Design File

Title and Authors

- This file is for Phase 4 of the Network Design group project
- Group Members:
    - Alexander Nunez Pepen
    - Md Zahidul Islam
    - Pablo Ruiz
    - Joseph Ayoka

Purpose of the Phase

In phase 4, we are implementing the RDT 2.2 protocol using the UDP server and client process. In Phase 2 we achieved the ability to transfer a (.BMP) image file from the UDP client to the UDP server. RDT 2.2 stands for reliable data transfer protocol version 2.2. The RDT 2.2 protocol can be modeled by two finite state machines (FSM) diagrams. The sender's FSM has a lot more states and transitions when compared to the receiver side FSM diagram. When the sender (client) wants to communicate with the receiver (server), the client creates a packet with a header that includes the packet's sequence number and checksum. Once the packet is sent (using UDP), the client waits for an ACK from the server before sending the next packet. On the server side, if the sequence number is correct and the checksum computed correctly (no corruption), then the server extracts the packet and sends an "ACK" to the client. The server then goes back to waiting for the next packet. If the packet is corrupt (checksum computes

incorrectly) or has an unexpected sequence number, the server discards the packet and waits for the client to resend the packet. An ACK response from the server can be lost, failed to send, or corrupt. So if the client receives a packet that is supposed to be the server's ACK response but it's been corrupted and the client does not know that. So to combat this the client side resends the data packet repeatedly until it gets a correct ACK response from the server.

RDT 2.2 only sends ACKs because it is a NAK-less protocol. Meaning that the server does not send negative acknowledgments (NAKs) when a packet is corrupted. Instead, RDT 2.2 sends back an ACK to the sender but with the corrupted packet's expected sequence number in the ACK response. This sequence number demonstrates to the client that something went wrong with the recently sent packet (essentially a NAK).

The RDT 2.2 protocol was implemented using the code developed for phase 2. Changes that were made for RDT 2.2 include a function to create/compute the checksum, include a method to handle ACKs (server side and client side), have the client repeatedly send a packet until it is ACKed, and functions to purposely corrupt packets.

For RDT 3.0, we had to build off of the Phase 3 RDT 2.2 code to include a way to handle packet loss by using a countdown timer. The count-down timer upgrades RDT 2.2 to RDT 3.0. The countdown timer can account for packet loss and ACK loss. The

Real-Time Transport (RTT) is set to calculate a timeout threshold. If a packet or ACK is not received during the RTT time period then the client will consider the packet or ACK as lost and resend the packet to the server. This function generates RTT values based on message exchanges. This function updates the real-time timeout threshold by actively doing calculations and updating the RTT value as packets are sent.

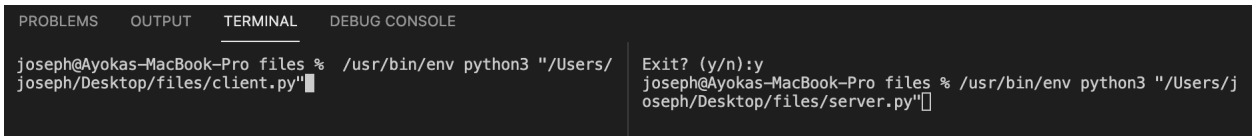
Code Explanation

Execution Example

Procedure for Communication between client and server are described below.

Step 1:

Run Server.py and Client.py in two separate terminals as shown below



Step 2:

In the Server.py terminal, press enter to initiate the server process

```
joseph@Ayokas-MacBook-Pro files % /usr/bin/env python3 "/Users/j
oseph/Desktop/files/server.py"

Awaiting Connection...
```

