

School of Arts and Sciences

CSC 450 Computer Networks

Spring 2024

**Group Number 5 Team Members:**

Ali Hassoun 202202847

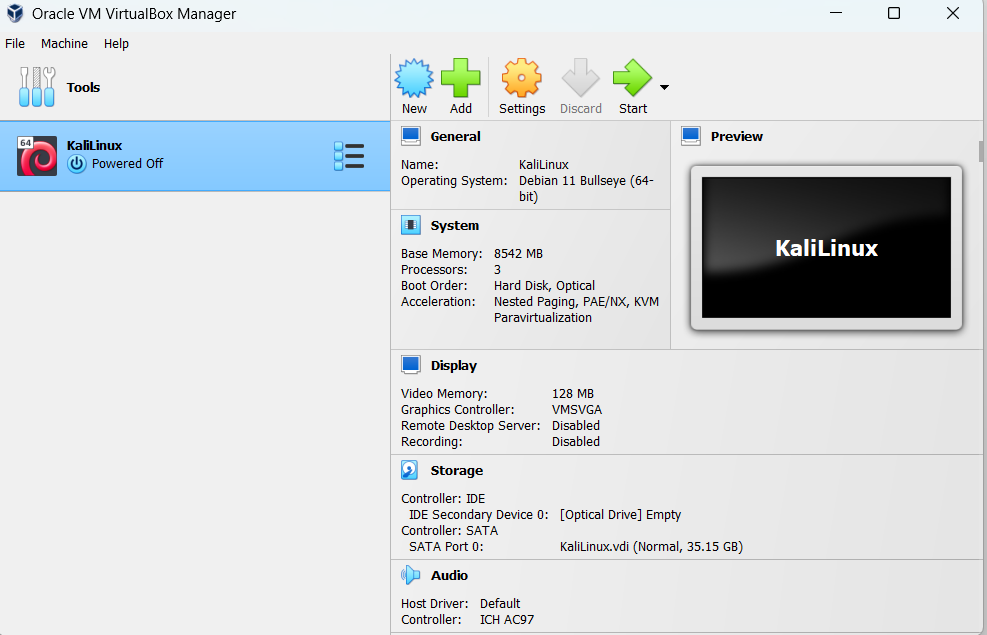
Majid El Ezzi 202206742

Mohammad Al Asal 202202154

**Phase I: Preparation Phase:**

1. **Environment Preparation:**

Oracle VM virtual box was downloaded in order to access the Linux operating system. Debain 11 Bullseye (64-bit) operating system was used.



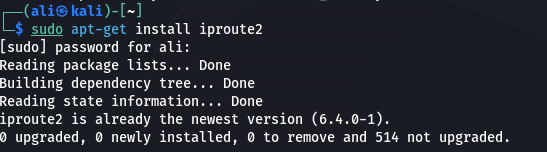
1. **Testing the Network Emulator Netem:**

**First: Install Netem:**

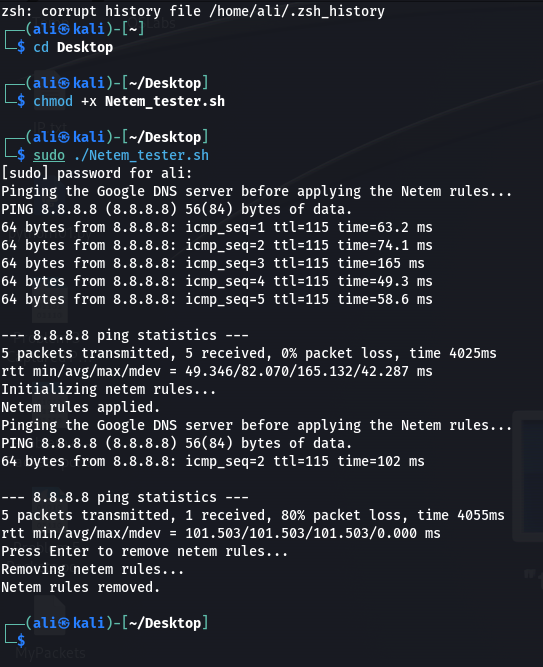
To install Netem we enter the following command:

sudo apt-get install iproute2

This command will install the iproute2 package that has the tc command, which is the traffic control utility in Linux responsible for configuring Netem.

****

**Second: Netem testing:**

We tried to ping 5 times the Google DNS server that has an IP address of 8.8.8.8, before and after adding 100ms of delay, 75 ms of jitter with a normal distribution simulating network latency with variability, and 50% packet loss rate as well as 20% corruption rate. The results clearly show that Netem is working perfectly fine. 

We could see that the packet loss jumped from 0% to 80%, and the delay increased a bit in compared to the results before applying the Netem rules.

