Design File

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Title and Authors

* This is for Phase 3 of the Network Design group project
* Group Members:
  + Alexander Nunez Pepen
  + Md Zahidul Islam
  + Pablo Ruiz
  + Joseph Ayoka

Purpose of the Phase

In phase 3, 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.

Code Explanation

Code for client.py:

from socket import \*

from copy import copy

from packet import packetizeBinary

from time import \*

from rdt22 import \*​

# TImeout threshold for socket operations in secs

SOCK\_TIMEOUT = 10​

# Obtain the name of the local host

serverName = gethostname()

# Declare the port number of the server that the client will be communicating with

serverPort = 12000​

# Assign a socket to the client using the arguments for a IPV4 address (AF\_INET) and UDP protocol (SOCK\_DGRAM)

clientSocket = socket(AF\_INET, SOCK\_DGRAM)

# Set timeout value for the socket operations

clientSocket.settimeout(SOCK\_TIMEOUT)

# Rquest input from user for image to send

image = input('Enter the name of the image you want to send (include file type extension): ')​

# Open and packetize the image message

pktList = packetizeBinary(image, 1024)

# Request input from user for image to send

imageSave = input('\nEnter the name the image will be saved under (include file type extension): ')

​# The header contains the number of packets in the message and its name separated by a space

header = str(len(pktList)) + ' ' + imageSave

# Send the number of packets and the name of the file as the first message so that the server knows when the transmission has ended

pktList.insert(0, bytearray(header, 'utf-8'))

# Loop over the list of packets and send them one by one in order

for i in range(len(pktList)):

# Generate a sequence number for the given package

seq = i%2

# Add the sequence number and a checksum to the package

makeRDT22(pktList[i], seq)

​ # Continue resending package until matching uncorrupted ACK has been received

while (True):

​ # Randomly corrupt the checksum or some packages

out = randomCorrupt(copy(pktList[i]))

​ # Send out the packet

clientSocket.sendto(bytearray(out), (serverName, serverPort))

​ print('Packet ' + str(i) + ' has been sent')

​ # Wait until ack is receieved

ack = clientSocket.recv(2048)

​

# Validate the ack, send next packet valid, else resend current packet

if checkRDT22(ack, seq):

print('Valid ACK received for packet ' + str(i) + '\n')

break

​ print('Invalid ACK received for packet ' + str(i) + ' RESENDING\n')

​​print('\nAll packets have been sent succesfully!')

# Close the client's socket

clientSocket.close()

The code for the client is the same as the code from phase 2 except for the code after line 40 (for loop). The for loop does a number of loops based on the number of packets generated using the *packetizeBinary* function from “packet.py”. On each loop iteration, a sequence number is computed using i modulo 2 (i is the loop iterator). This will give you either a 0 or a 1 for the sequence number of the packet being assembled. After the sequence number is computed, the *makeRDT22* function is called and the sequence number with the payload of the packet are passed in. The for loop has an embedded while true loop. This loop actively corrupts the assembled packets and sends them to the server. The client then waits in the while loop until an ACK is received from the server. The received ACK from the server is then checked for corruption by passing the packet into the *checkRDT22* function. This function computes the checksum and confirms the sequence number is correct. IF everything is correct then the function breaks and the socket is closed. IF the ACK packet was corrupted then the client packet is resent to the server and the process repeats itself.

Code for server.py

from socket import \*

from packet import binaryFromPackets, getHeaderParts

from rdt22 import \*

from time import \*

from copy import copy

pktList = []

# Declare the port that will be assigned to the server

serverPort = 12000

# Assign a socket to the client using the arguments for a IPV4 address (AF\_INET) and UDP protocol (SOCK\_DGRAM)

serverSocket = socket(AF\_INET, SOCK\_DGRAM)

# Bind the socket to the chosen port on the local host

serverSocket.bind(('', serverPort))

while True:

print('\nAwaiting Connection...\n')

header = validatePkt(serverSocket, 0)

# Start timer

timer = [time(), 0]

# Unpack header

numPackets, name = getHeaderParts(header)

