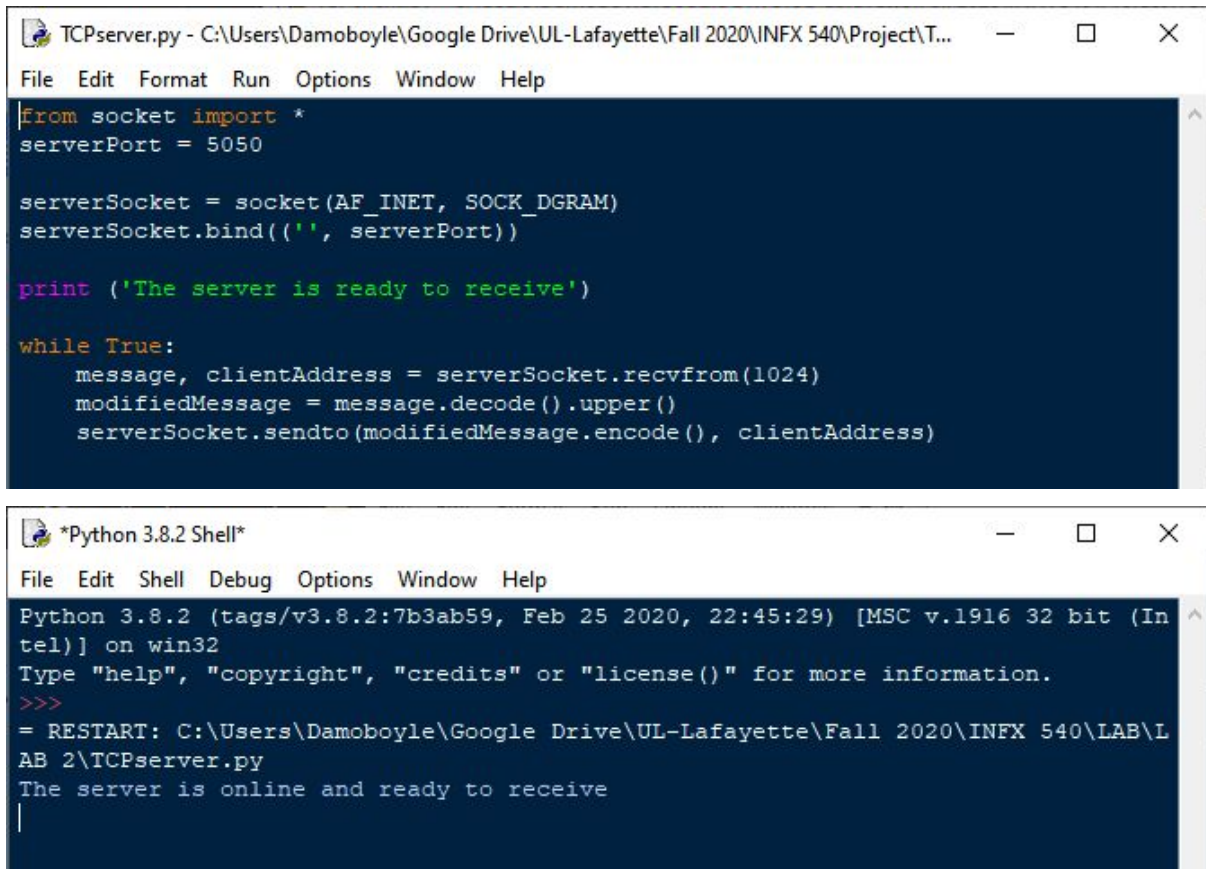


Part 1

Client–Server Communication using TCP/IP

A) Server



The image shows two windows from a Python IDE. The top window, titled 'TCPserver.py', contains the following Python code:

```
from socket import *
serverPort = 5050

serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))

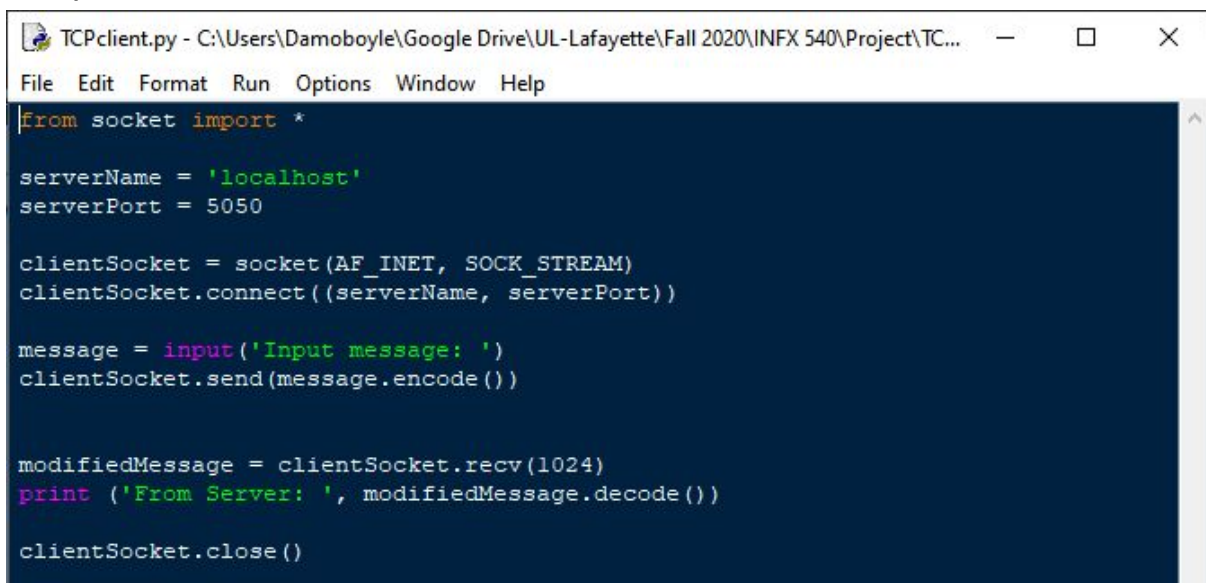
print('The server is ready to receive')

while True:
    message, clientAddress = serverSocket.recvfrom(1024)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

The bottom window, titled '*Python 3.8.2 Shell*', shows the terminal output:

```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\Damoboyale\Google Drive\UL-Lafayette\Fall 2020\INFX 540\LAB\LAB 2\TCPserver.py
The server is online and ready to receive
|
```