**The used bash script is as follows:**

A screenshot of a computer

Description automatically generated

1. **Check if Python is running:**

To check if python is normally running on the virtual machine we wrote a very simple python program:



The output was as follows:

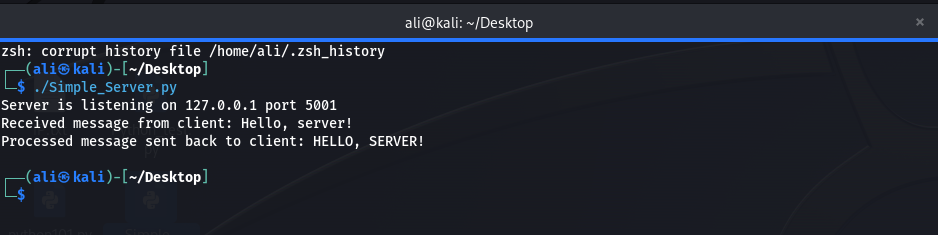
A screen shot of a computer program

Description automatically generated

1. **Check For Connectivity:**

We implemented a simple Client-Server code to check for connectivity. These codes where tested on UDP before implementing reliability over it. The code snippets and outputs are as follows:

**Server (output):**



**Client (output):**

**A screen shot of a computer

Description automatically generated**

**The code snippets are:**

**Server:**

**A screen shot of a computer code

Description automatically generated**

**Client:**

**A screen shot of a computer screen

Description automatically generated**

**Phase II: Reliability over UDP:**

To fulfill the requirements of a chatting app, both peers should be able to:

* Send messages reliably through the UDP sockets.
* Always listen for incoming messages.

The above functionalities should be available at the same time and thus we decided to use threads. The benefits of threads are:

* **Improved performance:** By allowing different parts of our program to run concurrently, threads can improve the overall performance of our application.
* **Simplified programming:** In many cases, using threads can simplify the programming process by allowing us to break up complex tasks into smaller, more manageable pieces that can be executed concurrently.
* **Resource sharing:** Threads can share resources such as memory, files, and network connections, which can reduce overall resource usage and improve the efficiency of our program. (Resource sharing would be beneficial for the overall added reliability where some of the features are to be accessible by the different threads e.g., sequence number)

**(Note the above information was taken from the Operating System’s course slides)**

Our implementation of the reliability over UDP was based on the RDT 3.0, however we added sequence numbers. To write clear code we partitioned the application into multiple files, each of which contains may contain one or more classes in which they all communicate with each other to be able to fully implement the application. Now we are going to go over each file and explain the classes purposes along with their functions. The first file we are going to go over is the util.py file:

def make\_packet(ack: int, data: str):

    return get\_checksum(data) + '\t' + str(ack) + '\t' + data

def make\_ack(ack: int):

    return str(ack) + '\t '

def corrupt\_from\_sender(data: str):

    try:

        checksum, ack, data = data.split('\t', 2)

        int(ack)

        if get\_checksum(data) != checksum:

            return True

        return False

    except:

        print("Corrupt: ", data)

        return True

def corrupt\_from\_receiver(data: str):

    try:

        ack, \_ = data.split('\t', 1)

        int(ack)

        return False

    except:

        return True

def get\_ack(data: str):

    ack, \_ = data.split('\t', 1)

    return int(ack)

def get\_sender\_ack(data: str):

    \_\_, ack, \_ = data.split('\t', 2)

    return int(ack)

def get\_checksum(data: str):

    # Calculate checksum

    checksum = 0

    data = data.encode('utf-8')

    # Padding if the length of data is odd

    if len(data) % 2 != 0:

        data += b'\x00'

    # Calculate the one's complement sum

    for i in range(0, len(data), 2):

        word = (data[i] << 8) + data[i + 1]

        checksum += word

    left\_part = (checksum >> 16)

    right\_part = (checksum & 0xffff)

    checksum = left\_part + right\_part

    checksum += checksum >> 16  # add any trailing 1

    checksum = ~checksum & 0xffff  # ones complement (the & 0xffff is to discard the higher order bits)

    return str(checksum)

def read\_file\_in\_chunks(file\_path, chunk\_size=1024):

    with open(file\_path, 'rb') as file:

        while True:

            data = file.read(chunk\_size)

            if not data:

                break

            yield data

def get\_content(message: str):

    return message.split('\t', maxsplit=2)[-1]

def tcp\_create\_first\_packet(name, size):

    return str(size).ljust(10, " ") + '\t' + str(name).ljust(40, " ")

def tcp\_read\_first\_packet(packt: str):

    size, name = packt.split('\t', 1)

    size = size.strip()

    name = name.strip()

    return name, int(size)

* make\_packet(ack: int, data: str): This function would return a packet where it would first contain a checksum that is calculated by the get\_checksum() function and then the ack number that should be returned by the receiver and the data that should be included in the packet, all separated by a \t.
* make\_ack(ack: int): This function returns the ack number that should be returned by the receiver.
* corrupt\_from\_sender(data: str): This function would take the checksum, ack number, and data from the packet received. Then, this function will calculate the checksum of the data received from the packet and compare it to the checksum that is in the packet if they are equal then this function will return false indicating that the received data is not corrupt, else it will return true indicating that the received data is corrupt.
* corrupt\_from\_receiver(data: str): This function would extract the ack number with respect to a standard formatted string that would be passed to it. After that, the function will try to transform the ack into an integer, if the operation was successful the function will return false indicating that the function is not corrupted. Else, if the operation failed an exception would be raised and it would be handled by returning true indicating that the ack is corrupted as it changed from being a number.
* get\_ack(data: str): This function would return the ack that is sent by a receiver’s ack message.
* get\_sender\_ack(data: str): This function will return the Ack number that is in the sender’s message.
* get\_checksum(data: str): This function would carry out arithmetic operations in order to calculate the checksum of a packet.
* read\_file\_in\_chunks(file\_path, chunk\_size=1024): This function would take a file and will read it in bytes. So, as it starts a while loop it will read the first chunk of data that is of size 1024 bytes as a default. If the data is empty meaning there is no data read then the loop will break, else it will return each chunk is read.
* get\_content(message: str): This function will split the message into the three parts checksum, Ack number, and content and it is going to only return the content.
* tcp\_create\_first\_packet(name, size): This function will take as input the name and the size of a file. Then, it will create the first message that is going to be sent to the receiver, which is the size of the file followed by 10 spaces and a \t and the file name followed by the 40 spaces and a \t.
* tcp\_read\_first\_packet(packt: str): This function will extract the name and the size of the file from the packet that is created by the tcp\_create\_first\_packet() function and it will return them in a tuple.

