



**Maroun Semaan Faculty of
Engineering and Architecture**

Technical Report of Design Project

EECE 350 – Networking

Section 2

Due Date: April 27, 2022

For:

Professor Ayman Tajjedine

By:

Group 17

Tia Al Hajj

Talah Bou Hadir

Houssein Rachini

Bechara Rizk

Table of Contents:

I.	Design Choices Abstract	3
II.	Technical Decisions and Implementations	
i.	Phase 1	3
ii.	Phase 2	4
iii.	Phase 3	5
iv.	Phase 4	6
III.	Challenges	6
IV.	Resolutions	7
V.	Testing and Running Codes	7

Table of Figures:

Figure 1: Phase 1 completed	3
Figure 2: Attributes of the Class "packet" Figure 3: Decoding function	4
Figure 4: Chat Room of GUI	6
Figure 5: Login with username GUI	6
Figure 6: Successful Chatting on Unreliable Network and File Transfer	7

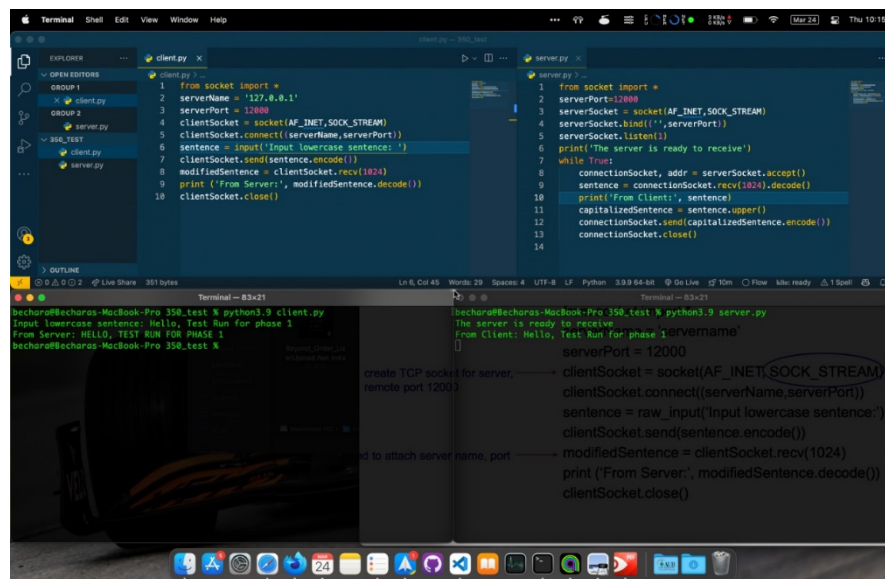
I. Design Choice Abstract:

In the current Design Project, we are required to create and model a chatting application that utilizes python socket programming. The project is distributed into three main phases, and the design choices met the required obstacles placed by the project layout. Phase 2's approach of the team was laying down the attributes of the RDT 3.0 protocol and implementing these into the lines of code of the application layer to achieve reliability and correctness in receiving the data packets. Phase 3 was a direct application of file manipulation depending on the type, and our code supported 10 file types. The bulk of Phase 3 was focused on decapsulating, extracting, and sending the given file over a TCP connection. Finally, for the bonus part of phase 4, we adapted a ready-made code to the chatting application, adding a send file button and usernames displayed as a part of our graphical user interface.

II. Technical Decisions and Implementation

i. Phase 1

Implementing VM software, setting up Ubuntu and adjusting to the technicalities with the new software. On Linux, we downloaded VS Code in order to implement the basic step in establishing a connection between two peers. The code used in Phase 1 was provided from the lecture's notes and we ran it on 4 different laptops of each team member to ensure we all have an up and running system and we have set a functioning environment for the next phases.



```
client.py
1 from socket import *
2 serverName = '127.0.0.1'
3 serverPort = 12000
4 clientSocket = socket(AF_INET, SOCK_STREAM)
5 clientSocket.connect((serverName, serverPort))
6 sentence = input('Input lowercase sentence: ')
7 clientSocket.send(sentence.encode())
8 modifiedSentence = clientSocket.recv(1024)
9 print('From Server:', modifiedSentence.decode())
10 clientSocket.close()

server.py
1 from socket import *
2 serverPort = 12000
3 serverSocket = socket(AF_INET, SOCK_STREAM)
4 serverSocket.bind(('', serverPort))
5 serverSocket.listen(1)
6 print('The server is ready to receive')
7 while True:
8     connectionSocket, addr = serverSocket.accept()
9     sentence = connectionSocket.recv(1024).decode()
10    print('From Client:', sentence)
11    capitalizedSentence = sentence.upper()
12    connectionSocket.send(capitalizedSentence.encode())
13    connectionSocket.close()
```

```
Terminal - 83x21
bechora@Bechoras-MacBook-Pro:~$ python3.9 client.py
Input lowercase sentence: Hello, Test Run for phase 1
From Server: HELLO, TEST RUN FOR PHASE 1
bechora@Bechoras-MacBook-Pro:~$

Terminal - 83x21
bechora@Bechoras-MacBook-Pro:~$ python3.9 server.py
The server is ready to receive
From Client: Hello, Test Run for phase 1
serverName = 127.0.0.1
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence: ')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print('From Server:', modifiedSentence.decode())
clientSocket.close()
```