# Loop so that the server checks for all the packets in the message

for i in range(1,numPackets+1):

# Generate correct sequence number for current packet

seq = i%2

# Check for received message and loop until an in-order uncorrupted message has been received and Acked

pkt = validatePkt(serverSocket, seq, i, numPackets)

# Add a copy of the pacekt to the list of packets

pktList.append(copy(pkt))

# Stop Timer

timer[1] = time()

# Reassemble the file from packets

data = binaryFromPackets(pktList)

pktList.clear()

# Print information about the received file

print('\nReceived file ' + name + ' of size ' + str(len(data)) + ' bytes divided into ' + str(numPackets) + ' packets')

print('Transmission Time: ' + str(timer[1] - timer[0]) + ' seconds')

# Save file

newFile = open(name, 'wb')

newFile.write(data)

print('\nSaved Succesfully!')

# Ask for permission to continue

while True:

done = input('\nExit? (y/n):')

if done == 'y' or done =='n':

break

print('Invalid Input:')

if done == 'y':

Break

The code for this server is the same as the code for the Phase 2 server up until the While loop. Inside the while loop needed functionality for RDT 2.2 was added to satisfy the FSM. Inside the while True loop, the received packet’s header is passed into the *validatePkt()* function. This function waits for a valid packet and is in charge of sending an ACK to the client. The received packet header is broken up into its separate parts (Total number of packets, Sequence number, checksum). This is done using the *getHeaderParts* function. Within the while loop, there is a for loop that iterates based on the range from 1 to the number of packets received plus 1. A sequence number is then generated for the current packet using i modulo 2 (i is the loop iterator). The loop then checks for a received message and then loops until an uncorrupted message that is in order is received and ACKed. This is done using the *validatePkt* function (found in rdt22.py). If the packet is validated then it is appended into a list (*pktList*). This list holds the received file packets in order. The list is then converted into a binary variable (*data*) using the function *binaryFromPackets* from “packet.py”. The converted binary is used to open the transmitted image file. The transmitted image file is saved on the server side under a user-specified name (transmitted in the first packet header) using the *open* python function. The .write constructor is used to build the image file using the transmitted binary data. This causes the image file to be saved onto the server’s side.

Code for packet.py

import math

import copy

def packetizeBinary(filePath, packetSize):

# Tested with png, jpg, and bmp images

packet = []

packetList = []

# Open file in reading and binary mode using the filePath

with open(filePath, 'rb') as image:

# Convert into an array of bytes that can be easily manipulated

data = bytearray(image.read())

# Loop one time per packet and round up because the packets can't be fractional

for i in range(math.ceil(len(data)/packetSize)):

# Define start and end of the bytes to be put into packet

start = i\*packetSize

end = i\*packetSize + packetSize

# If the index is larger than the length of the array, just include from the start point until the end of the array

if end < len(data):

# Append the bytes into a list which will represent one packet

packet.extend(data[start : end])

else:

packet.extend(data[start :])

# Append each packet into a list where each member is a packet

packetList.append(copy.copy(packet))

# Clear the packet list so that it can be filled with the next 1024 bytes in the next iteration

packet.clear()

# Print information about the packets

print('\nThe file ' + filePath + ' contains ' + str(len(data)) + ' bytes divided into ' + str(len(packetList)) + ', ' + str(packetSize) + ' byte packets')

return packetList

The code above represents the *packetizeBinary* function from “packet.py”. This function is the same as phase 2.

def binaryFromPackets(packetList):

data = bytearray()

for i in packetList:

data.extend(i)

return data

The code above represents the binaryFromPackets function. The purpose of this function is to take in a List full of the payload from many transmitted packets (the header was removed from these packets). It converts the List of packets from being in bytearray format to being in Binary.

def getHeaderParts(header):

# Convert the header to unicode

header = header.decode('utf-8')

# Find the index of the first space

spacePosition = header.find(' ')

# Extract everything after the space as the name

name = header[spacePosition+1:]

# Extract everything before the space as the number of packets

packetNumber = int(header[:spacePosition])

return (packetNumber, name)

The code above represents the *getHeaderParts* function. The purpose of this function is to decode the header information (unicode) and separate it into its constituents. It splits the header into the name the user would like the file saved as and how many total packets will be transmitted for the message. The function returns the number of total packets (packetNumber) and the user-specified file name (name).

