, args=(user_TCP_socket,)).start() # Thread for checking if there is a file to receive
```

Figure 2: Code Sample from Peer 1 for Threading

- Use the methods socket.send() and socket.recvfrom(1024) to send and receive messages.
- We have created two advanced functions: send_message() and receive_packet() which respectively sent messages as packets, saved them in “sent_packets”, and their sent times in “outgoing_packets” and

received them by appending them to “incoming_queue”, which acts like a buffer, until the packets are extracted, decoded, and processed in a separate function: “process_packet()”.

- “While True” for infinite loop

3.2 Reliable Data Transfer via Unreliable Channel

We implemented reliability in transfer the following ways:

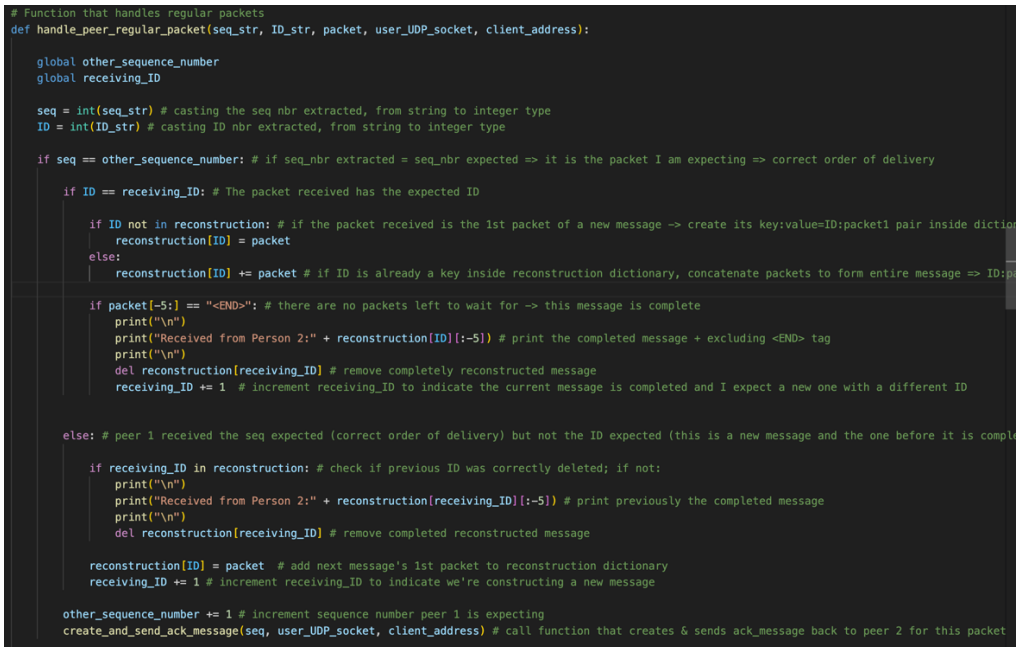
A) Ensurance of Order Packet Delivery

We settled for the implementation of unique sequence numbers as headers when sending packets which play a double role: correct reconstruction of the full message at receiver-end and correct order of delivery.

```
# Function that allows for sending messages in packets
def send_message(user_UDP_socket, client_address, user_TCP_socket):

    global my_sequence_number
    global sending_ID

    max_packet_size = 50 # initialize variable of max_packet_size of 1,000B

    while True: # infinite loop

        print("\n")
        message = input() # Input message

        if message == "<I WANT TO SEND A FILE>": # Handle sending files with TCP
            file_and_extension_name = input("Input the name of the file and its extension that you want to send: ")
            print("\n")
            send_TCP_file(client_address, file_and_extension_name)

        elif message == "": # if peer 1 pressed enter (did not write any message to be sent)
            print("Enter a valid message to send!")

        else: # Handle normal text messages

            message_packets = divide_message(message, max_packet_size) # divide the message into packets & store them in message_packets array

            for packet in message_packets: # loop over all the packets inside array

                packet_with_headers = str(my_sequence_number) + ";" + str(sending_ID) + ";" + packet # Add sequence_number & ID as headers to the packet I
                packet_with_headers = packet_with_headers.encode() # encode packet_with_headers
                user_UDP_socket.sendto(packet_with_headers,("127.0.0.1", 50010)) # send packet_with_headers to peer 2

                outgoing_packets[my_sequence_number] = packet_with_headers # insert ENCODED packet_with_headers into outgoing_packets
                sent_packets[my_sequence_number] = time.time() # insert in dictionary the time at which the packet was sent at key=sequence_number
                my_sequence_number += 1 # Increment sequence number
            sending_ID += 1 # increment message_ID so that the next message has a different ID
```

Figure 3: send_message() Function

Once a packet is received and stored at the receiver’s end in incoming_queue, the receiver gets the encoded packet, demultiplexes itand extracts the sequence number attached as a header. He then compares the extracted sequence number with the one he expects. If they match, it is an indicator of a correct of delivery of the packet and moves on to reconstruct the message it is part of.



```
# Function that handles regular packets
def handle_peer_regular_packet(seq_str, ID_str, packet, user_UDP_socket, client_address):

    global other_sequence_number
    global receiving_ID

    seq = int(seq_str) # casting the seq nbr extracted, from string to integer type
    ID = int(ID_str) # casting ID nbr extracted, from string to integer type

    if seq == other_sequence_number: # if seq_nbr extracted = seq_nbr expected => it is the packet I am expecting => correct order of delivery

        if ID == receiving_ID: # The packet received has the expected ID

            if ID not in reconstruction: # if the packet received is the 1st packet of a new message -> create its key:value=ID:packet1 pair inside dictio
                reconstruction[ID] = packet
            else:
                reconstruction[ID] += packet # if ID is already a key inside reconstruction dictionary, concatenate packets to form entire message => ID:p

            if packet[-5:] == "<END>": # there are no packets left to wait for -> this message is complete
                print("\n")
                print("Received from Person 2:" + reconstruction[ID][:-5]) # print the completed message + excluding <END> tag
                print("\n")
                del reconstruction[receiving_ID] # remove completely reconstructed message
                receiving_ID += 1  # increment receiving_ID to indicate the current message is completed and I expect a new one with a different ID