These are the descriptions of the functions of the util.py file. Later, we will see how these functions are used in our application program. Now, we will discuss the Timeout class in the Timeout.py file:

import time

class Singleton:

    \_instance = None

    def \_\_new\_\_(cls, \*args, \*\*kwargs):

        if cls.\_instance is None:

            cls.\_instance = super().\_\_new\_\_(cls)

        return cls.\_instance

class Timeout(Singleton):

    alpha = 0.25

    beta = 0.3

    def \_\_init\_\_(self):

        self.start = None

        self.end = None

        self.rtt = None

        self.dev\_rtt = None

    def register\_start(self):

        self.start = time.time()

    def register\_end(self):

        self.end = time.time()

        duration = self.end - self.start

        self.update\_rtt(duration)

        self.update\_dev\_rtt(duration)

        Timeout.current\_estimate = self.get\_timeout()

    def update\_rtt(self, sample):

        if self.rtt is None:

            self.rtt = sample

            return

        self.rtt = (1-self.alpha) \* self.rtt + self.alpha \* sample

    def update\_dev\_rtt(self, sample):

        if self.dev\_rtt is None:

            self.dev\_rtt = 0.25 \* self.rtt

        self.dev\_rtt = (1-self.beta)\*self.dev\_rtt + self.beta\*abs(self.rtt - sample)

    def get\_timeout(self):

        if self.rtt is None:

            return 2

        return self.rtt + 4 \* self.dev\_rtt

The timeout class inherits the singleton class to make sure that only one instance is created from this class to avoid inefficiencies as it will provide a global point of access to that instance. Whenever we initialize an instance of this class we will have the following variables:

* start = none.
* end = none.
* rtt = none.
* devrtt = none.
* register\_start(self): What this function does is that it saves the current time in the self.start variable this will later be used to calculate the delay in the network.
* register\_end(self): It is going to be triggered upon the receipt of an ack. What will happen in this function is that the self.end variable will be storing the time when the ack was received and then calculating the delay in the network by subtracting the self.start variable firstly initialized by the register\_start() function with the self.end variable and storing them in the duration variable. This duration variable will be used to update the RTT and DevRTT using the update\_rtt() and update\_dev\_rtt() functions and then will be utilized to calculate the Timeout using the get\_timeout() function.
* update\_rtt(self, sample): This will update the value of the variable self.rtt according to the value of the duration, using the same formulas we learnt in class. It will take as input the sample rtt and if the rtt is none then the current rtt of the system will be the sample rtt and after that it will carry out the calculations.
* update\_dev\_rtt(self, sample): This will update the value of the self.dev\_rtt according to the value of the duration variable, using the same formulas we learnt in class. It will take the sample rtt as input.
* get\_timeout(self): This function will return the value of the timeout, using the same formulas we learnt in class. However, if the rtt was not set for some reason it will consider that the timeout is 2 seconds which is relatively high.

That is it for the timeout.py file. Now we will discuss the base\_classes.py file, that has the two classes sender and receiver, and we will see how the functions in the util.py and timeout.py files will be used:

class Sender:

    def \_\_init\_\_(self, address, friend\_address):

        self.address = address

        self.friend\_address = friend\_address

        self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

        self.sock.bind(self.address)

        self.send\_buffer = Queue()

        self.state = 0

        main\_thread = threading.Thread(target=self.run)

        main\_thread.daemon = True

        main\_thread.start()

        self.timeout = Timeout()

    def send(self, data: str):

        self.sock.sendto(data.encode('utf-8'), self.friend\_address)

        self.timeout.register\_start()

    def receive(self):

        data, addr = self.sock.recvfrom(1024)

        return data.decode('utf-8')

    def state\_sending\_from(self):

        packt = self.send\_buffer.get()  # This is a blocking queue call

        packt = make\_packet(self.state, packt)

        self.send(packt)

        return packt

    def state\_wait\_for\_ack(self, packt: str):

        self.sock.settimeout(self.timeout.get\_timeout())

        while True:

            try:

                data = self.receive()

                if corrupt\_from\_receiver(data) or get\_ack(data) != self.state:

                    continue

                break

            except socket.timeout:

                # retry sending

                self.send(packt)

                # restart timer

                continue

        self.timeout.register\_end()

        self.sock.settimeout(None)

    def run(self):

        while True:

            packt = self.state\_sending\_from()

            self.state\_wait\_for\_ack(packt)

            self.state += 1

The class first would initialize the following variables:

* self.address = address, this variable is the IP address and port number of the sender in a tuple.
* self.friend\_address = friend\_address, this variable is the IP address and port number of the receiver in a tuple.
* self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM), here we are creating a socket for the sender so it will be able to send and receive messages on it.
* self.sock.bind(self.address), here we bind the port number with the IP address of the sender in order to be able to send messages.
* self.rcv\_buffer = Queue(), this is a queue that will contain the messages that will be sent to the receiver.
* self.ack\_state = 0, specifies the number of the message, in which the receiver would ack indicating that this message is received correctly. The state can either be 1 or 0.
* main\_thread = threading.Thread(target=self.run), here we are starting a thread with the run function that will be discussed later on.
* main\_thread.daemon = True, this indicates that the thread is a daemon thread. The daemon thread will run in the background and will not prevent the program from exiting once the main thread completes its execution. So, it is a thread that runs independently of the body and other threads.
* main\_thread.start(), the thread will start running.
* self.timeout = Timeout(), this is an instance of the timeout class it will be used to control the timeout throughout the program.
* send(self, data: str): This function takes an argument data of string type and then starts the register\_start() function after encoding the data and then sending it.
* receive(self): This function is for receiving messages. Where the received encoded data will be stored in the data variable and the IP address and port number of the contacted client would be saved in the addr variable as a tuple. Finally, the function would return the decoded data.
* state\_sending\_from(self, state\_ack): first the program would check if the queue is empty. If this condition is true, the .get() function will block the execution of the code until there is something available in the queue. As soon as it finds something, it will remove it from the queue and return it to the packt variable. Then, the packt variable will be used to construct a packet with ack number and checksum using the make\_packet() function in the util file. After that we will send the packet using the send() function. Finally, it will return the packet.
* state\_wait\_for\_ack(self, packt: str, state\_ack): first, this function would set the timeout as the calculated timeout in the timeout class using the get\_timeout() function. After that, it will enter a while loop waiting for the ack to be received. If the ack is corrupted or has a number different than the one that the sender is waiting for it will wait for the timeout to occur. if the timeout occurs the sender will resend the packet and will wait for the ack again. When, the ack is successfully received the register\_end() function will be called to set the end variable to the time of the receipt of the ack. This value will then be used to calculate the new timeout. Finally, we will reset the value of the timeout.
* run(self): This function combines all of the above functions. It sends the packet using the state\_sending\_from() function, waits for an ack using the state\_wait\_for\_ack() function, and switches states from the state before (i) to (i+1).

That is it for the sender class. Moving on to the receiver class:

class Receiver:

    def \_\_init\_\_(self, address, friend\_address):

        self.address = address

        self.friend\_address = friend\_address

        self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

        self.sock.bind(self.address)

        self.rcv\_buffer = Queue()

        self.ack\_state = 0

        main\_thread = threading.Thread(target=self.run)

        main\_thread.daemon = True

        main\_thread.start()

    def send(self, data: str):

        self.sock.sendto(data.encode('utf-8'), self.friend\_address)

    def receive(self):

        data, addr = self.sock.recvfrom(1024)

        return data.decode('utf-8')

    def send\_ack(self, ack\_num):

        self.send(str(ack\_num) + '\t ')

    def run(self):

        while True:

            packt = self.receive()

            if corrupt\_from\_sender(packt):

                continue

            if get\_sender\_ack(packt) != self.ack\_state:

                # send ack

                self.send\_ack(get\_sender\_ack(packt))

                continue

            self.rcv\_buffer.put(packt)

            self.send\_ack(self.ack\_state)

            self.ack\_state += 1

When it comes to the initialization process:

self.address = address, this is a tuple containing the IP address and port number of the receiver.

self.friend\_address = friend\_address, this is a tuple containing the IP address of the sender and its port number.

self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM), creating a socket for the receiver to communicate the acks to the sender.

self.sock.bind(self.address), here we are binding the IP address and the port number of the receiver, this is a necessary part of the socket initialization process giving the socket a unique identifier.

self.rcv\_buffer = Queue(), this buffer will contain the received messages.

self.ack\_state = 0, this will be the number that will be sent as an ack number to the sender. This is to assure the sender that the data that it sent with a certain state number has been received successfully.

main\_thread = threading.Thread(target=self.run)

main\_thread.daemon = True

main\_thread.start()

That is it regarding the initialization process. Now we are going to discuss the functions:

* send(self, data: str): This function will be used to send data to the client the receiver is contacting. Mostly, it will be used to send Acks, this will be clearer later own.
* receive(self): This function will be used to receive the encoded data from the client the receiver is contacting as well as the IP address and port number of the client. Finally. It will return the decoded data.
* send\_ack(self, ack\_num): This function will send the ack number of the received data to the client using the send() function defined above.
* run(self): First the function will be waiting for data to be received. If the packet is received and it appears to be corrupted, it will not send an ack instead it will wait for sender’s timeout to finish so that it will resend the data. Else, if the message is not corrupted but the ack number included in the packet is different than the ack number the receiver expects, the receiver will send an ack with the received ack number to indicate to the sender that the packet was successfully received. If nothing goes wrong, the received data will be put in the receiver’s buffer, an ack numbered as the current state will be sent, and the receiver will change its state from i to i+1

So, upon the observations made on the roles of the functions we can see how RDT 3.0 works, and so does our program. Whenever a message is available, we will send this message with a state number for the message, and then we will change our states from the sending state to the waiting for ack with the same state number sent. In case the sender receives an ack number different than the one it is waiting for, it will discard it wait for the time out and resend the data. Else, it would send a new message with a new state number and the process repeats. For the receiver, it will be waiting for a message with a certain state. If the received packet is of a different state or a packet containing corrupted data it will discard it and will wait for time out of the sender in which it will resend the data message. Else, if it receives the data and state number is expected, the receiver will accept the message put it in a buffer and change it’s state which is waiting for the new message with the new state number. The sending and receiving processes will be available for every client using the application at the same time, thanks to threads.

