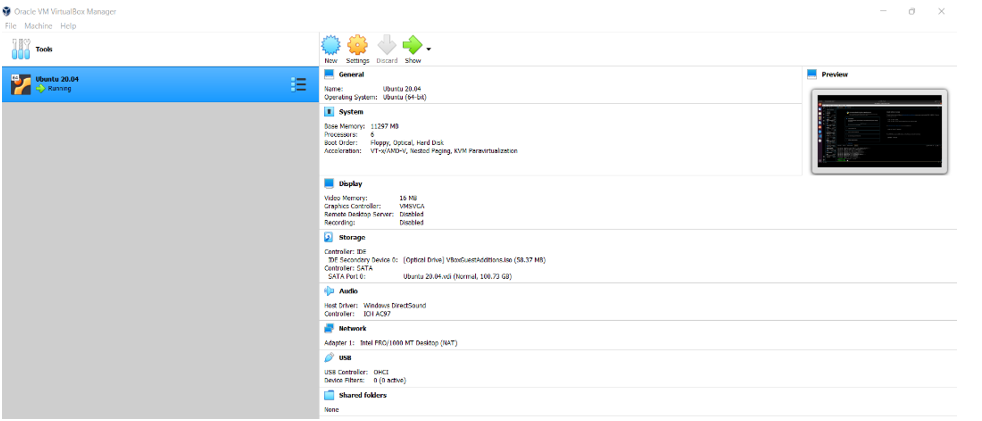
Final Report-EECE 350-Group02

**Phase 1**: Preparation Phase

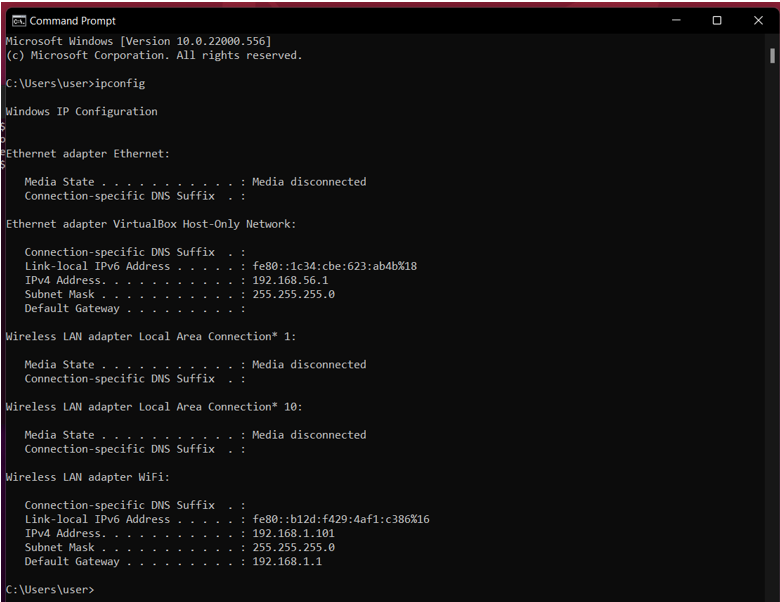
* We have downloaded Linux via the Oracle VM VirtualBox Manager



* On the Virtual Machine, we then proceeded to download Ubuntu 20.04 LTS via <https://ubuntu.com/#download>
* After setting up ubuntu’s initial properties, the virtual desktop was not maximized. To solve the issue, we installed virtual box guest editions. The command “Sudo apt install build-essential dkms Linux-headers-$(uname –r)” in the terminal solved the problem.
* We proceeded to activate netem in Ubuntu using the “Sudo apt-get install iproute2” in the command terminal
* Afterwards, we have installed vscode and added the extension for python and jupyter. Then, we tested out the correct functioning.



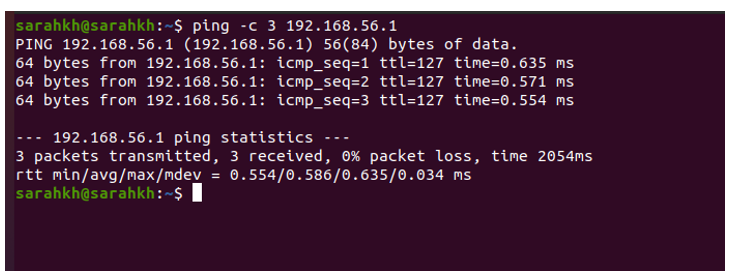
* The next step consisted of testing netem and familiarizing ourselves with its main functionalities. A) found the internal IP by inserting “iconfig” in the command prompt

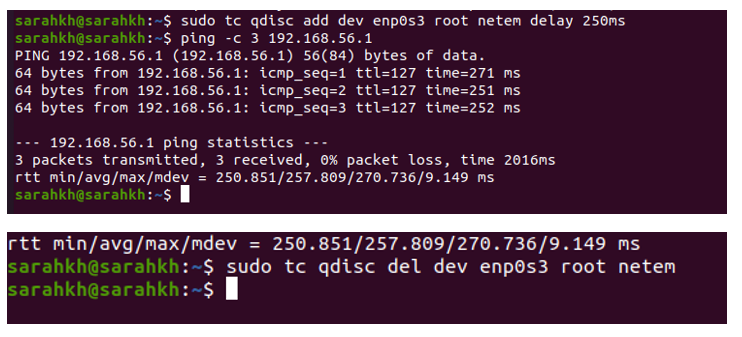


B) Sent a ping to the IP address: “ping-c 3 192.168.58” (the ping stops after 3 packets)