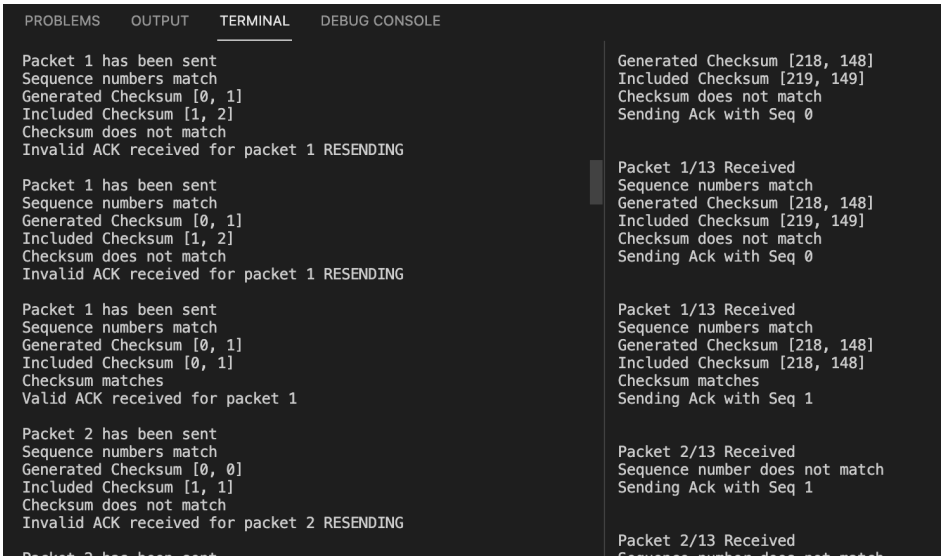
Step 3:

In the Client.py terminal enter the name of image to send and name that the new image will be saved as.

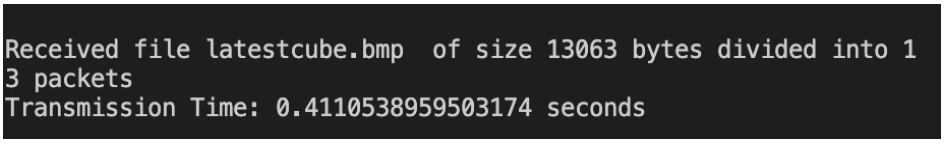
```
joseph@Ayokas-MacBook-Pro files %  /usr/bin/env python3 "/Users/
joseph/Desktop/files/client.py"
Enter the name of the image you want to send (include file type
extension): meme.png

The file meme.png contains 24725 bytes divided into 25, 1024 byt
e packets

Enter the name the image will be saved under (include file type
extension): newmeme.png
```

Results:

The result below shows that the client and server behavior works as how it was intended to. As shown, the client will keep sending the packet until the ACK with check sum and sequence number match.

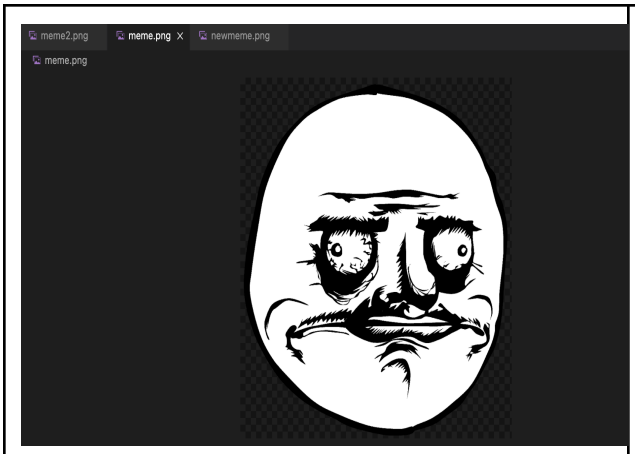


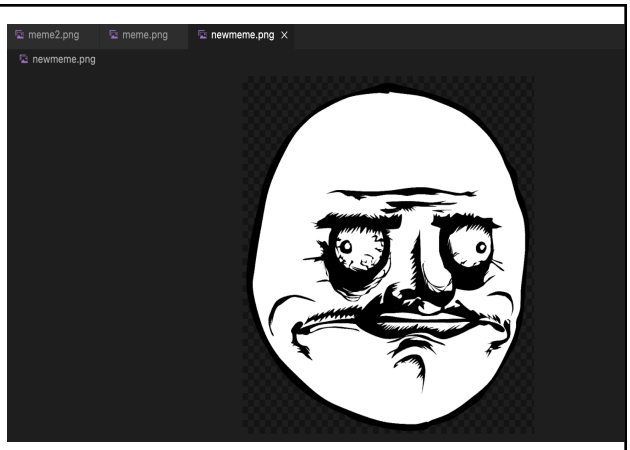
When all the packets are received successfully without errors a file will be created and would match the sending image to signify lack of data loss.