        else: # peer 1 received the seq expected (correct order of delivery) but not the ID expected (this is a new message and the one before it is compl

            if receiving_ID in reconstruction: # check if previous ID was correctly deleted; if not:
                print("\n")
                print("Received from Person 2:" + reconstruction[receiving_ID][:-5]) # print previously the completed message
                print("\n")
                del reconstruction[receiving_ID] # remove completed reconstructed message

            reconstruction[ID] = packet  # add next message's 1st packet to reconstruction dictionary
            receiving_ID += 1 # increment receiving_ID to indicate we're constructing a new message

        other_sequence_number += 1 # increment sequence number peer 1 is expecting
        create_and_send_ack_message(seq, user_UDP_socket, client_address) # call function that creates & sends ack_message back to peer 2 for this packet
```

Figure 4: Snippet of Code for Identifying Packets Received in Order using Sequence Numbers

B) Dealing with Out of Order Packet Delivery

We then used sequence numbers to identify out of order packets: if the sequence number extracted from the current packet was larger than the sequence number the receiver expects from the sender, then this indicates that the current packet is premature to the one needed to complete the message in the correct order. So, the extracted sequence number is added back to the packet as a header before adding the latter back in the incoming_queue for later processing, once the expected packet is processed beforehand.

C) Dealing with Duplicated Packet Delivery

When the sequence number extracted is smaller than the one the receiver is expecting, this is a clear indication that the current packet is just a duplication of an identical previously processed packet. We would opt to send an ACK message, again, to avoid the sender to retransmit a previously processed packet.

```
elif seq > other_sequence_number: #seq_nbr extracted > seq_nbr expected => packet delivered in Wrong Order
    pckt_with_headers = str(seq) + ";" +  str(ID) + ";" + packet # add corresponding headers back to the packet
    incoming_queue.put(pckt_with_headers) # insert the pckt_with_headers back at the tail of incoming_queue to be processed again later

else: # seq < other_sequence_number => duplicated packet! I will not print it but I will send an ACK so that it is NOT retransmitted again
    create_and_send_ack_message(seq, user_UDP_socket, client_address)
```

Figure 5: Snippet of Code for Identifying Packets Received out of Order or Duplicated using Sequence Numbers

D) Handling Missing Packets using ACKs & Timeout

We opted for the implementation of ACK exchanges with the aim to detect missing packets and request for their retransmission. Our code would create and send an ACK, corresponding to the processed packet's sequence number, back to the sender. We used a similar format to normal text messages, but replacing the sequence number header field we made by "ACK" string to differentiate between ACK messages and text messages. We then made a function to handle receiving ACK messages correctly. Finally, we made a thread function that always handles missing packets by checking whether a packet that is awaiting an ACK has passed the allocated Timeout. If Timeout is detected, the message is retransmitted with a new ACK.

```python
# Function that creates and sends ack message back to peer 2
def create_and_send_ack_message(seq_int, user_UDP_socket, client_address):
    ack_message = "ACK;0;" + str(seq_int) # create ACK to send back to peer 2
    ack_message = ack_message.encode() # encode ACK message
    user_UDP_socket.sendto(ack_message, client_address) # send ACK message via socket

# Function that handles ACKs
def handle_peer_ack_message(ack_nbr_str):
    ack_nbr = int(ack_nbr_str) # casting the ack nbr extracted from string to integer type
    if ack_nbr in sent_packets:
        del sent_packets[ack_nbr] # deleting the ack's corresponding value (timestamp) from the dictionary
    if ack_nbr in outgoing_packets:
        del outgoing_packets[ack_nbr] # remove & return the ack's corresponding message out of the outgoing_messages


# Function that handles missing packets
def handle_missing_packets(user_UDP_socket, client_address):

    while True: # infinite loop

        if len(sent_packets) != 0:

            for seq, timestamp in sent_packets.items(): # iterate over the dictionary's key:value pairs

                if time.time() - timestamp > ACK_timeout: # if the pair still exists in the dictionary for more than ACK_timeout -> retransmit the packet!

                    print("Timeout! Resending...")
                    print("\n")

                    message = outgoing_packets[seq] # remove & return the ENCODED packet from outgoing_queue

                    user_UDP_socket.sendto(message, client_address) # retransmit the packet to peer 2
                    create_and_send_ack_message(seq, user_UDP_socket, client_address) # send a new ACK to peer 2
                    sent_packets[seq] = time.time() # reset the sent packet's timestamp