C) Found the logical name of the network device

D) Added a 250mc latency delay to the interface enp0s3 and resend a ping to the prior IP address

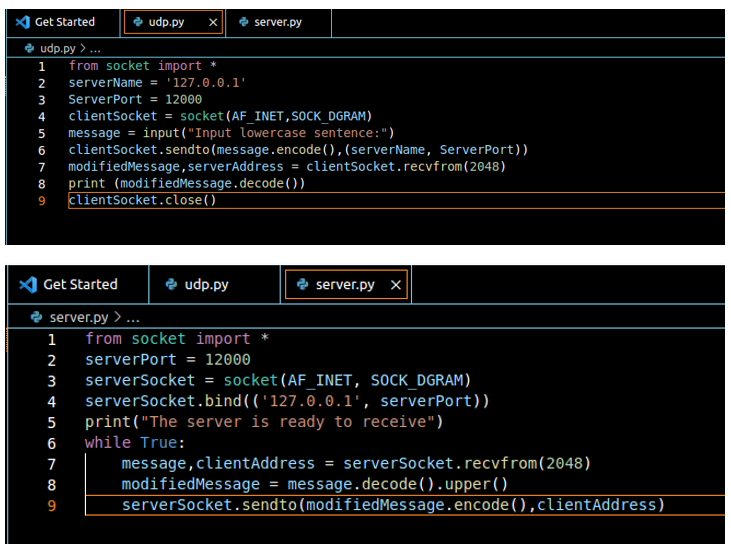




The initial average rtt time=0.586 ms

After using netem: it became 257.809 ms

* For the connection part: we tested out the initial UDP connection between client and server



* As basis for Phase 2, we then looked at different online resources and gathered different codes that potentially

**Phase 2:**

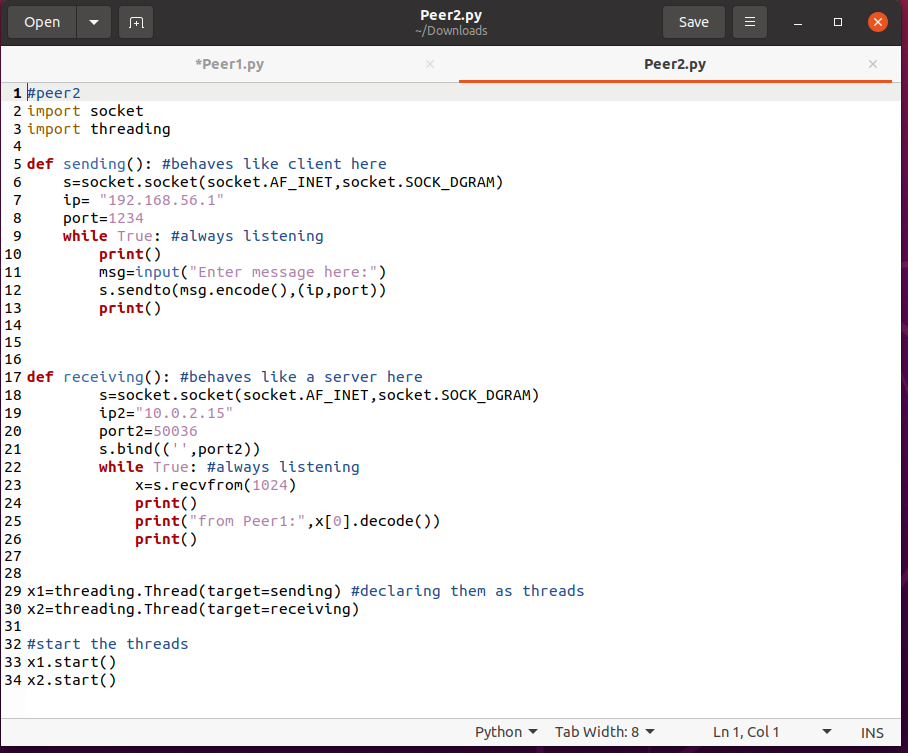
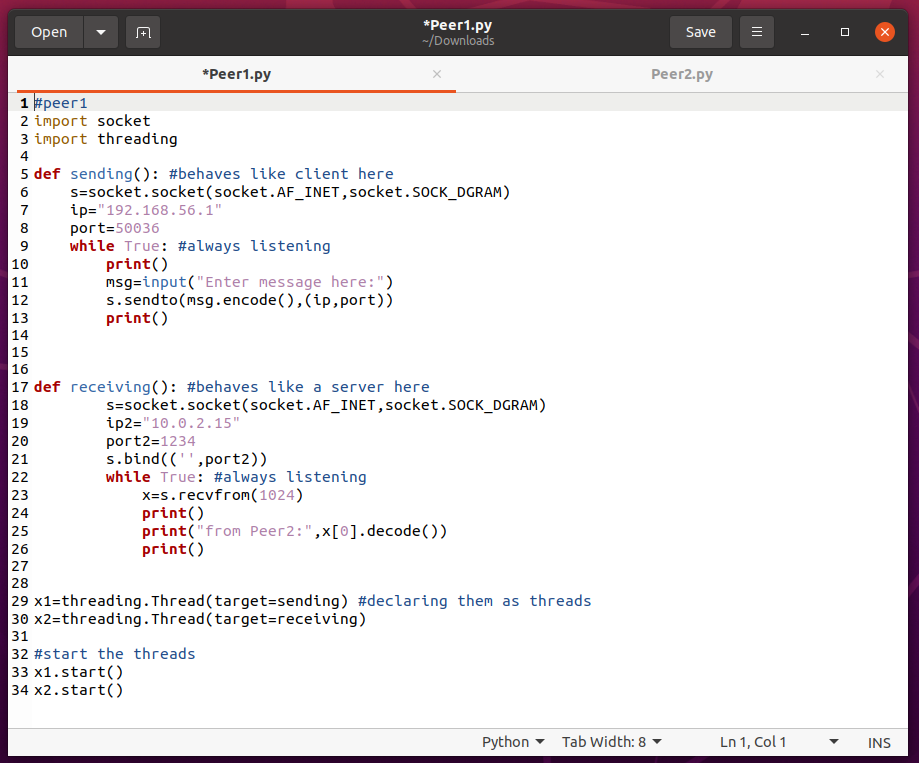
1. Part a of phase two constituted of the basic connectivity between a peer (client) and a server.