Figure 1: Phase 1 completed

ii. Phase 2

We divided phase 2 into two files each containing the base, as in the needed elements and, the actions, as in the action of implementation.

File: Udp_base.py

In our code, we created a class “packet” that contains all the attributes. The attributes are username, message, checksum, corrupted and other attributes. The class “packet” are all the variables needed to further implement reliability needed.

The encoding function “def encoding(self):” is used to encode all the contents of the message written by a peer into a byte encoded string, that is send through the socket to then to be decoded. The decode function “def decode(self, encoding): ” where the byte encoded string is decoded back into the class. Also, in the decoding function, if an error was detected the packet is corrupted and then dropped. In addition, the function checks for Checksum, were we use hashing algorithm “md5” returning it to hexadecimal form and comparing it the original checksum given by the sender. This hashing algorithm ensures that the string decoded is unique, hence no errors encountered.

```
import hashlib

class packet:
    def __init__(self):
        self.username="nDef"
        self.message="nDef"
        self.checksum="nDef"
        self.corrupted=False
        self.ack_flag=False
        self.syn_flag=False
        self.ack_nb=0
        self.seq_nb=0
        self.fin_flag=False
        # self.timing=100
        # self.breaker=False
```

Figure 2: Attributes of the Class "packet"

```
def decode(self, encoding):
    try:
        string=encoding.decode()
        data=string.split("\n")
        new=packet()
        new.username=data[0]
        new.message=data[1]
        new.checksum=data[2]
        new.ack_flag=self.str_to_bool(data[3])
        new.syn_flag=self.str_to_bool(data[4])
        new.ack_nb=int(data[5])
        new.seq_nb=int(data[6])
        new.fin_flag=self.str_to_bool(data[7])
        self.corrupted=not(self.verify_checksum(encoding))
        return new
    except: #any error while decoding says that the packet is corrupted
        self.corrupted=True
    # except:
    #     pass
```

Figure 3: Decoding function

File: Udp_actions.py

The code contains, whether the packet was correctly received or not, and assigns a randomized sequence number. This randomized sequence number is unique for the specific message, in this certain mechanism we guarantee that if an ACK wasn't received, in the retransmission process the unique sequence number that was sent by the sender stays the same. Also, we have a variable timeout function, that increases the timeout interval every retransmission.

File: Peer1.py and Peer2.py

(These files aren't included in submission folder but are a detailed step to describe the work process, but are included in the Main Script)

The two python files are the main files that contain both of the imported "udp_actions.py" and the "udp_base.py".

iii. Phase 3

File: tcp_actions.py

- File Sender: def file_sender(receiverName, receiverPort)

We create a TCP socket and connect to the receiver. Next, we ask the peer to input the file name, and then the application checks if the file exists in the current working directory. Then the file is opened to be read as binary, and the sending process through the socket occurs by chunks, when the file is completely sent, the files is closed

- File Receiver: def file_receiver(receiverName, receiverPort)

We create a socket to constantly listen, when we receive a connection request, we accept it creating an empty file. The data received from the connection socket is then written in binary mode in the empty file.

Also, using Magic library we determine the received file type and then renaming the "new_file" to its correct extension (.pdf, .txt , etc..). Then we call the receive_file to remain within the existing socket listening for new connections.

iv. Phase 4

We took a GUI from the internet, but we had to alter the code to fit the project design assigned to us.

First, it creates a window to enter the username of the peer which is then used to by the other peer to communicate and identify the peer on the other end.

Second, it creates the chat room window which contains a text field which displays the messages received, and a text box to write the peer's messages. Also, it contains a send and send file button for user use.