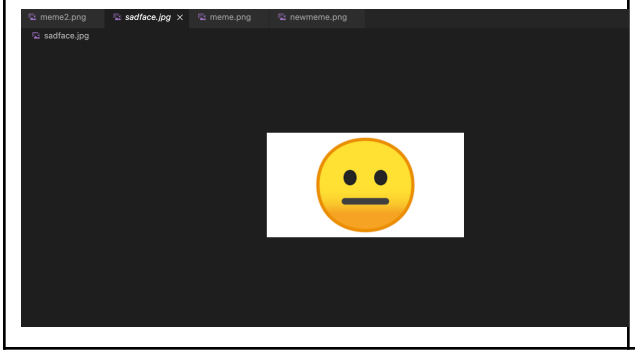


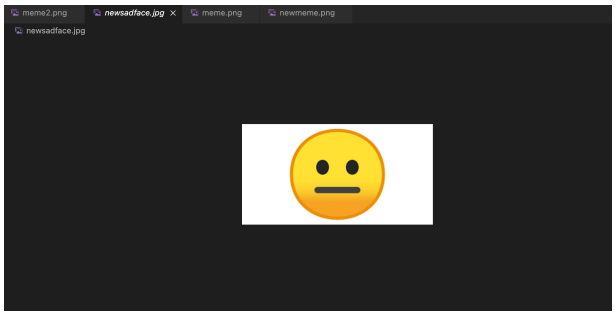
The table below shows the original sample PNG image been sent and what was received back on two different occasions.