- With the help of the slides and codes provided in class, we were able to establish the UDP connection as attached above

b) The second part used the first part to establish a peer-to-peer connection between a peer and another peer (peer-to-peer connection)

- A sending and a receiving function were defined to be then called by the main program.

- Both peers have both functions to send and receive data on both ends of the “conversation”



c) For the third part, we went over different methods and codes. We experimented with a few client-serve codes. After trial and error, we set our code to be a class based one. In that matter, we would be able to decentralize the code within turn would facilitate the building phase 3 on top of phase 2.

The detailed level implementation of the final code of phase 2 is explained line-by-line in “peer2.py” in the zip file as comments. It was easier to explain the logic of phase 2 through comments, but here are some pointers:

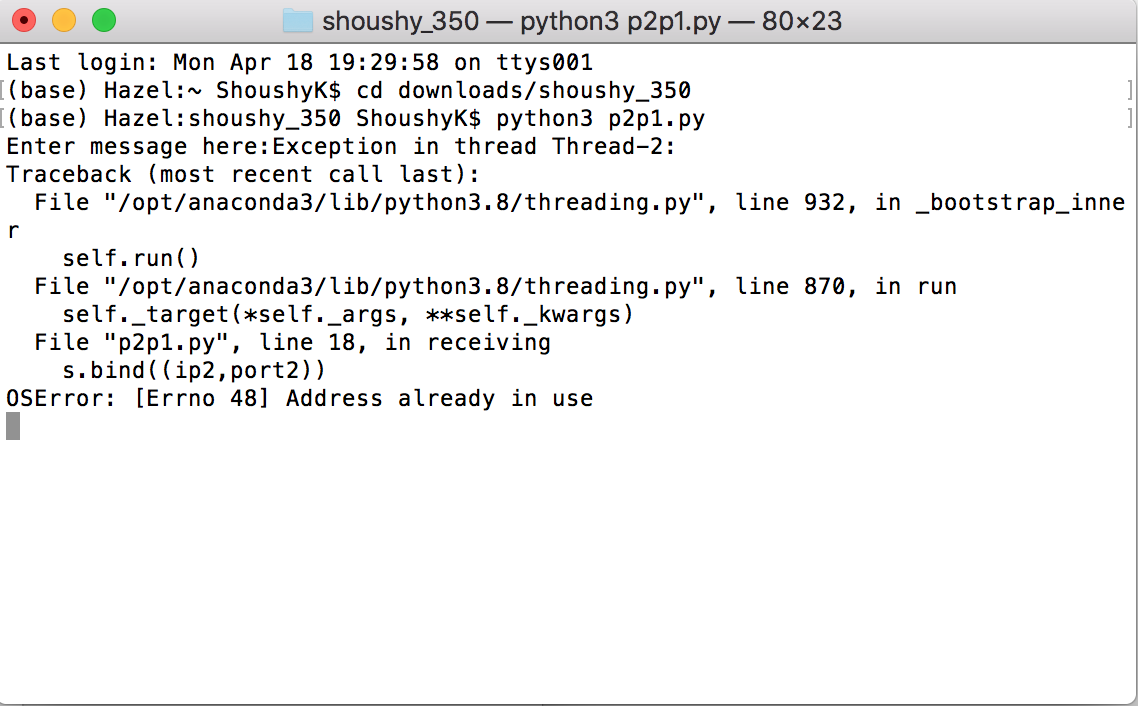
Steps taken (in brief):

1. Tested the UDP client/server code snippet given in the PowerPoint (task a)
2. Modified the above for both the server & client to be able to listen and send messages to each other using threading (task b)
3. Extensively researched python codes to get ideas on how others have approached similar problem statements requiring application layer intervention for reliability; familiarized oneself with potentially useful python libraries (task c)
4. Decided on what protocol to follow to address select unreliability issues (task c)
5. Testing the code with netem (task d)

Challenge: See what ports are already in use

There are port numbers already in use, so use any port number in the range of unused/available port numbers. To see what ports are in use by UDP I used the command: **lsof -I -n -P | grp UDP**

Challenge: Kill ports already in use (to reuse ports after you run the code) A useful command used to kill a port number and reuse it: **kill -15 $(lsof -t -i :12000)** to solve issues like the one screenshotted below.