Finally, we will look at the RDT file, which is the main file where it contains a class such that we can create instances that are able to reliably send and receive data, by creating instances of the sender and receiver classes. Let us look at the class:

"""

EVERYTHING IS NON-BLOCKING AND THREADED

"""

from infrastructure.base\_classes import Sender, Receiver

from infrastructure.file\_sending import TCPReceiver, TCPSender

from infrastructure.util import get\_content

class AppClient:

    # Note that each client needs 4 ports. 2 for each protocol (one is receiving and one is sending).

    def \_\_init\_\_(self, rcv\_address, send\_address, friend\_rcv\_address, friend\_send\_address, path\_to\_files\_save,

                 rcv\_address\_tcp, send\_address\_tcp, friend\_rcv\_address\_tcp, friend\_send\_address\_tcp):

        self.sender = Sender(send\_address, friend\_rcv\_address)

        self.receiver = Receiver(rcv\_address, friend\_send\_address)

        self.tcp\_sender = TCPSender(send\_address\_tcp, friend\_rcv\_address\_tcp)

        self.tcp\_receiver = TCPReceiver(rcv\_address\_tcp, friend\_send\_address\_tcp, dir\_path=path\_to\_files\_save)

    def send\_message(self, message):

        self.sender.send\_buffer.put(message)

    def get\_messages(self):

        messages = []

        while not self.receiver.rcv\_buffer.empty():

            m = self.receiver.rcv\_buffer.get()

            messages.append(get\_content(m))

        return messages

    def send\_file(self, path\_to\_file):

        if self.tcp\_sender.is\_sending():  # cannot send two files simultaneously

            return False

        self.tcp\_sender.send\_file(path\_to\_file)

        return True

    def get\_next\_message(self):

        """

        This is a blocking call

        :return:

        """

        message = self.receiver.rcv\_buffer.get(block=True, timeout=None)

        return get\_content(message)

    def get\_next\_file\_sent(self):

        filename = self.tcp\_receiver.buffer.get(block=True, timeout=None)

        return filename

    def close\_all(self):

        self.sender.sock.close()

        self.receiver.sock.close()

        self.tcp\_sender.sock.close()

        self.tcp\_receiver.sock.close()

First, as mentioned before we will instantiate instances of the sender and receiver classes to be able to send and receive data:

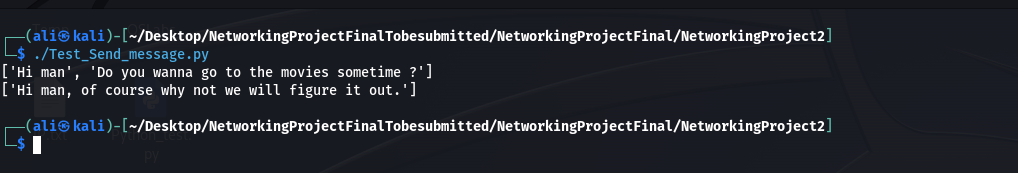
* As mentioned before, this class is for doing a whole application client. This client will be able to send and receive messages as well as send and receive files. There will be 4 ports for each client as we have 4 different functionalities running on four different threads. Threading would help us achieve efficiency when sending messages since functions would run independent of each other. This will help the client to receive messages, send messages, send files, and receive files at the same time without overhead.
* send\_message(self, message): This function will facilitate the sending of the messages by putting them in the send buffer. As the send buffer becomes populated with data, the run method that is being executed by the thread initialized in the init function in the AppClient class will use the state\_sending\_from() method to send the data, as mentioned before this method will send a message as long as the buffer is not empty. Then, it will wait for the ack to be received correctly by using the state\_wait\_for\_ack() function and finally the it will change state. As it receives the correct Ack, it will then send the next message in the queue.
* get\_messages(self): This function will put the messages received from the receiving buffer in an array and it will continue to do this as long as the buffer is not empty and finally it will return the array. This will happen as the receiver’s run function that is being activated by the thread in the init function will get the data, check if it corrupted and the packet has the same ack number as its state by using the functions corrupt\_from\_sender(packt) and get\_sender\_ack(packt) in the util class.
* get\_next\_message(self): This function will take the received data from the receive buffer, remove it from the buffer and then it will return its content only.
* The other functions are for phase 3. They will be discussed in details later on.

Now, it is time to test the program using netem:

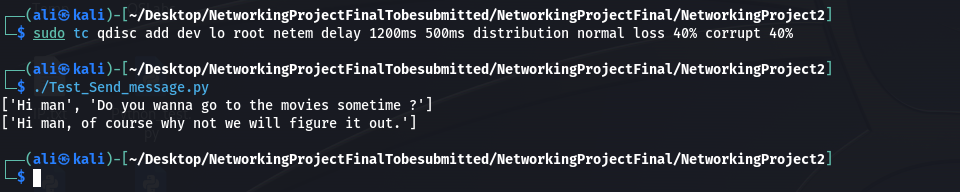
We will use this code:



This is the output without activating netem:



This is the output after using netem adding a 1200ms delay and 500ms jitter 40% packet loss and 40% corruption rate:



**Phase III: Testing File Transmission over TCP:**

The implementation decision was to send small chunks of the file to the receiver, each chunk is of 10 bytes. The File sending was done in the File\_sending.py file, where we have to classes TCPSender and TCPReceiver:

TCPSender:

* CHUNK\_SIZE = 10
* FIRST\_PACKET\_SIZE = 51
* class TCPSender:
* def \_\_init\_\_(self, my\_address, friend\_address):
* self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
* self.timeout = Timeout()
* self.update\_timeout()
* self.other\_address = friend\_address
* self.this\_address = my\_address
* self.main\_thread = None
* def run(self, path):
* # Cover me, I'm reloading
* while True:
* try:
* self.update\_timeout()
* self.sock.connect(self.other\_address)
* break
* except socket.error as e:
* continue
* # This is it, soldiers. Time to make a difference
* file\_name = os.path.basename(path)
* file\_size = os.path.getsize(path)
* first\_packet = tcp\_create\_first\_packet(file\_name, file\_size)
* self.update\_timeout()
* self.sock.sendall(first\_packet.encode('utf-8'))
* # send data
* for data in read\_file\_in\_chunks(path, chunk\_size=CHUNK\_SIZE):
* self.update\_timeout()
* self.sock.sendall(data)
* self.sock.close()
* self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
* def send\_file(self, path):  # Listen up boys. We've got a new mission
* self.main\_thread = threading.Thread(target=self.run, args=(path, ))
* self.main\_thread.daemon = True
* self.main\_thread.start()
* def is\_sending(self):
* if self.main\_thread is None:
* return False
* return self.main\_thread.is\_alive()
* def update\_timeout(self):
* self.sock.settimeout(self.timeout.get\_timeout())
* run(self, path): First this function will update the timeout using the update\_timeout() function which will be discussed later on. After that, the client would keep on trying to establish a connection between him and the other client. As the connection is established, the function will get the name of the file and save it in the file\_name variable and it would also get the file size and save it in the file\_size variable. Then, the function will make the first packet which has the name and size of the file using the tcp\_create\_first\_packet(file\_name, file\_size) function, update the timeout and reliably send the packet. Finally, the function would take chunks of data from the generator function read\_file\_in\_chunks(path, chunk\_size=CHUNK\_SIZE), where the chunk size is 10 bytes and will send the received data after updating the timeout.
* send\_file(self, path): This function will create a daemon thread and will run the run function above. Since this thread is a daemon thread, it will run independently of the main program and other threads.
* is\_sending(self): This function will check if the thread is alive, so that we could decide if we will send a new file or not.
* update\_timeout(self): This function will be used to set the timeout to the new timeout which will be calculated by the get\_timeout() function in the timeout.py file.

TCPReceiver:

class TCPReceiver:

    def \_\_init\_\_(self, my\_address, friend\_address, dir\_path='./received files'):

        self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

        self.sock.bind(my\_address)

        self.other\_address = friend\_address

        self.this\_address = my\_address

        self.directory = dir\_path

        self.buffer = Queue()

        self.timeout = Timeout()

        main\_thread = threading.Thread(target=self.run)

        main\_thread.daemon = True

        main\_thread.start()

    def run(self):

        while True:

            self.sock.listen()

            conn, addr = self.sock.accept()

            # get first packet

            first\_packet = conn.recv(FIRST\_PACKET\_SIZE)

            file\_name, file\_size = tcp\_read\_first\_packet(first\_packet.decode('utf-8'))

            file\_path = os.path.join(self.directory, file\_name)

            file = open(file\_path, 'wb+')

            bytes\_received = 0

            while True:

                data = conn.recv(CHUNK\_SIZE)

                bytes\_received += len(data)

                file.write(data)

                print("Received:", data, "\t\t", bytes\_received, file\_size, self.timeout.get\_timeout())

                if bytes\_received == file\_size:  # Extraction inbound

                    file.close()

                    break

            # we're done here boys, return to base

            self.buffer.put(file\_name)

* The init function will initialize a socket with the needed IP addresses of both the sender and receiver, and the directory where the file will be saved.
* run(self): First the function would listen for incoming connections, and when one is available it will connect. After that it will receive the first packet send by the sender client that has the file name and size and the tuple would be saved in the first\_packet variable. Then, the tcp\_read\_first\_packet() function will be used to extract the file name and size and save them in the file\_name, and file\_size variables respectively. The file path would be created by joining the file name with the directory used to store the file in it. If the file already exists its content will be overwritten or else it will be created, as it is opened file in “rb+”, thus it will be opened in both . After all of this is done, the function will start receiving data and it will calculate the length of the received data, and it will write the data on the file and print a message that shows the status of the received data. If the bytes received are equal to the file size, then this means that the whole file has been received so the function will close the file and the connection and would add the file name to the buffer in order to show it to the user later on in the gui.

Coming back to the AppClient Class:

"""

EVERYTHING IS NON-BLOCKING AND THREADED

"""

from infrastructure.base\_classes import Sender, Receiver

from infrastructure.file\_sending import TCPReceiver, TCPSender

from infrastructure.util import get\_content

class AppClient:

    # Note that each client needs 4 ports. 2 for each protocol (one is receiving and one is sending).

    def \_\_init\_\_(self, rcv\_address, send\_address, friend\_rcv\_address, friend\_send\_address, path\_to\_files\_save,

                 rcv\_address\_tcp, send\_address\_tcp, friend\_rcv\_address\_tcp, friend\_send\_address\_tcp):

        self.sender = Sender(send\_address, friend\_rcv\_address)

        self.receiver = Receiver(rcv\_address, friend\_send\_address)

        self.tcp\_sender = TCPSender(send\_address\_tcp, friend\_rcv\_address\_tcp)

        self.tcp\_receiver = TCPReceiver(rcv\_address\_tcp, friend\_send\_address\_tcp, dir\_path=path\_to\_files\_save)

    def send\_message(self, message):

        self.sender.send\_buffer.put(message)

    def get\_messages(self):

        messages = []

        while not self.receiver.rcv\_buffer.empty():

            m = self.receiver.rcv\_buffer.get()

            messages.append(get\_content(m))

        return messages

    def send\_file(self, path\_to\_file):

        if self.tcp\_sender.is\_sending():  # cannot send two files simultaneously

            return False

        self.tcp\_sender.send\_file(path\_to\_file)

        return True

    def get\_next\_message(self):