B) Client



The image shows a single window from a Python IDE, titled 'TCPclient.py', containing the following Python code:

```
from socket import *

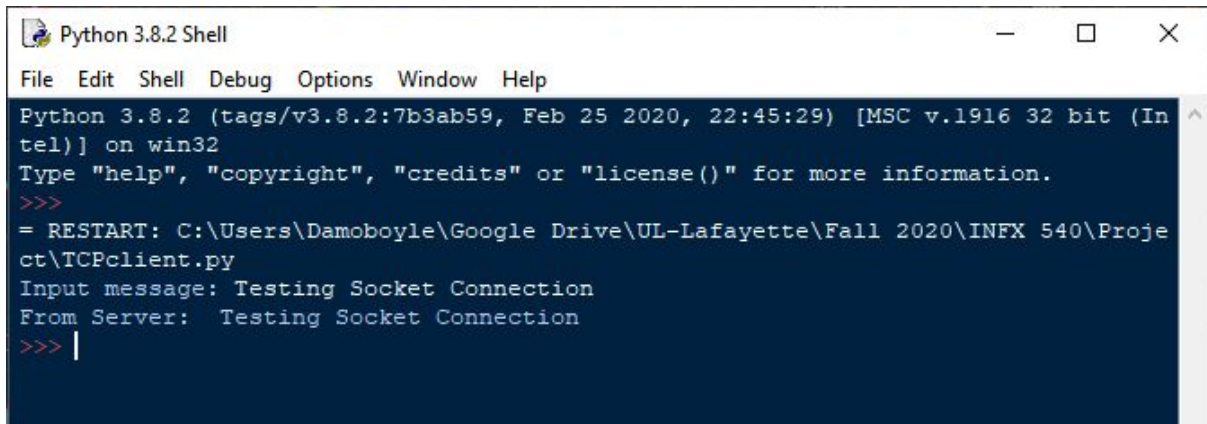
serverName = 'localhost'
serverPort = 5050

clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

message = input('Input message: ')
clientSocket.send(message.encode())

modifiedMessage = clientSocket.recv(1024)
print('From Server: ', modifiedMessage.decode())

clientSocket.close()
```



```
Python 3.8.2 Shell
File Edit Shell Debug Options Window Help
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\Damoboyale\Google Drive\UL-Lafayette\Fall 2020\INFX 540\Project\TCPclient.py
Input message: Testing Socket Connection
From Server: Testing Socket Connection
>>> |
```

Part 2

Advanced Client