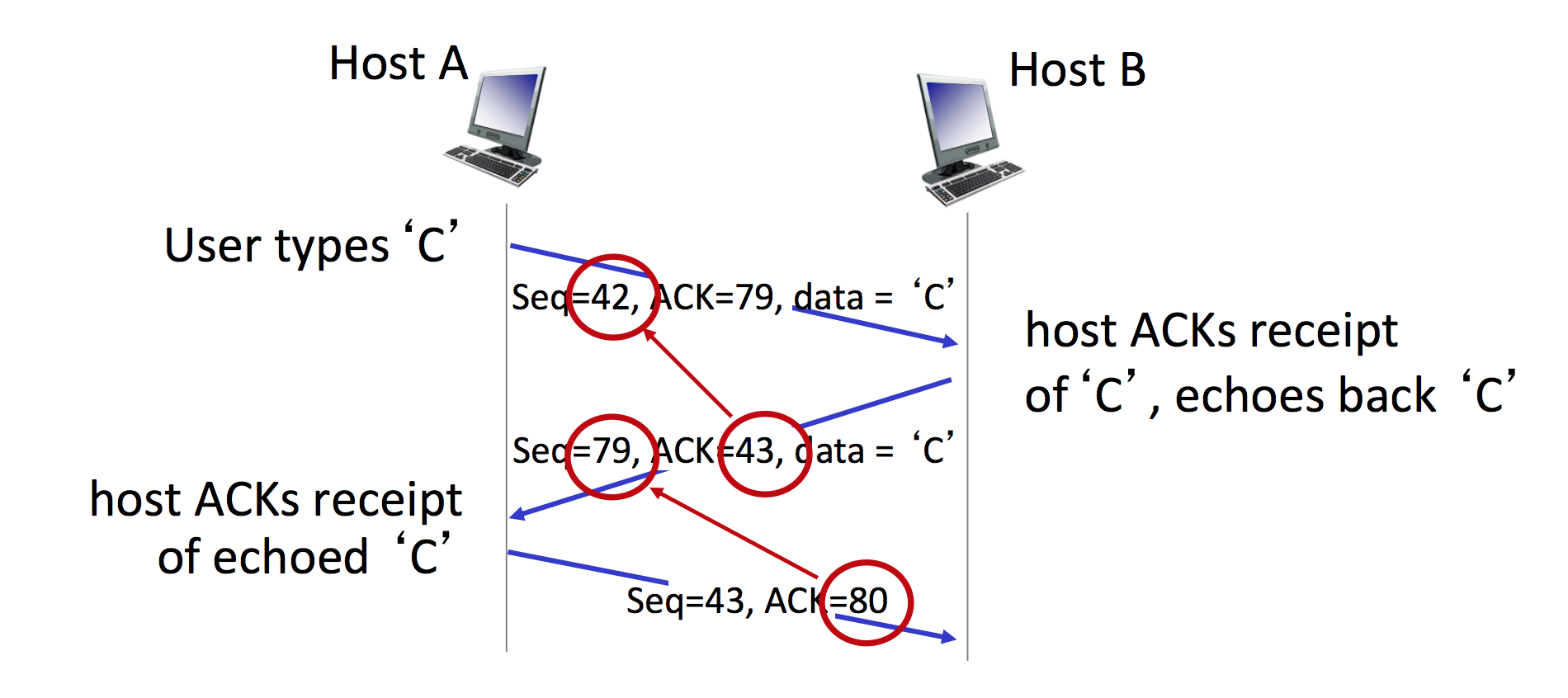
Challenge: What is multi-threading? How can it be used to implement an “always listening” and “always sending” socket for the 2 peers?

Multithreading in python does not actually execute multiple threads simultaneously, but acts as if it does.

Challenge: Which protocol to use?

1. Dropped: Taken care of by retransmitting the dropped packet

The following figure in the slides was basically implemented in my phase 2 code:



1. Damaged: UDP has a checksum by default so I saw no use to implement another checksum
2. Duplicated: The receiver prints duplicate packet if its expected sequence number (which initially, at the start of program execution is set to 0) exceeds the received packet sequence number field “Seq”
3. Delayed: Taken care of by retransmission i.e. sender waits 2-3 seconds (timeout) and retransmits the packet
4. Link Latencies/different bitrates: Same as case of delay

Challenge: Eliminate redundancy by using modules i.e. classes

* The packet module consists of a class called packet that has the following member variables: isACK (true/false), payload (actual data sent), its sequence number, and receiver sequence number.
* The UDP socket module consists of a class called socket that has important class functions created for smooth calling/functionality: create socket and bind to specific port.