        time.sleep(1) # Check for missing messages periodically every 1 second to save resources
```

Figure 6: Code Implementation of ACKs & Timeout

3.3 Testing under Unreliable Conditions (1)

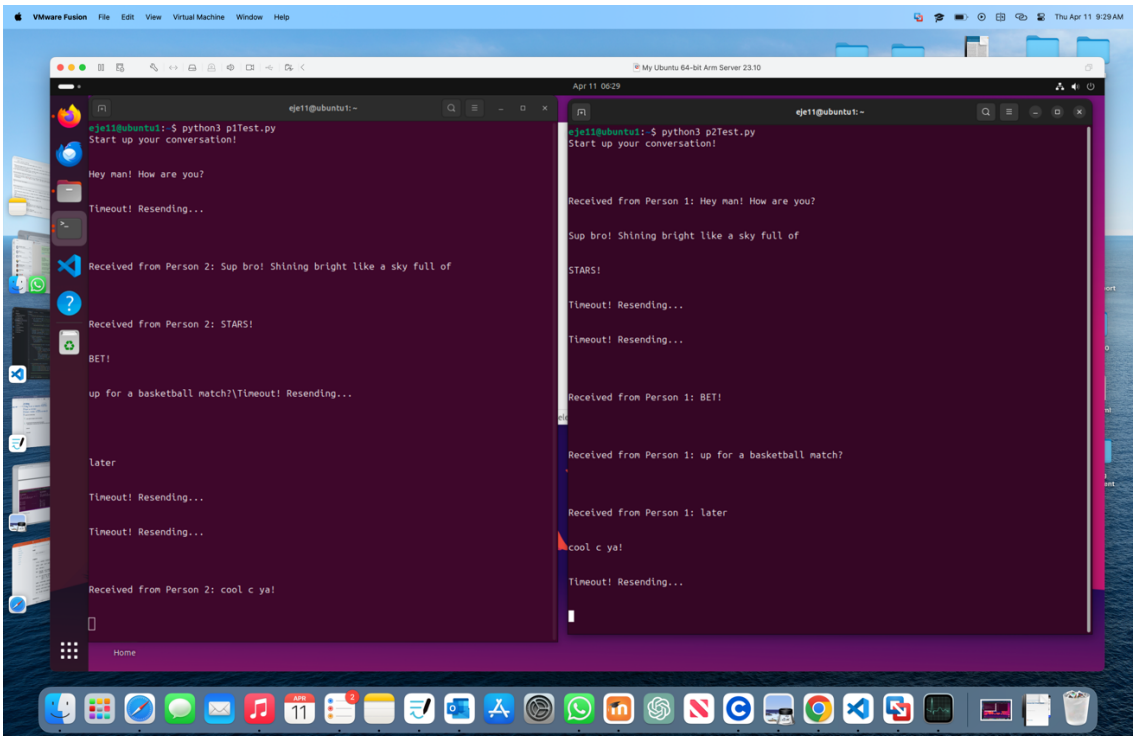


Figure 7: Delay of 6,000ms = 6s (UDP)

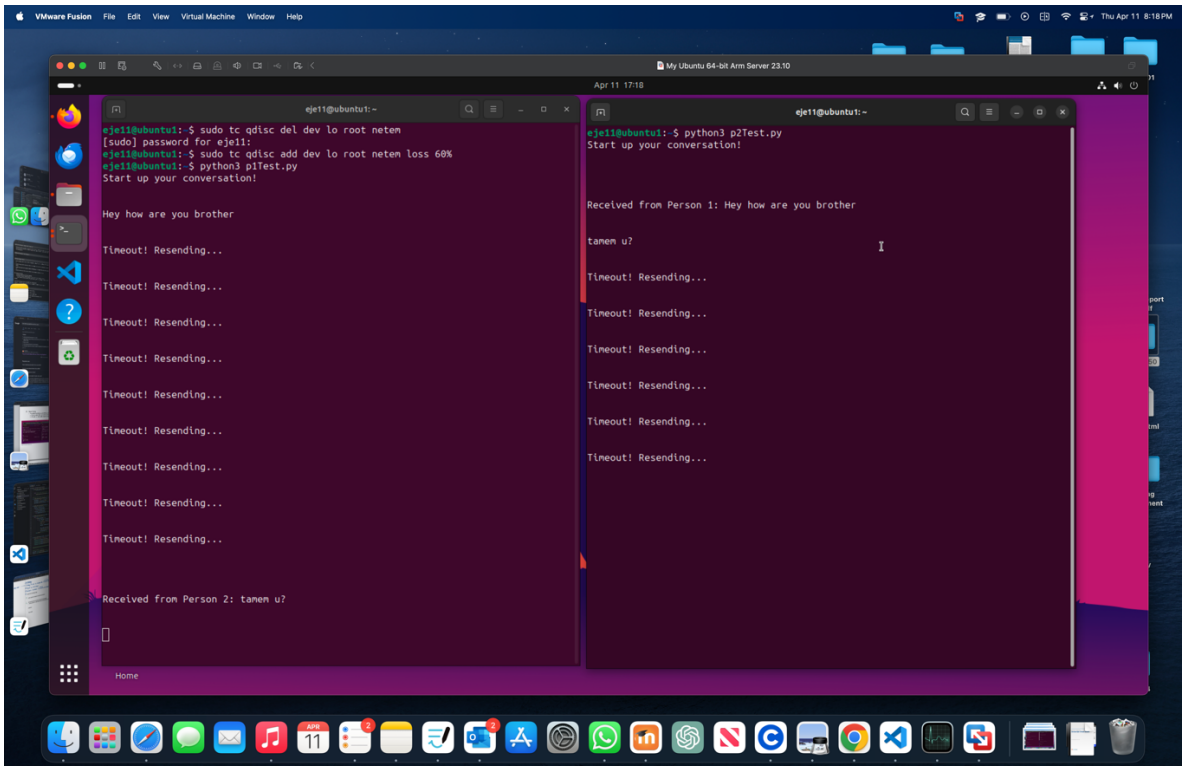


Figure 8: Packet Loss at 60% (UDP)

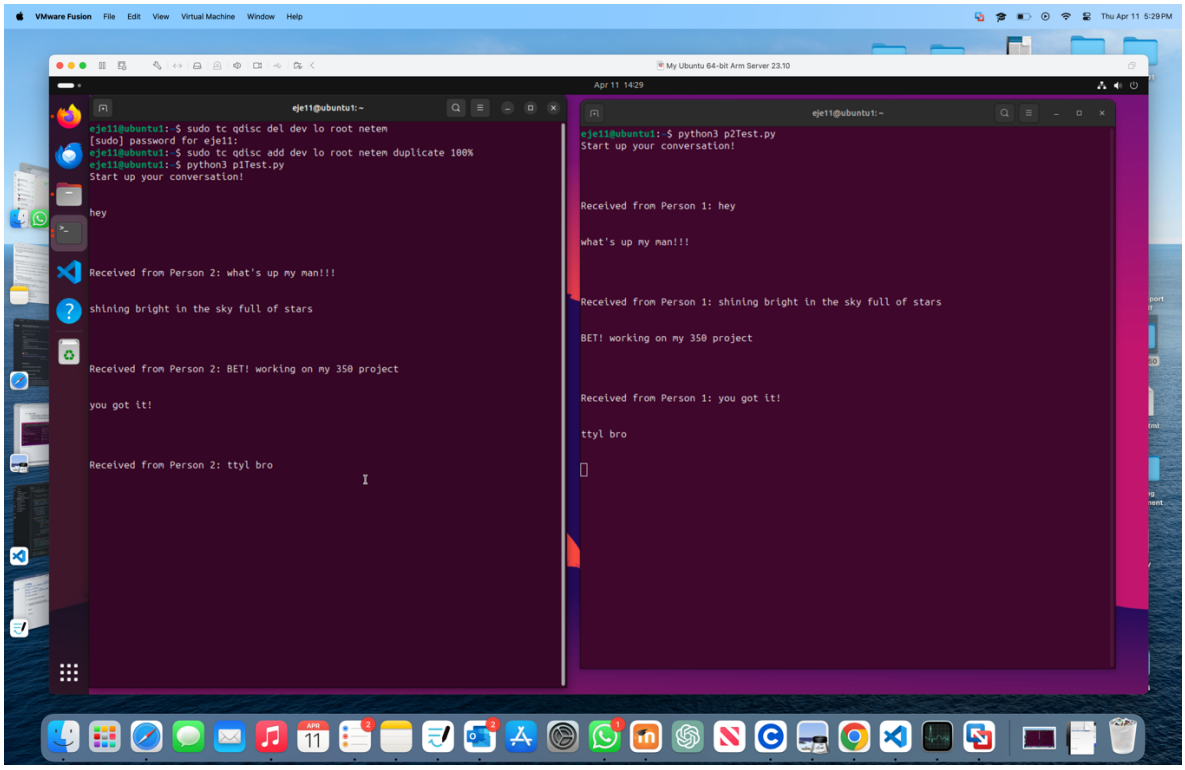


Figure 9: Duplication at 100% (UDP)

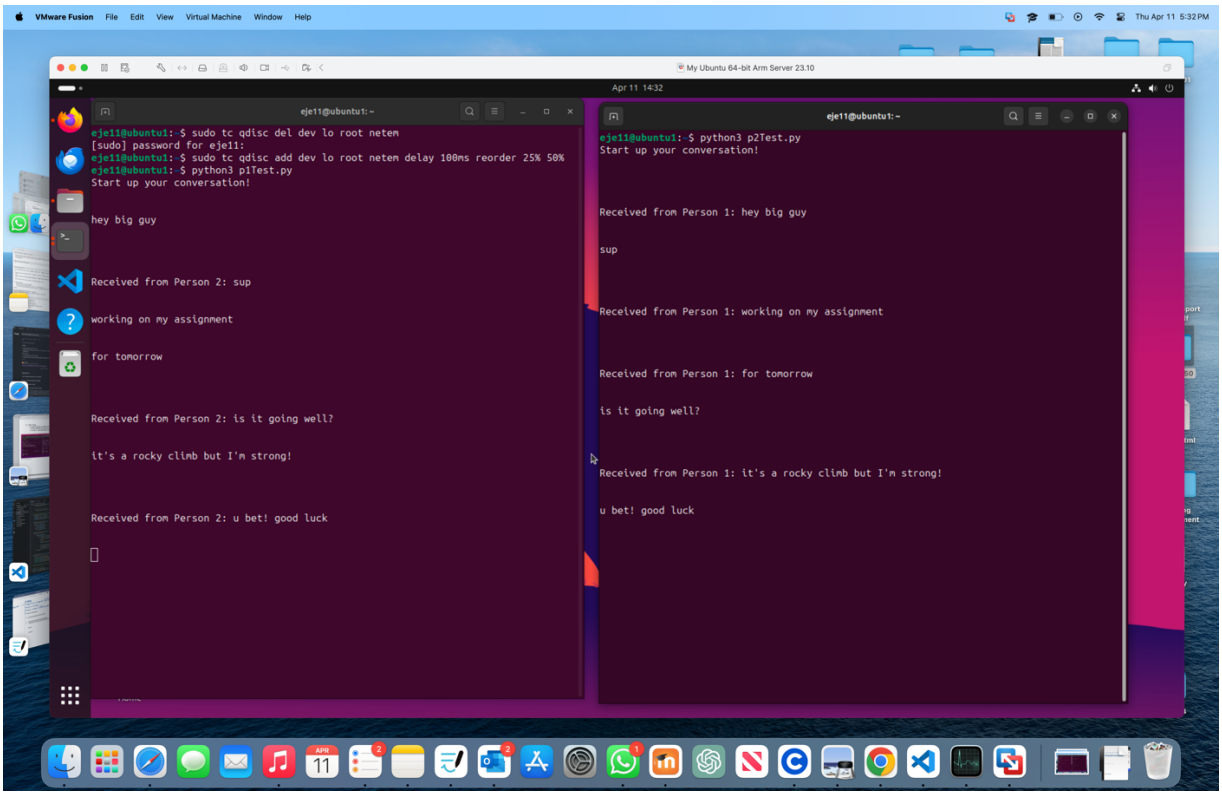


Figure 10: Packet Reordering (UDP)

04 Phase 3: Sending Files

4.1 Implementing TCP File Exchange System

To exchange a file on a TCP connection, we decided for the sender to input the message "<I WANT TO SEND A FILE>", which then requests the exact file name and extension. This calls send_TCP_file() function which sends the file through TCP. It also uses <END> tag to indicate end of message! On the other end, the receiver uses receive_TCP_file() which constantly listens for peers to connect to. After a connection is established, a file is opened, written on, and reconstructed, byte by byte.

4.2 Testing under Unreliable Conditions (2)

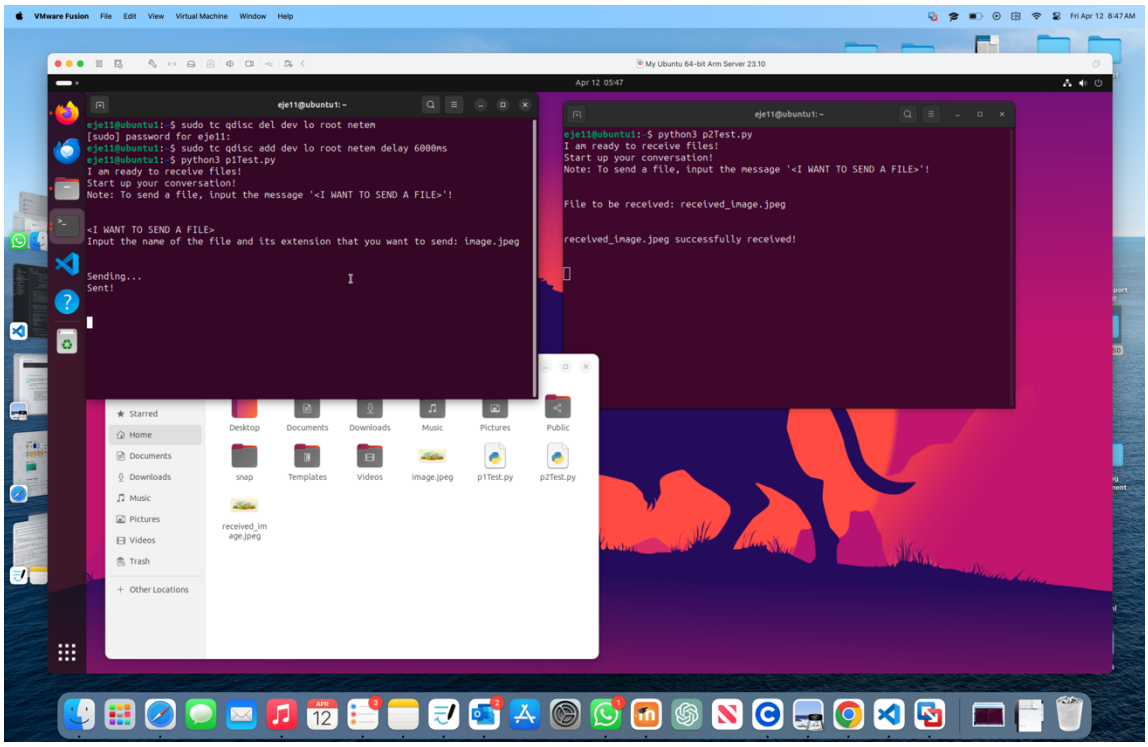


Figure 11: Delay of 6,000ms = 6s (TCP)

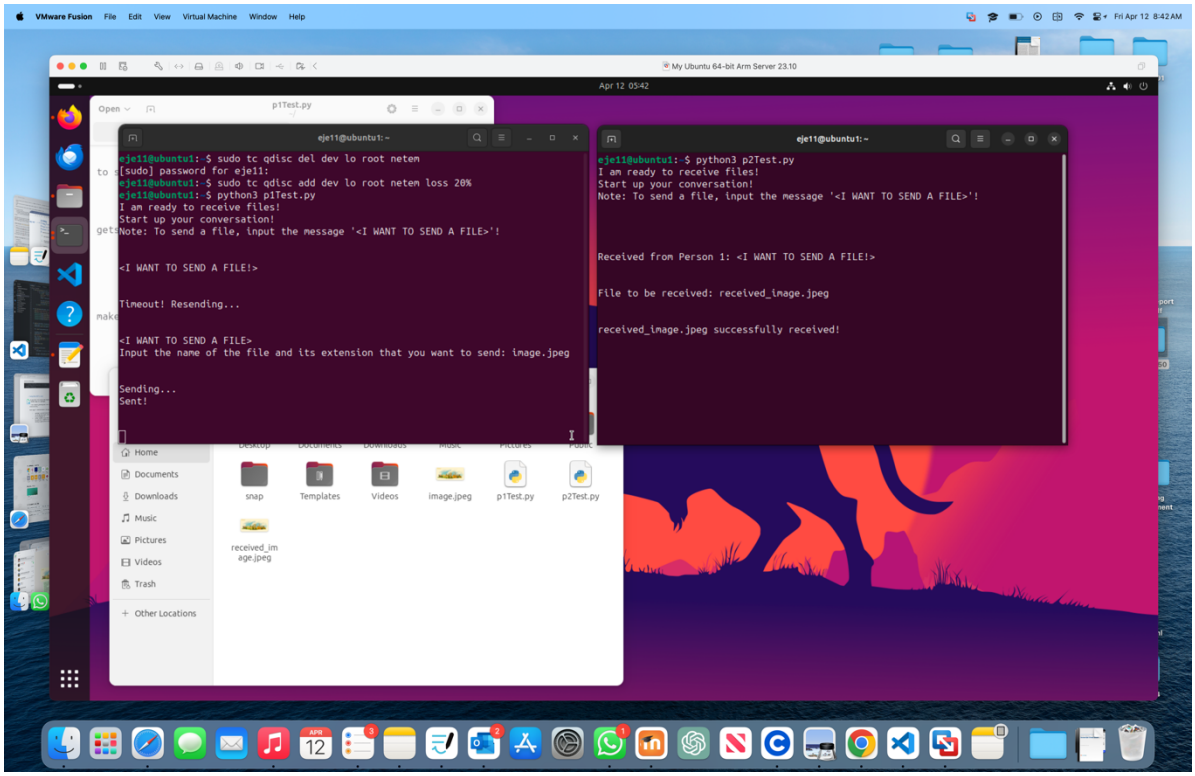


Figure 12: Packet Loss at 20% (TCP)

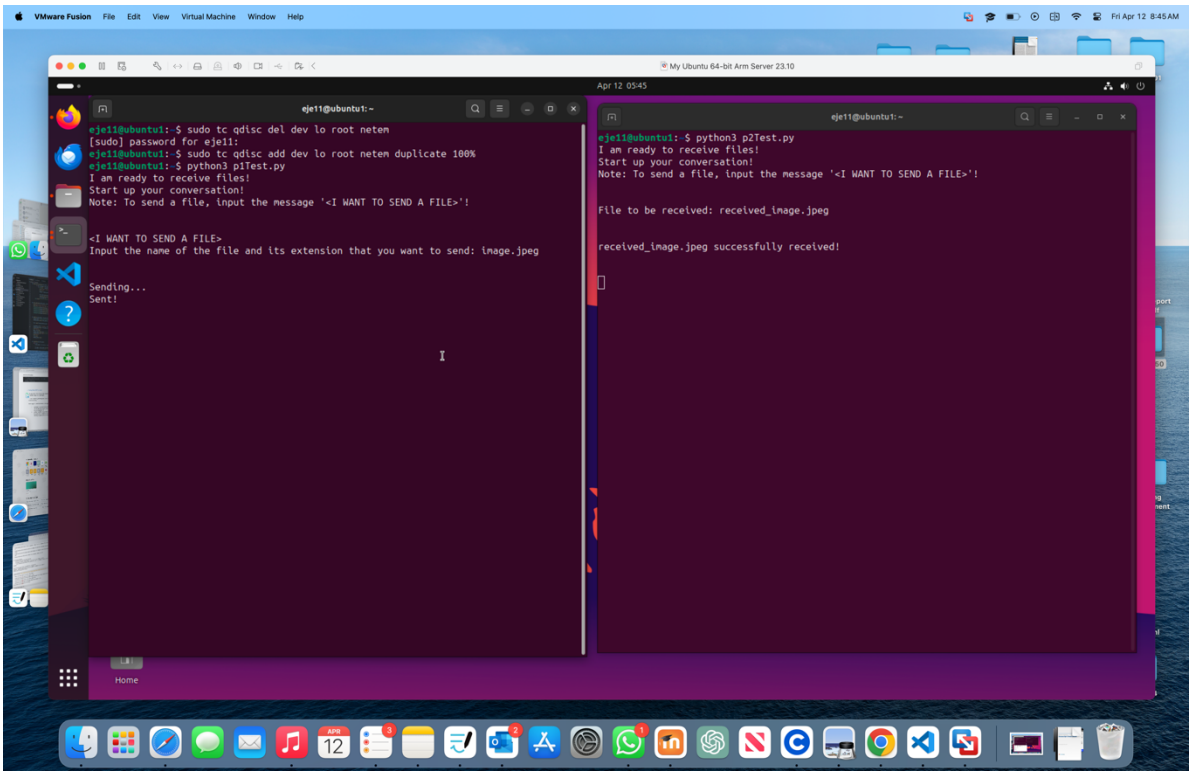


Figure 13: Duplication at 100% (TCP)

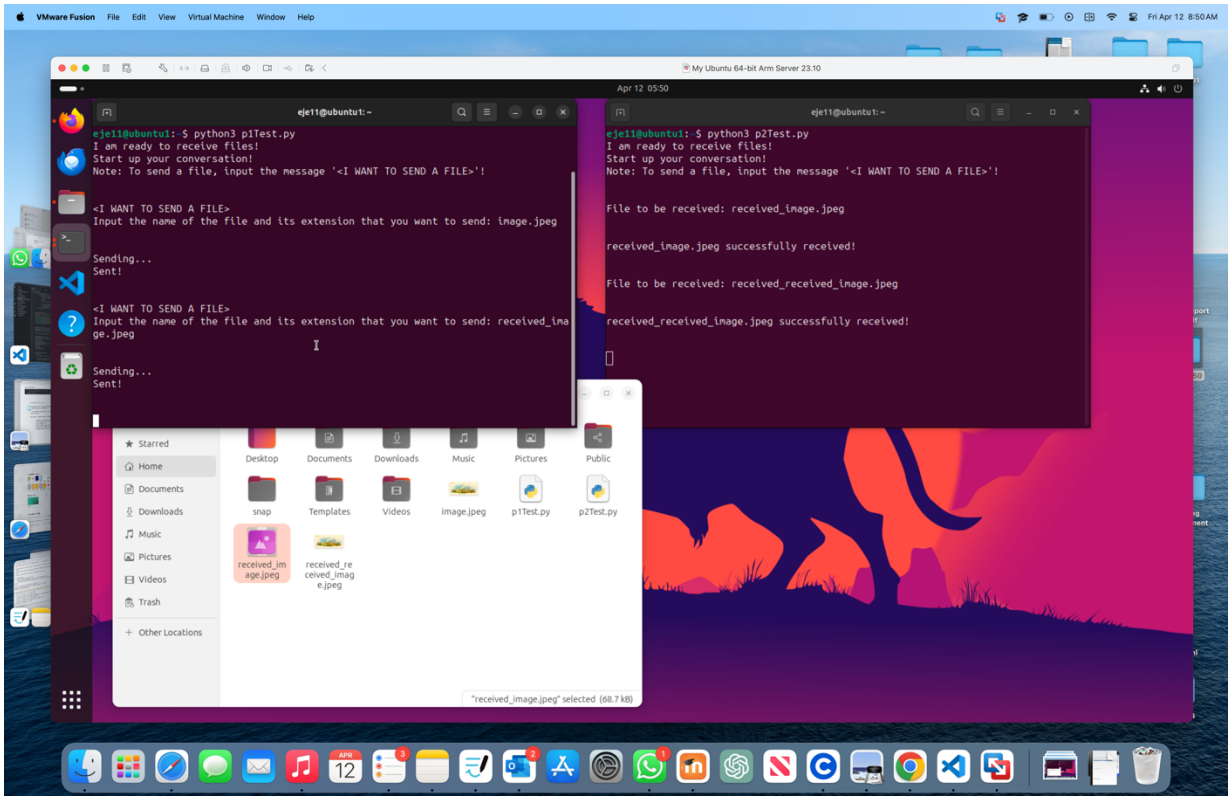


Figure 14: Packet Reordering (TCP)

05 Special Features in the Design (BONUS)

Our design supports many special features ranging from the GUI to the backend code such as:

Backend code:

1. Closing connection automatically after 30 seconds of inactivity (note that we can change the time of inactivity). Implementing this feature required new global variables and flags to identify whether either of the peers are messaging or receiving or being idle. (*However, this feature was not fully tested using Netem).*