Code for rdt22.py

The rdt22.py contains several defined functions.

1. def addSequence(pkt, n):

pkt.insert(0, n)

The addSequence() function receives packet as bytearray and sequence number, and inserts the sequence number at the front.

1. def makeRDT22(pkt, n):

addSequence(pkt, n)

addChecksum(pkt)

The makeRDT22() function takes a bytearray packet and a sequence number as input and adds sequence number and checksum to the packet.

1. def AckRDT22(n):​

ack = bytearray()

makeRDT22(ack, n)

return ack

The ActRDT22() receives a sequence number (one byte) and generates an ACK message from it including a checksum.

1. def checkRDT22(pkt, n):

pkt = bytearray(pkt)

if n == pkt[2]:

print('Sequence numbers match')

if compareChecksum(pkt):

print('Checksum matches')

return True

else:

print('Checksum does not match')

else:

print('Sequence number does not match')

return False

The function checkRDT22() checks a packet for its sequence and checksum and returns a boolean True if both matches, otherwise False.

1. def removeHeader(pkt):

return pkt[3:]

The above function removes the leading 3 bytes (checksum and sequence) of a packet.

1. def validatePkt(socket, seq, numPkt = None, totPkt = None):

​ while True:

message, clientAddress = socket.recvfrom(2048)

if numPkt == None and totPkt == None:

print('Message Header Received')

else:

print('\nPacket ' + str(numPkt) + '/' + str(totPkt) + ' Received')

# Check message checksum

if checkRDT22(message, seq):

ack = randomCorruptAck(seq)

socket.sendto(ack, clientAddress)

print('Sending Ack with Seq '+str(seq)+'\n')

break

else:

ack = randomCorruptAck(0 if seq else 1)

socket.sendto(ack, clientAddress)

print('Sending Ack with Seq '+str(0 if seq else 1)+'\n')

The validatePkt() function is for the receiver/ server side. After receiving a packet from the sender/ client, it waits for the correct packet and sends the sender an acknowledgement (ACK) packet with the number of the expected sequence. Here, we corrupted the ACK packet using randomCorruptAck() function, which corrupts ACK packets randomly before sending them to the sender/ client.

1. def randomCorrupt(pkt):

if random()\*100 < CORRUPT\_PROB:

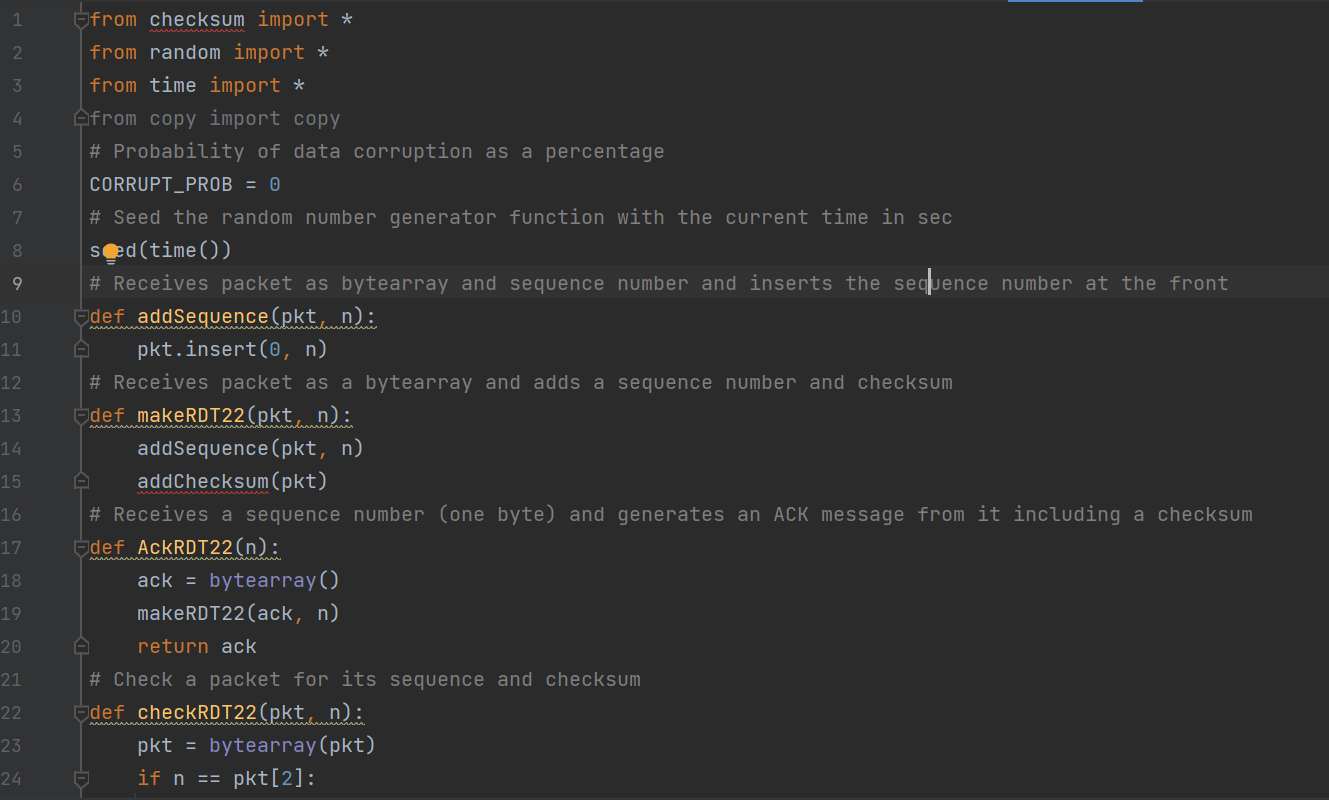
pkt[0] = abs(abs(~pkt[0]) - 10)