Challenge: How to send messages in the form of bytes (as byte stream)?

The pickle library was used to send and receive a byte-stream object through the UDP socket as required. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation.” I found it cleanly replaces the .encode() and .decode() functions.

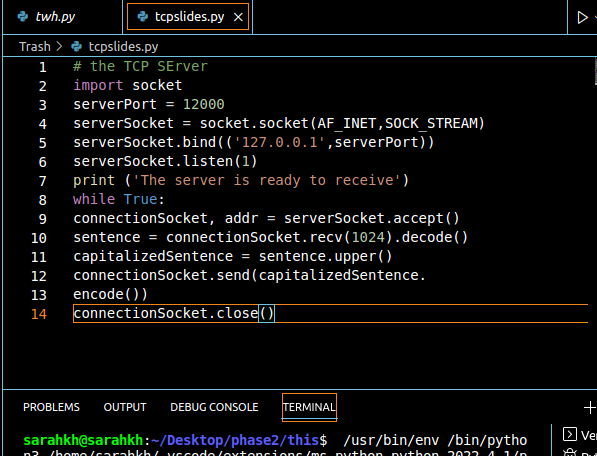
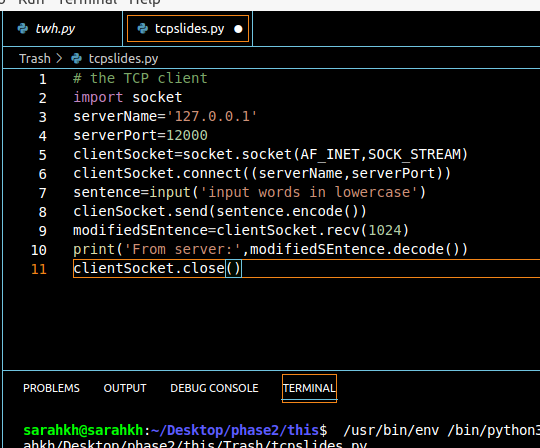
* The client module which uses the packet and UDP socket module: has member variables sequence number, receiver sequence number, receiver IP/port, and expected sequence number. It has other important functions such as functions for handling acks and timeouts, generating random sequence numbers, creating and sending packets etc.
* The server module which uses the packet and UDP socket module has all the member variables of the client module + my\_IP which is used to set its own IP/port number. Its main functions are generating random sequence number, sending packets, sending ACKs, handling expected packets and duplicates etc.

Extra Implementations:

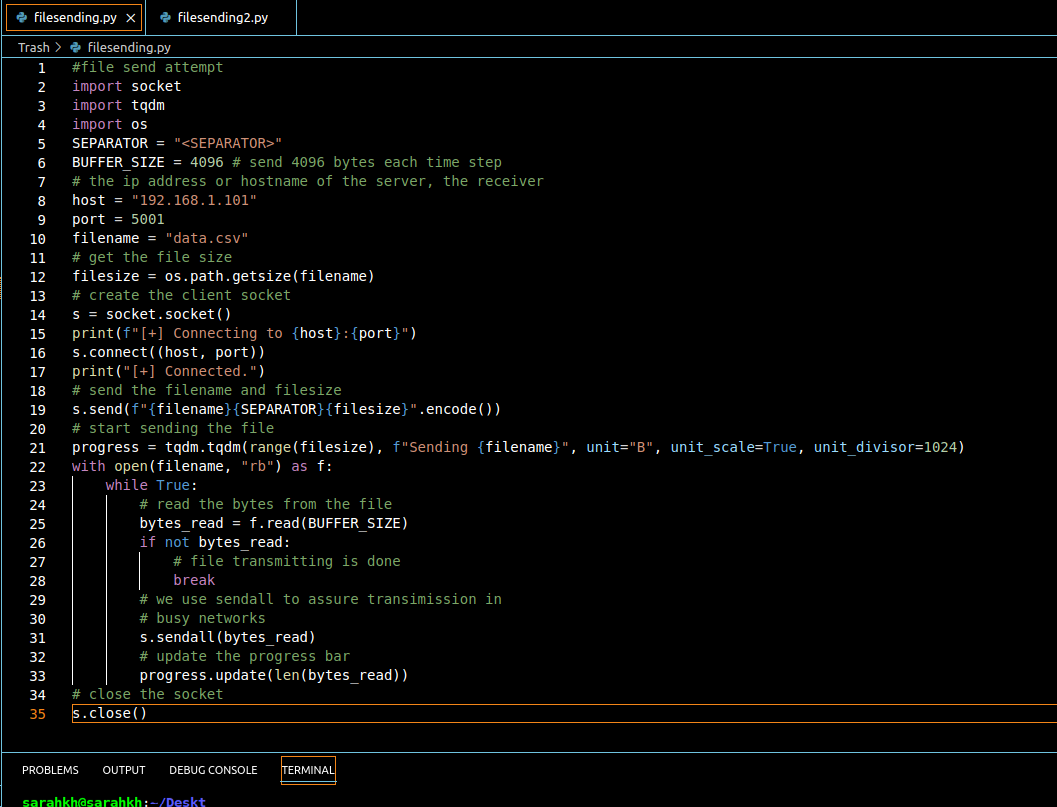
* To keep track of the message records being sent, a file peer\_sent\_messages was created that can store the messages of the peers as txt files
* Each server object has a dictionary which can keep track of its peers and their specific sequence numbers.
* The peers keep track of the records sent and the records received and displays them each time a message is sent (to know how many were lost)