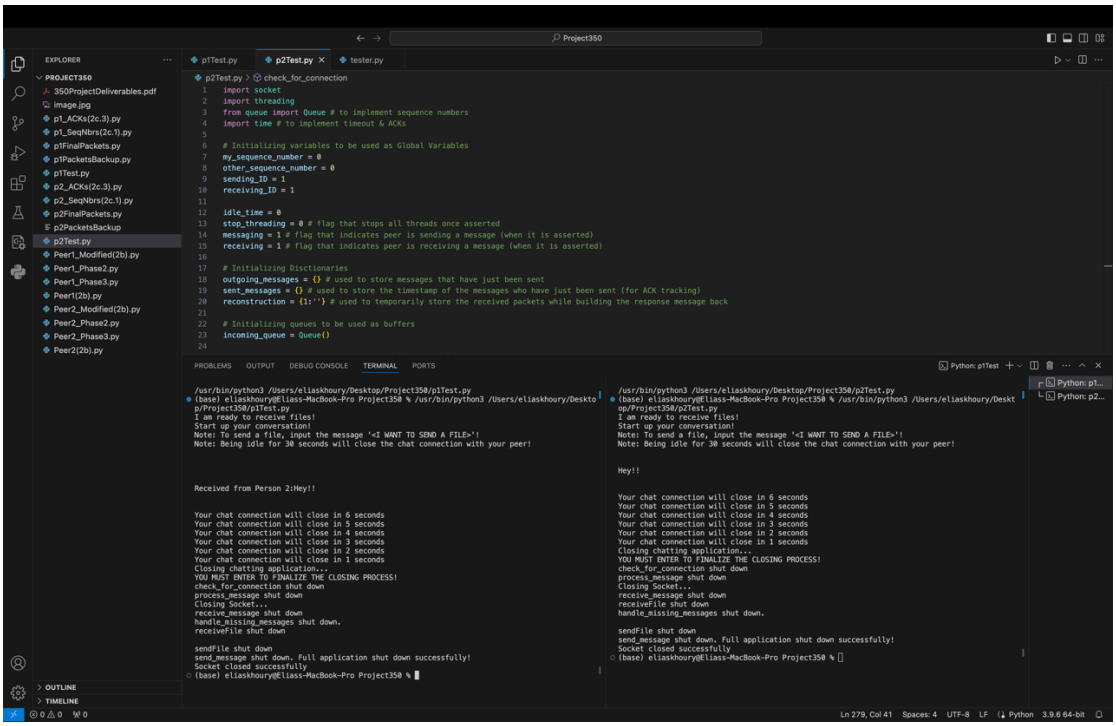


Figure 15: Closing Connection Output

2. Implemented Buffer at receiver end. This helps store the packets sent at the receiver's end to directly transmit them; allows for bursty data. This also differentiates our project from other teams' who made the stop-and-wait mechanism using just 1 bit. This also helps receiver decrease packet drop rate since the buffer is storing packets instead of simply dropping and constantly asking for their retransmission.

3. We first implemented our code in a way that sent the message as 1 packet. Then, we decided to divide the message into packets of 50 bytes each. (This number was selected after some research). That is for the following reasons:
    a. Faster Transmission: smaller packets tend to be transferred more quickly.
    b. Better Reliability: in case of packet loss or errors during transmission, smaller packets are easier to retransmit.

   For messages to be reconstructed at receiver end, we introduced new variables related to a message's ID as well as an <END> tag in the end of the LAST packet. This helps completely reassemble message's packets in order and without loss!

Frontend GUI:

1. A timestamp for each message sent and received.
2. A clock to keep track of the time.
3. A "Typing…" indicator.

4. Error notification if one of the peers is offline. Note that no error handling was implemented, and that it's the peer's responsibility to check for connectivity and to make sure that the network is working properly.

06 Phase 4: Graphical User Interface

In our code we implemented a GUI using the Tkinter library in python. We ran it using 2 terminals: One for each peer.

Our GUI supports the following:

5. Sending messages between two peers

6. File sharing between two peers
7. A timestamp for each message sent and received
8. A clock to keep track of the time