Original PNG file:	**Received PNG file:**



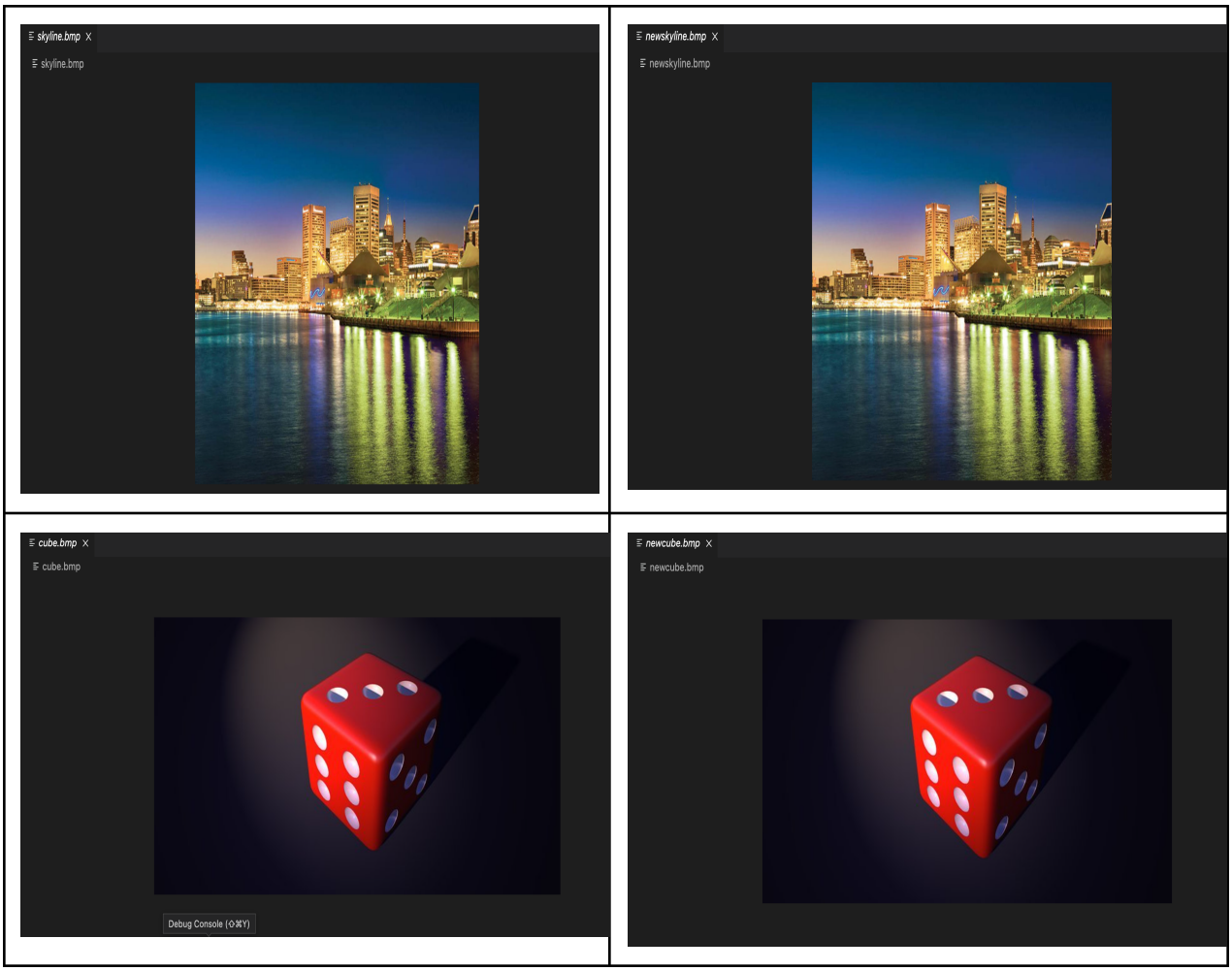






Test result using a different file format.

Original BMP file:	**Received BMP file:**

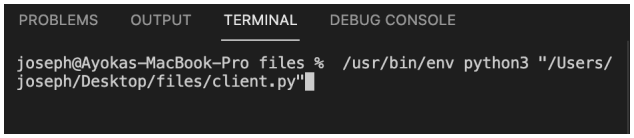


Execution Example with GUI

Procedure for Communication between client and server using the GUI are described and explained below.

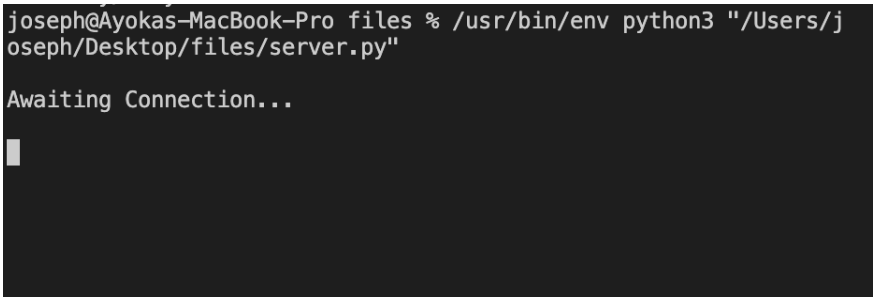
Step 1:

Run Server.py in a terminal as shown below



Step 2:

In the Server.py terminal, press enter to initiate the server process



Step 3:

In this step we will run the Client.py which contains the code for the GUI, the resulting interface will be shown as below.