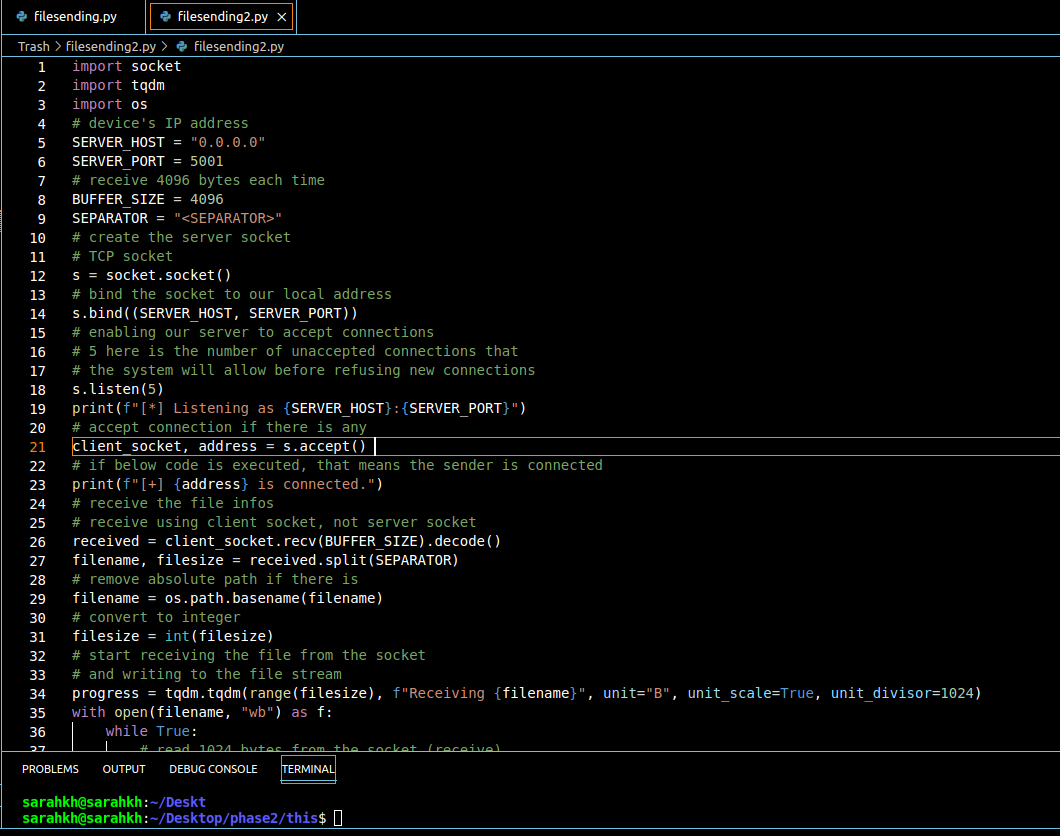
**Phase 3:**

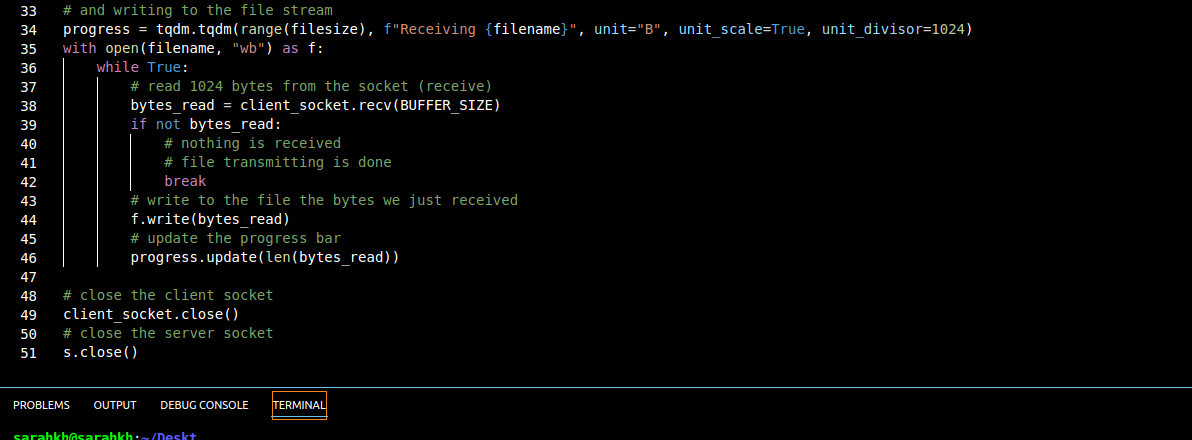
* Our first step in phase three was instantiating a TCP connection using the code given in class slides