9. A "Typing…" indicator
10. Error notification if one of the peers is offline

The code consists of a few functions that will be thoroughly explained throughout this report.

NOTE: When (1/2/3/4) is encountered, please refer to the notes below the functions for a brief clarification.

_ _init_ _ which takes a title, peer address, peer port, and listen port as input arguments. This function sets the overall connection and arranges the page with all its features. Additionally, it links the icons of the send button for messages and for file sharing. Effectively, this snippet contains the design of the page, the configuration of the buttons, and the bindings of the sockets.

accept_connections that operates in a server-client fashion and allows the "server" side peer to listen to and receive messages on a specific port.

receive_messages(1-4) takes as input argument the client socket in order to connect to it and send through it a maximum of 1024 bytes. This function also supports timestamps for received messages, and mentions the functions *receive_file* and *update_typing_status* as a part of receiving data from the other peer. All in all, this method decodes the message sent, displays it along with the current time in the chat box, and raises an error if its not received.

send_message(2-3) after establishing a connection, it prompts the user to enter a message in the text box and send it to the other peer also having an open console, by clicking on the send button. We modified this function to include the specific time messages were sent and received.

send_file(1-2) (very similar to *send_message*) it allows the user to browse through the files of his/her own computer and attach the desired file to send it using this line file_path = filedialog.askopenfilename(). It also notifies the user if their file was uploaded or not.