Step 4

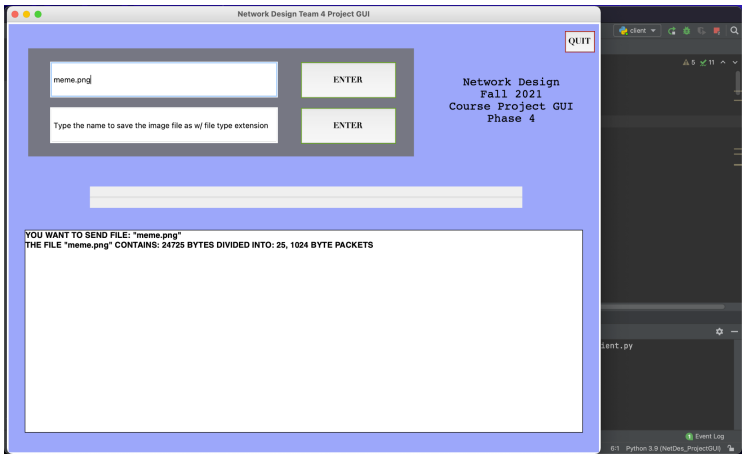
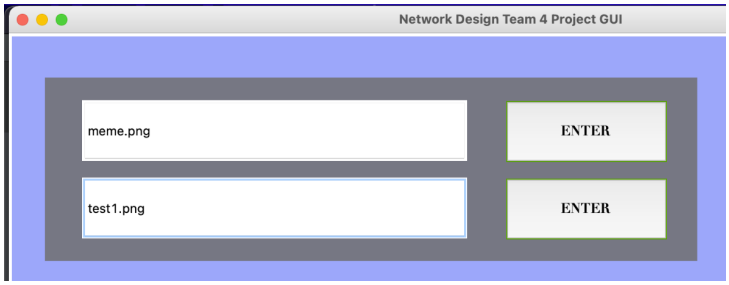


Figure 1

Figure 1 above shows that the interface of the GUI provides a space to input the name of the file to save along side the file extension and also a space to input the image name to be saved as. Once the name of file is declared the GUI shows information regarding how the bytes of the image will be divided into number of packets.

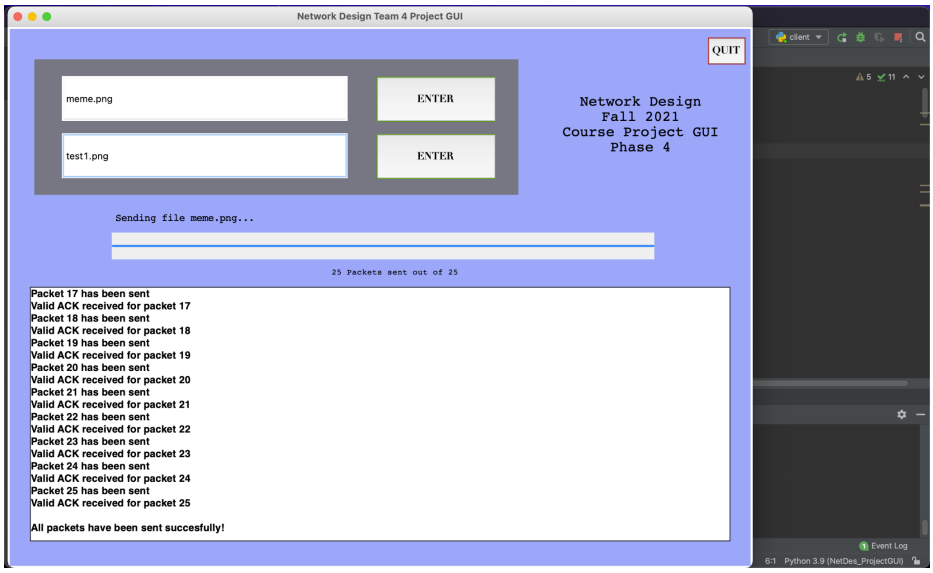


The image name to be saved as is entered and the process begins.

Results:

The GUI shows a comprehensive progress bar which updates by 1 as each packet is sent and a valid ACK is received in return.

The GUI was only built on the client process so it shows the progress as packets are being sent and valid ACK for the corresponding packets are received. Once the process finishes we exit the GUI and find the new saved file in our directory.



Original PNG file: *(meme.png)*	**Received PNG file:** *(test1.png)*