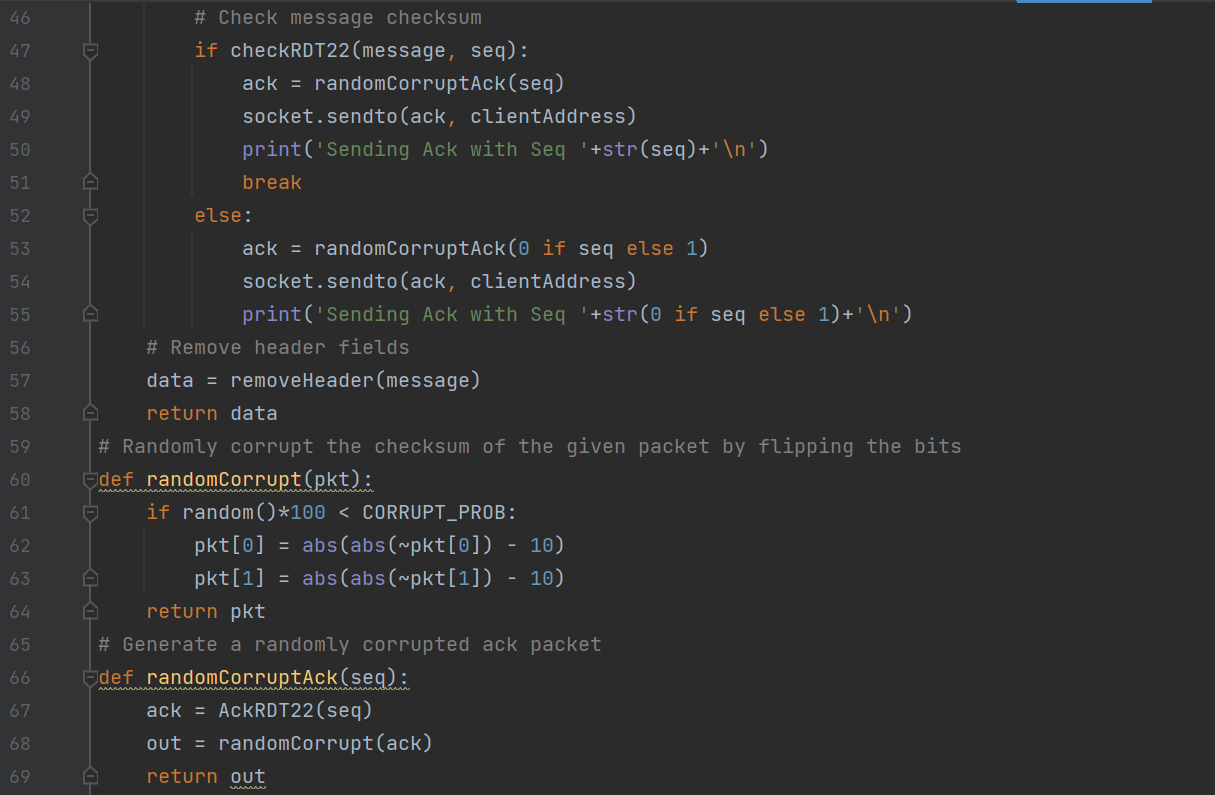
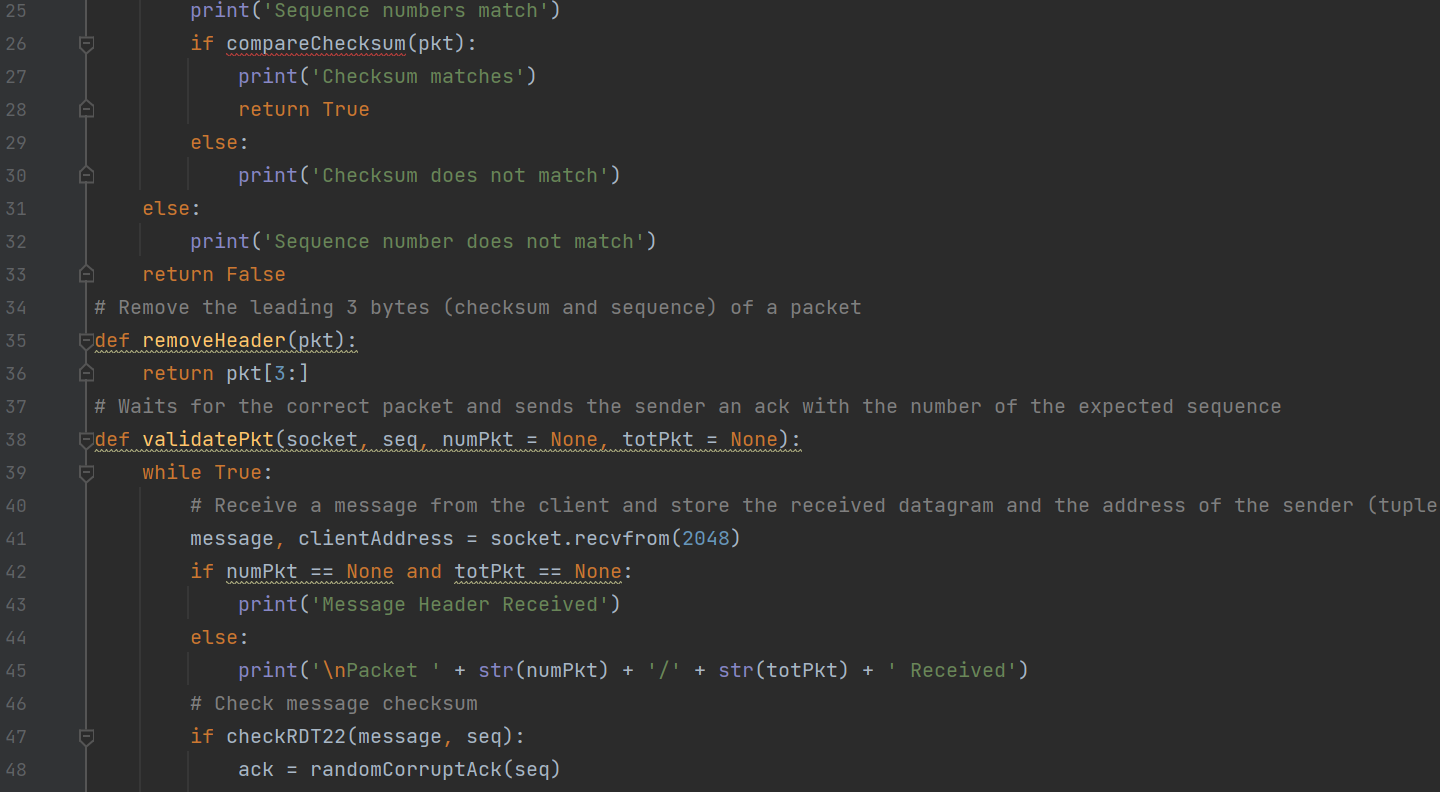
pkt[1] = abs(abs(~pkt[1]) - 10)