```
AdvancedClient.py - C:\Users\Damoboy\Google Drive\UL-Lafayette\Fall 2020\INFX 540\Project\AdvancedClient.py (3.8.2)
File Edit Format Run Options Window Help
from socket import *
from select import *
from threading import Thread
import tkinter

BUFSIZE = 1024

serverName = 'localhost'
serverPort = 5050

clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

def receive():
    while True:
        try:
            msg = clientSocket.recv(BUFSIZE).decode()
            msgList.insert(tkinter.END, msg)
        except OSError:
            break

def send(event = None): # event passed by binders
    msg = message.get()
    message.set('')
    clientSocket.send(msg.encode())

    if msg == '{quit}':
        clientSocket.send(msg.encode())
        clientSocket.close()
        app.destroy()

def close(event = None):
    message.set('{quit}')
    send()

app = tkinter.Tk()
app.title('ClassChat')

messageFrame = tkinter.Frame(app)
scrollbar = tkinter.Scrollbar(messageFrame)

msgList = tkinter.Listbox(messageFrame, height = 25, width = 60, yscrollcommand = scrollbar.set, bg = 'black', fg = 'white')

messageFrame.pack()
scrollbar.pack(side = tkinter.RIGHT, fill = tkinter.Y)
msgList.pack(side = tkinter.LEFT, fill = tkinter.BOTH)

#InputBar
message = tkinter.StringVar() #Needed for sending messages
entry = tkinter.Entry(app, textvariable = message, width = 50, bg = 'white')
entry.bind('<Return>', send)
entry.pack(padx = 5, side = tkinter.LEFT)

#Send Button
button = tkinter.Button(app, text = 'Send', command = send, width = 10, bg = 'blue', fg = 'white')
button.pack(padx = 5, pady = 10)

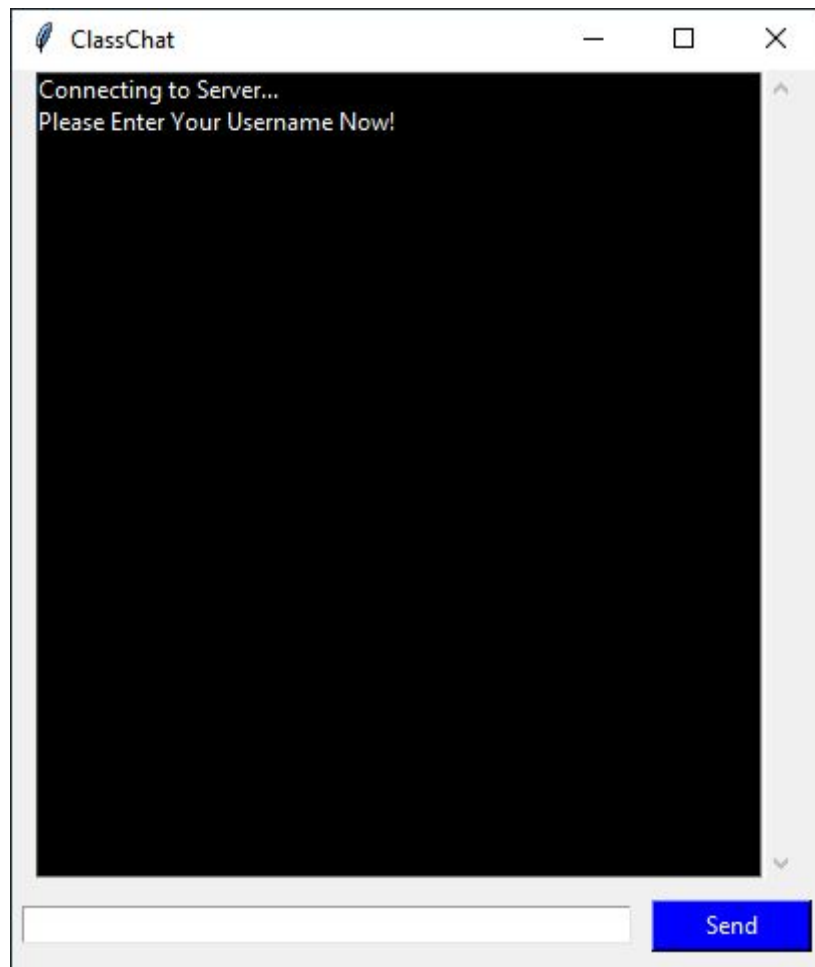
app.protocol('WM_DELETE_WINDOW', close)

receiveThread = Thread(target = receive)
receiveThread.start()