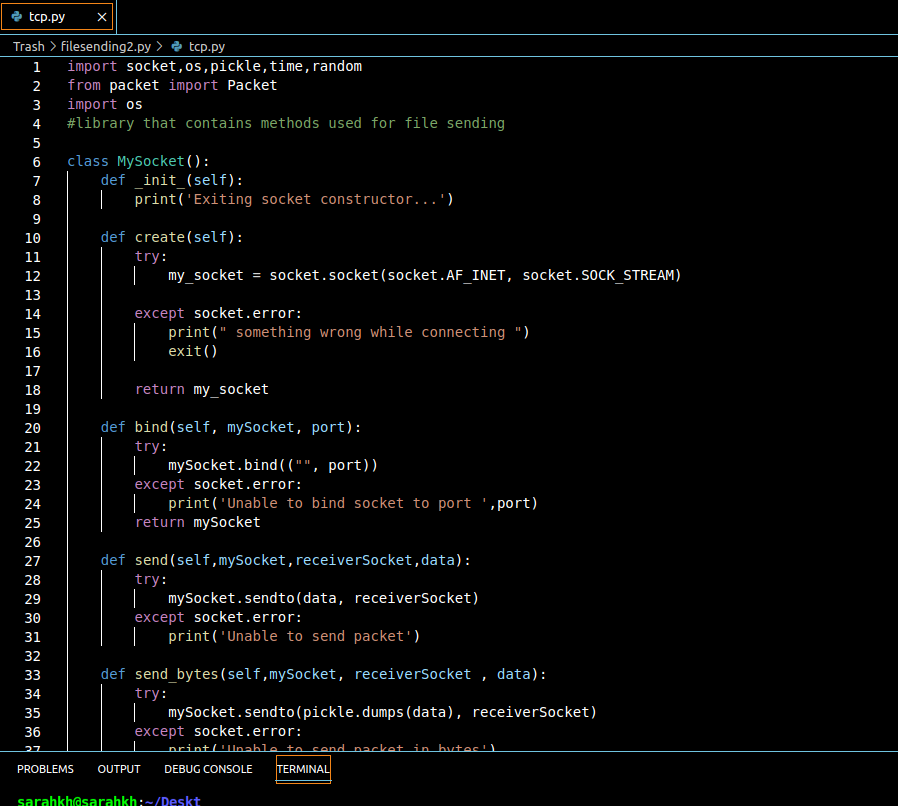
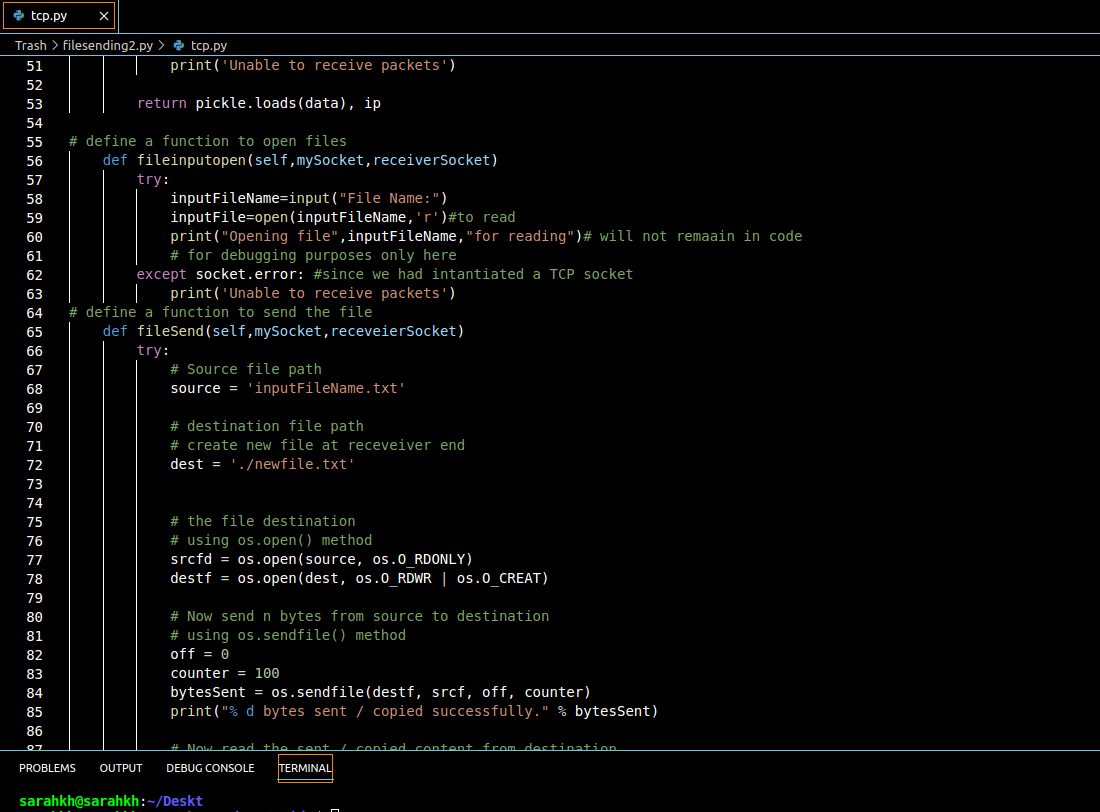
* After that, we tried different file-sending codes on python to get a feel on how to implement the code in this part.