receive_file(4) which takes as input arguments client_socket and file name for it to send the specific file over the specific socket. This function displays timestamps as well as a message in the peer's window indicating that the file was received as **"Peer: File Received: {file_name}".** Otherwise, the function raises an error.

update_typing_status, typing_indicator, send_typing_indicator these three functions are responsible for displaying the "Typing…" message above the chat box to inform the first peer that the second is typing. Respectively, the functions check for the status of the second peer and update the title of the window accordingly, poll for the messages displayed in the chat box and verifies whether or not they correspond to the current state of the "typing flag"(if already on, keep it. Else, turn it on. And vice versa), as well as establish a connection to inform the second peer of the changes made and decisions taken by the first two functions. This function also raises an assertion as to the availability of the 2nd peer.

run Lastly, this method is responsible for executing the code starting a new thread after the detection and acceptance of a new connection. Specifically, *self.window.mainloop()* starts the Tkinter loop, which means that it permits the code to constantly sense any changes such as button presses, typing messages,… and updates the GUI accordingly. It also keeps the floating docks of the GUI awake.

(1) base64 encoding is commonly used to encode binary data, such as images, audio, or other binary files, into a text-based human-readable format.

(2) This function asserts the availability of the other peer. If not, it raises a customizable error message "ERROR!!! Peer Unavailable: Please request connection".

(3) By setting event=None, the method can be called with or without an event parameter, even without being triggered. If no event is provided when calling the method, it will default to None. This allows the method to be used in different contexts. Its also like "while True".