        """

        This is a blocking call

        :return:

        """

        message = self.receiver.rcv\_buffer.get(block=True, timeout=None)

        return get\_content(message)

    def get\_next\_file\_sent(self):

        filename = self.tcp\_receiver.buffer.get(block=True, timeout=None)

        return filename

    def close\_all(self):

        self.sender.sock.close()

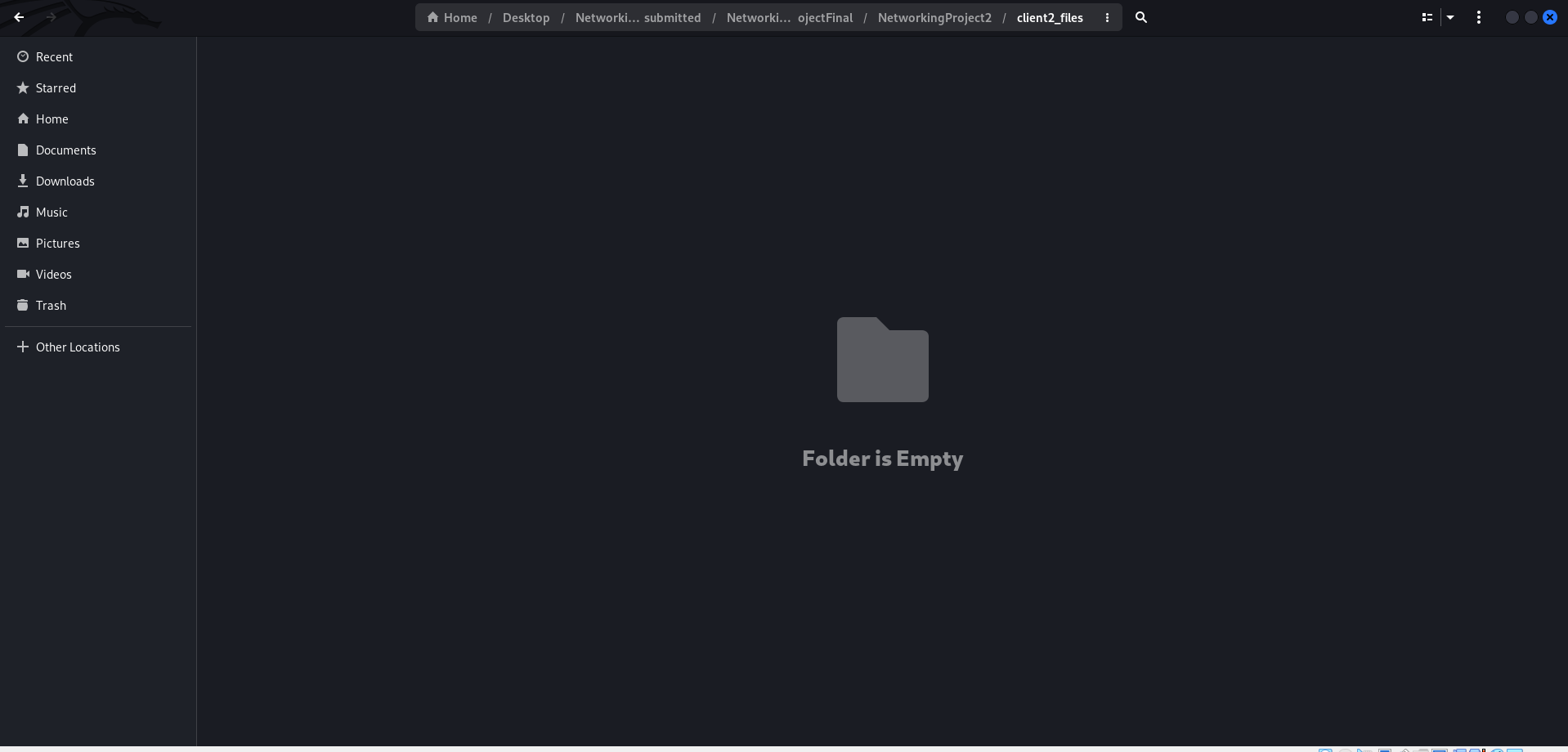
        self.receiver.sock.close()

        self.tcp\_sender.sock.close()

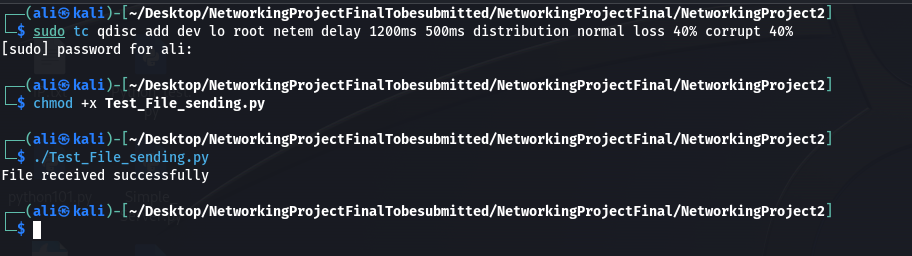
        self.tcp\_receiver.sock.close()

* send\_file(self, path\_to\_file): This function will check if we are able to send a file using the is\_sending() function defined in the TCPSender class. If we cannot send the file it will return false. Otherwise, it will utilize the send\_file() function from the TCPSender class in order to send the file. Finally, when the operation is done, the function will return true.