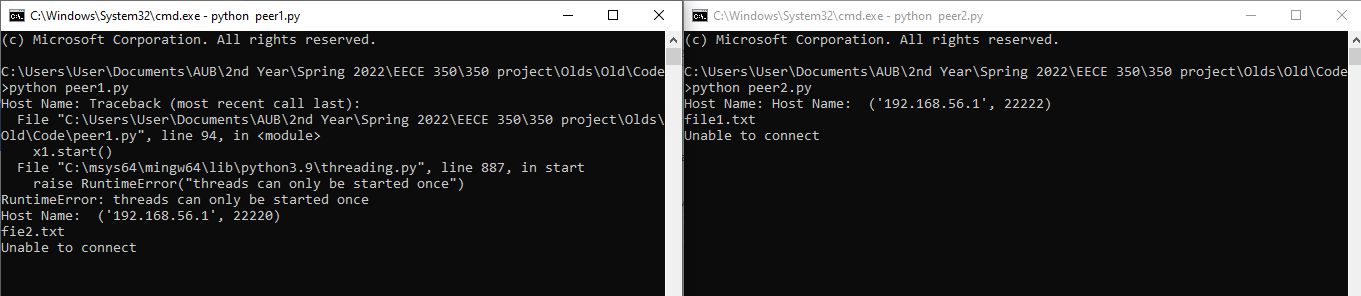
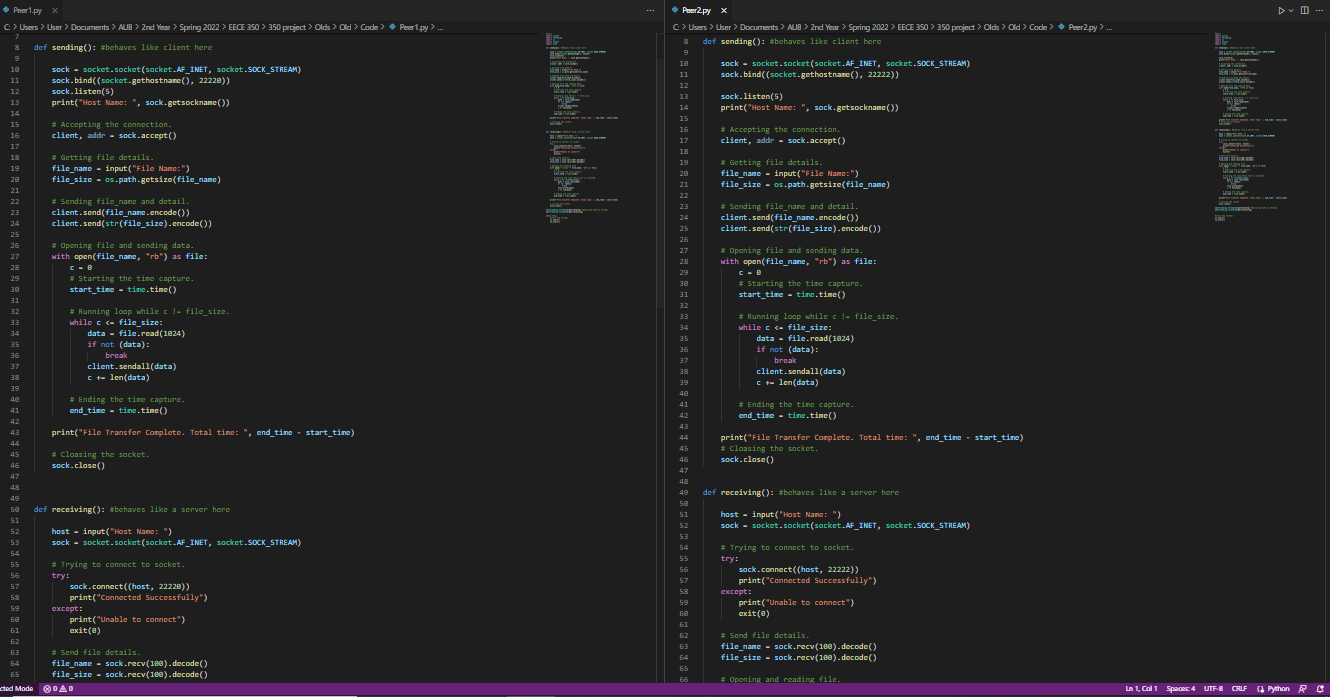




* Given that we had a class-based approach in phase two, we then tried to create a separate python file that had a class responsible for initiating a tcp connection and thus file sending. Then, we attempted to add the tcp class onto the main peer1 and peer 2 code. However, we faced a lot of errors.



* We tried to make a peer2peer file sending-receiving code without messages, and then attempted to combine it in a way with what we got in phase 3, but it did not work.



* When the separate python file method did not work, we implemented a sending file function and a receiving file function in both peers' main files, and we called them inside the client class using if statements to switch between sending messages and sending files. To toggle between messages and files, we used a sentinel type of approach whereby the user inputs a ‘c’ for messages and ‘sr’ for files.

READ ME FILE

The project was built using Python as the program language and Ubuntu 20.4 as the operating system

The

For the application to function dollow the steps below:

1)After downloading the file unzip it.

2)Run the server

3)