(4) It's important to note that this does not necessarily mean that the maximum file size that can be received is 1024 bytes. Instead, it means that the data is received in chunks of up to 1024 bytes at a time. If the file being transferred is larger than 1024 bytes, multiple calls to recv will be made to receive the entire file.

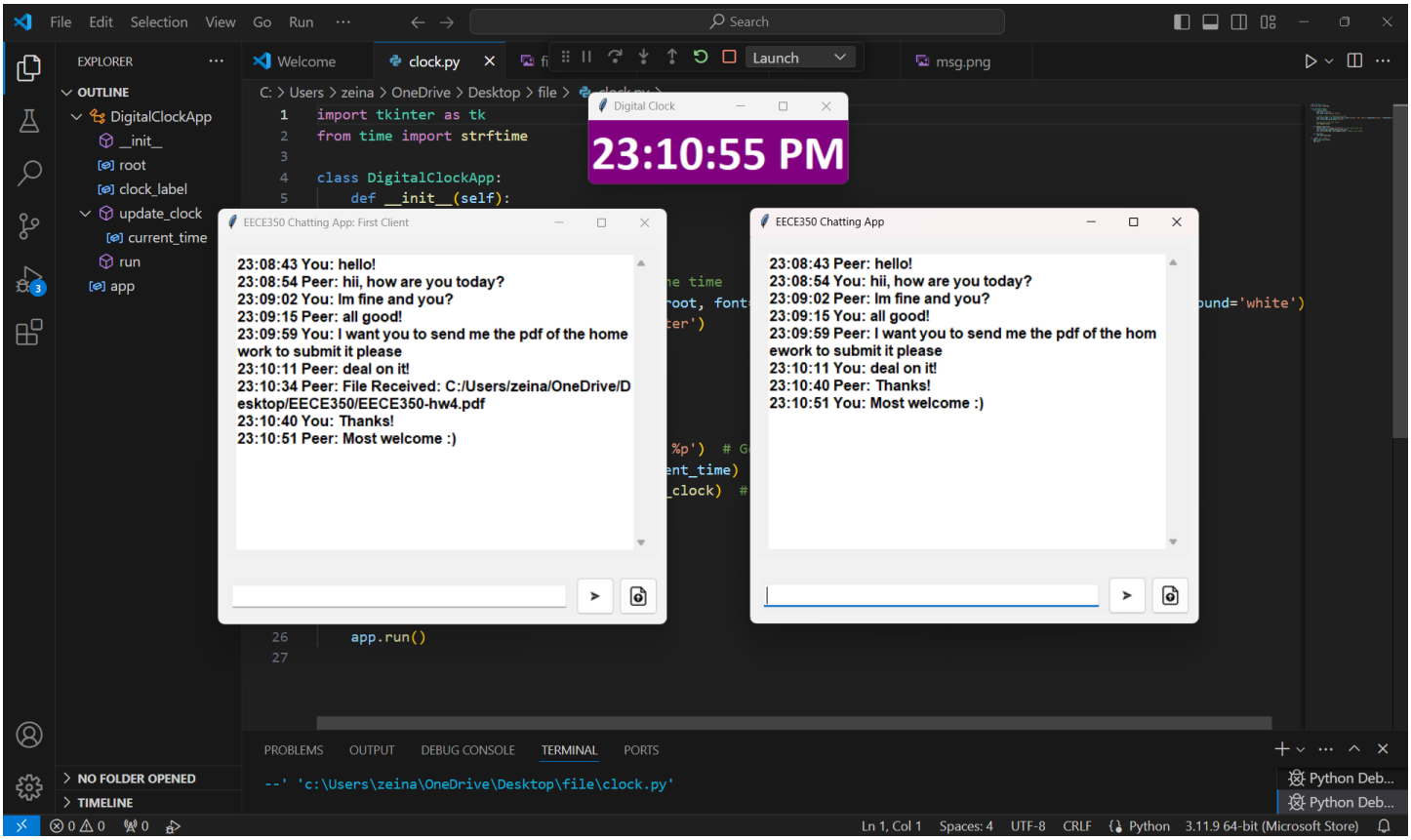


Figure 16: GUI Demo

07 Improvement

Upcoming updates and further improvements to our app include the following:

1. In our current implementation, the buffer (incoming_queue) is of unlimited size. Our approach considered ideal conditions, whereas a more realistic buffer size should be limited to an upper bound (preferably 4 messages in buffer at receiver end). Accounting for the responsiveness of the application, smaller buffers may result in quicker display of incoming messages but could increase the probability of message loss if the buffer overflows. So, after careful research, an optimal buffer size would be calculated as such:
    a. The average length of a message exchange between two peers is 200 bytes. So, we decided to divide the packets into sizes of 50 bytes for faster transfer. The following adds up to 4 packets per message.
    b. Say we would like our buffer at the receiver end to store up to 5 messages from the sender. This is equivalent to 5*4 = 20 packets: so limiting the buffer to 20 packets. Each packet is 50 bytes => 50*20 = 1,000 Bytes buffer
2. The sequence number in our code can go up to infinity (2^{64}-1=18446744073709551615). However, a sequence number should have a finite range because if sequence numbers were allowed to grow to infinity, they would eventually reach the maximum value for their field size and wrap around to zero. This wraparound can lead to confusion and misinterpretation of sequence numbers, causing errors in data delivery and acknowledgment.