return pkt

The randomCorrupt() function corrupts the checksum of the given packet by flipping the bits for a given probability (CORRUPT\_PROB). It is done to ensure corruption.

The screenshot of rdt22.py script is shown below.





Execution Example

Procedure for Communication between client and server are described below.

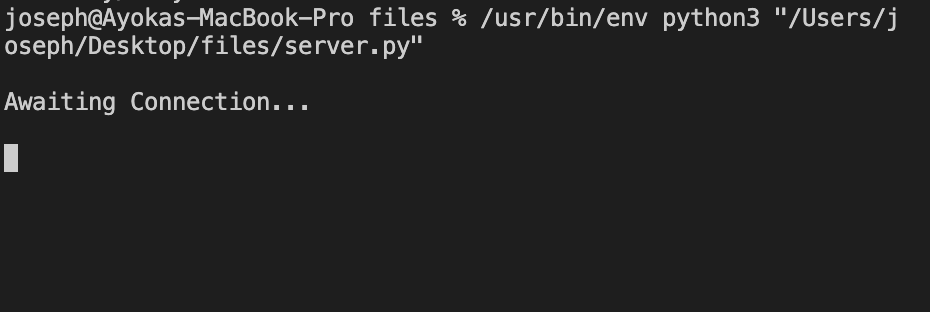
Step 1:

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



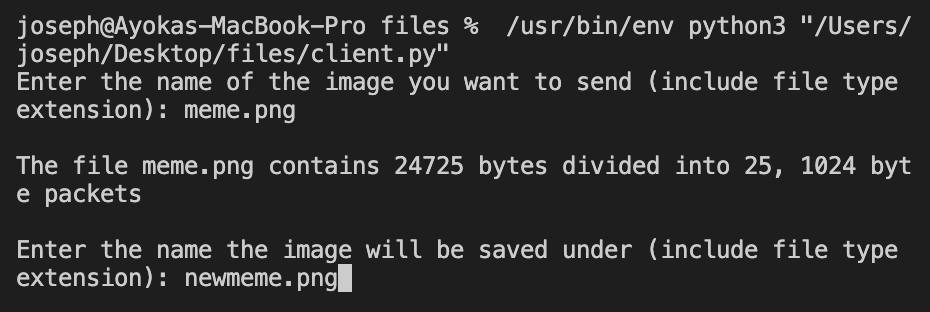
Step 2:

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



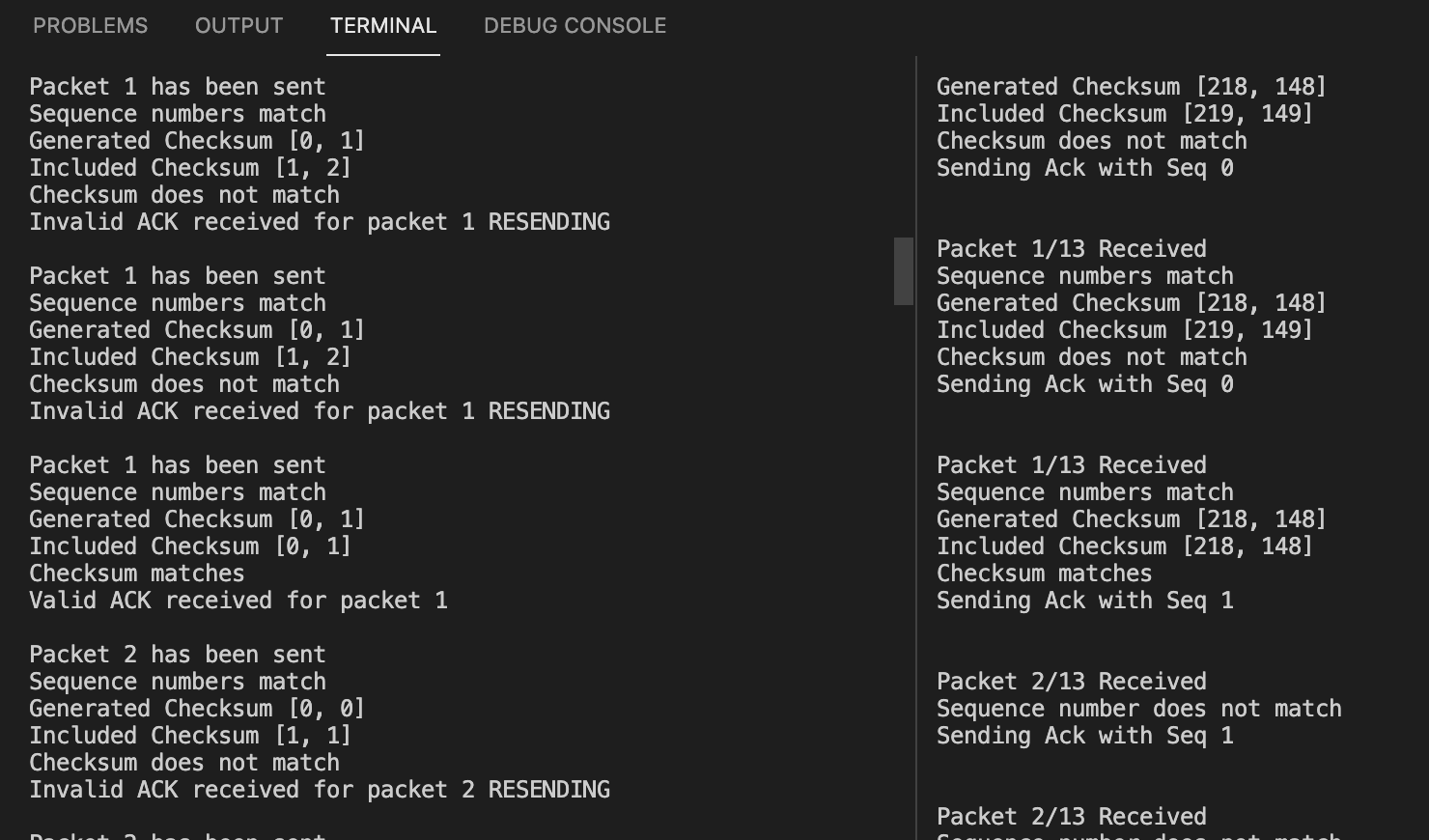
Step 3:

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

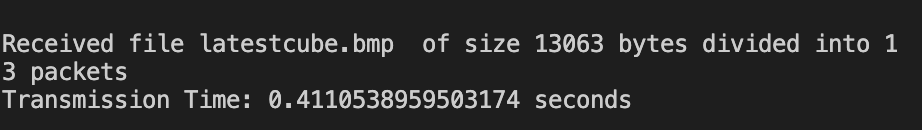


**Results:**

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:** |
| --- | --- |
|  |  |
|  |  |