In order to enable the server functionality of our code, we used threading and combined both phase 2 and 3's codes. We started to receive both UDP and TCP packets. Consequently, when the user sends a message, it's being sent through UDP; however, when we are sending any type of file, we use the TCP socket.

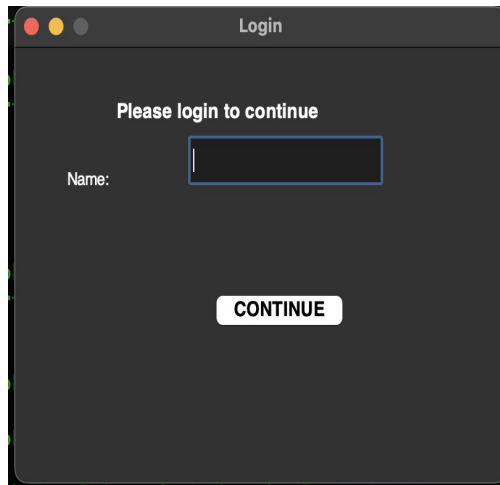


Figure 5: Login with username GUI

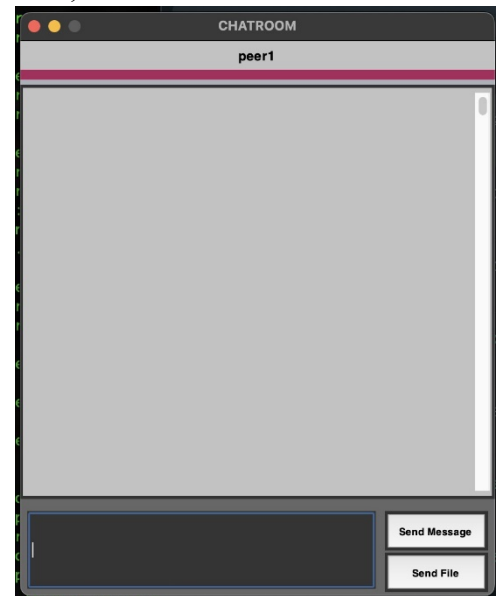


Figure 4: Chat Room of GUI

III. Challenges

We faced challenges in the course of the project, spanning from Phase 1 up to Phase 4. The challenges we faced were:

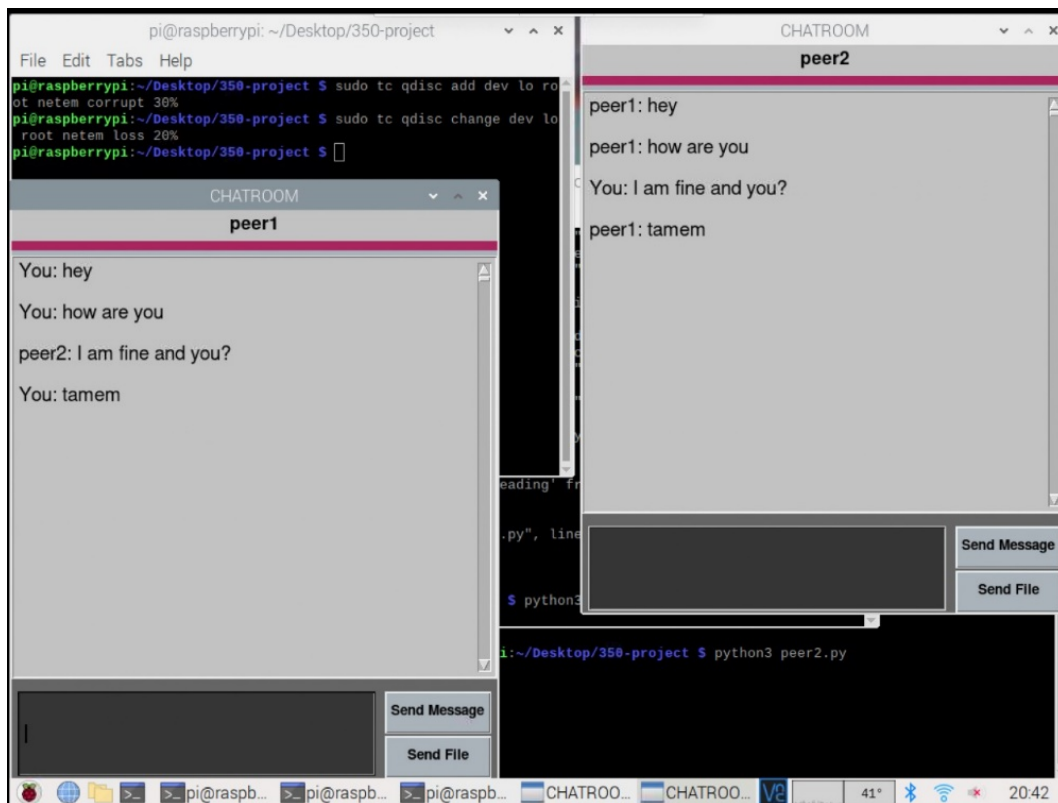
1. VM Virtual Box was glitching with one of our team members, and Ubuntu's launching failed many times
2. Creating a timeout function, that updates accordingly was a challenging task
3. The receiver of the file in phase 3 wasn't differentiating the type of file being sent, and it remained in binary mode
4. The GUI syntax of tkinter was not fully familiar with any of the team members, and customizing the color scheme with the appearance