3. The ID number must also converge due to network limitations and similar reasons as the sequence numbers.
4. Testing the connection closing mechanism due to stretched idle time.

08 Challenges

- Researching about the GUI:
  - Due to our lack of experience in the field of user interfaces, one major challenge was the ongoing research for a suitable software to use and implement in our chatting app. We first thought about using PyQt Designer Lab, which is an advanced software with rich libraries, we also tried Html/Css as well as Node.Js. The simplest to implement, understand and learn in such short notice was tkinter, so that's what we settled on at the end.
- Implementing the logic of phase 2 in code
- Thinking about edge cases and accounting for all possible scenarios of unusual network behavior
- Lots of debugging required to solve issues in our code once tested under traffic conditions and jammed networks (ensured by Netem)
- Finding a Virtual machine environment supported by both MAC and Windows, and learning how to use it as well as installing Ubuntu and activating Netem

09 Conclusion

In summary, this assignment provided valuable insight into the design of a peer-to-peer network chatting app in achieving the desired performance characteristics. All in all, our project included a backend code containing many functions and features to accomplish sharing files over a TCP connection and sending messages using a modified UDP protocol that supported extra reliability. Additionally, we used tkinter to implement a graphical user interface GUI as our frontend part. Although we faced many challenges, we managed to create a fully functional chatting app tested under traffic conditions and network jamming. Effectively, messages and files are delivered in order and without any losses.