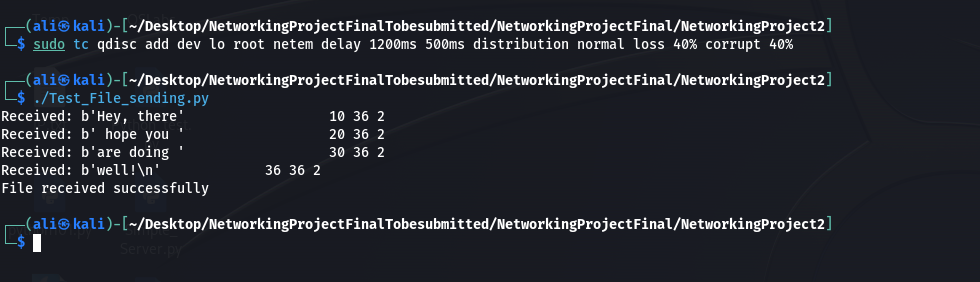
**Testing:**

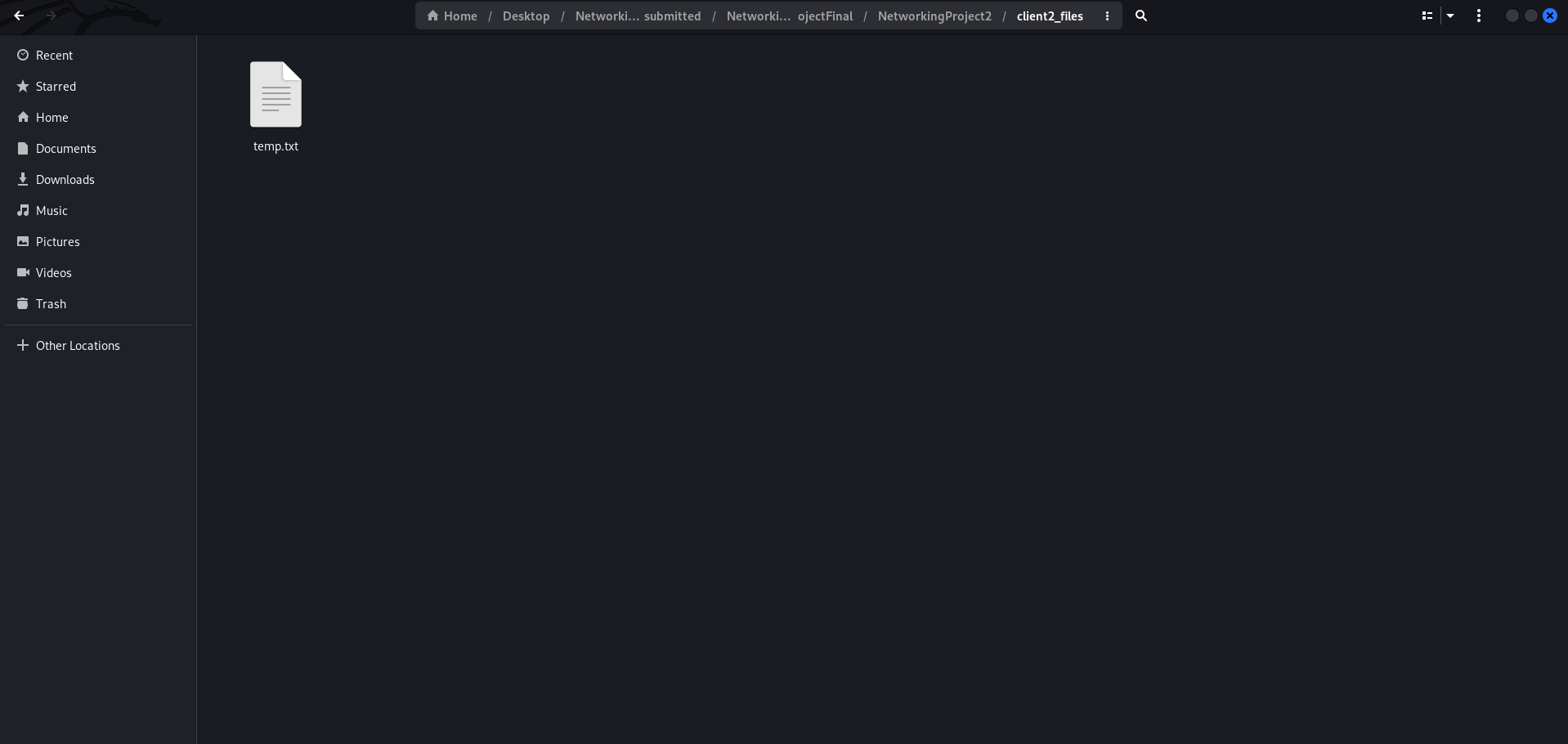
****

Adding delay, jitter, packet loss, and corruption:



Result:





**Phase IV: Application GUI:**