IV. Resolutions

To resolve the issues, we took the following steps:

1. We installed and uninstalled countless times, until the bug was fixed, and Ubuntu was up and running
2. After doing some research, we discovered that there was an existing built-in function that facilitated timeout
3. We used the Magic library in our python code, which can differentiate each file type at the receiver's end
4. Found a suitable online code that can be used and customized to fit our code

V. Testing and Running Codes



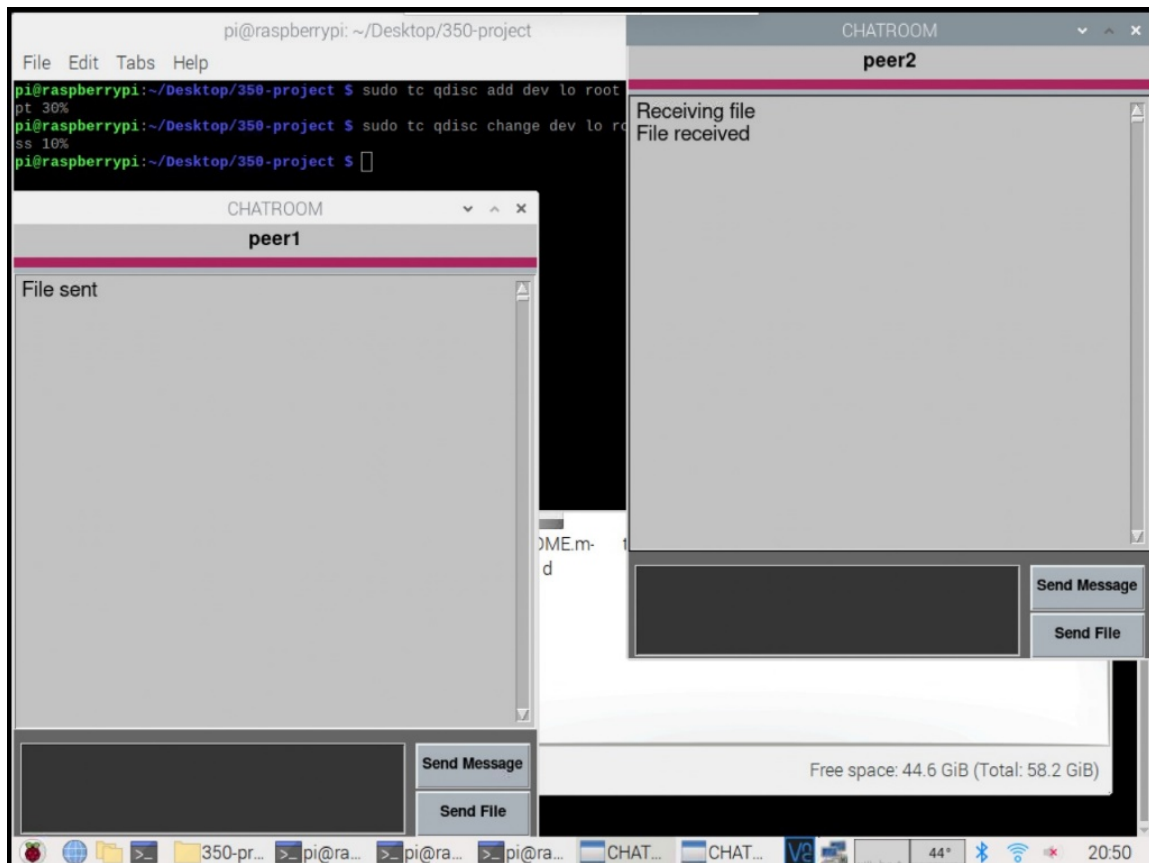


Figure 6: Successful Chatting on Unreliable Network and file Transfer