#Starts GUI
tkinter.mainloop()
```

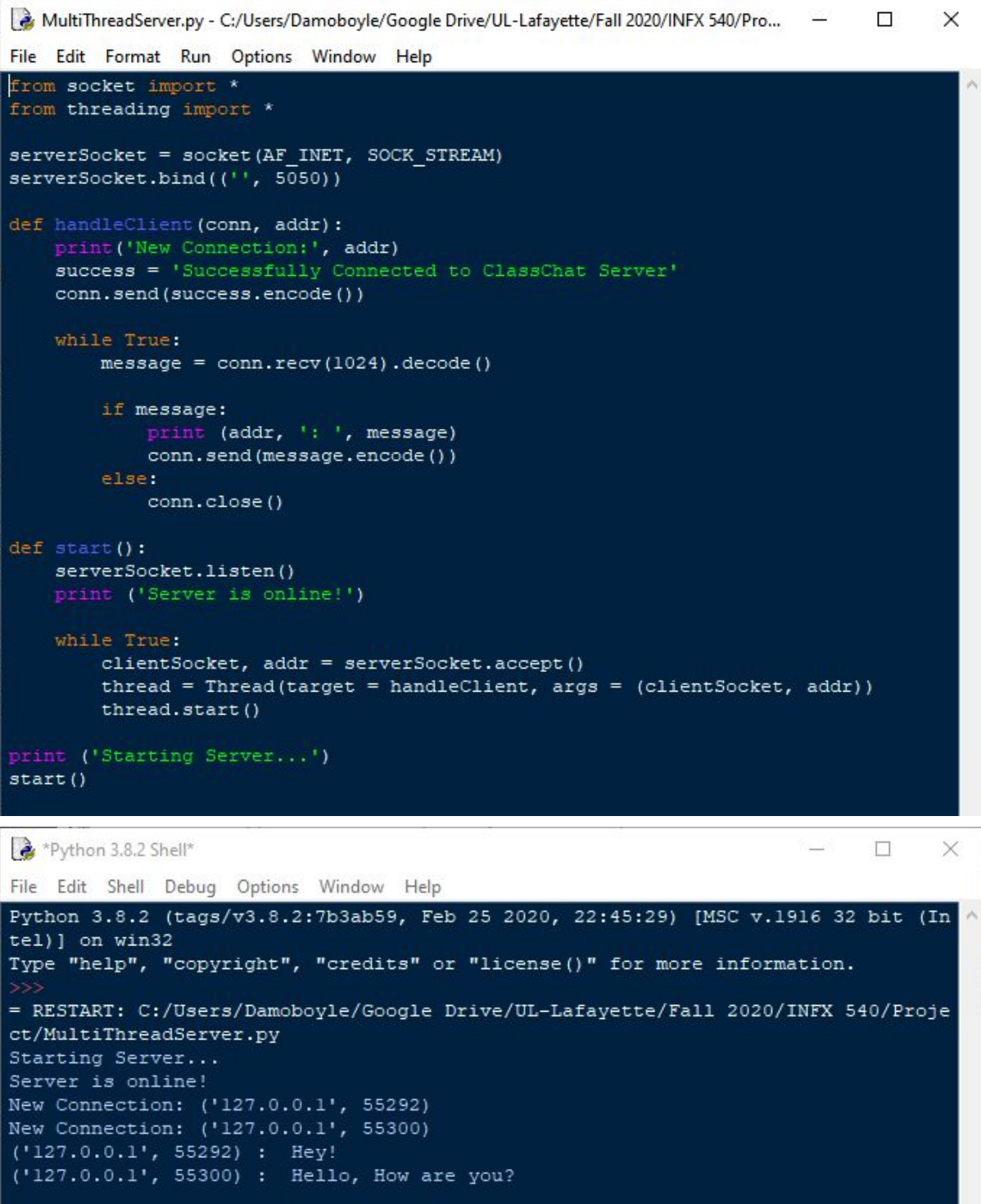
In order to implement the required functionality of the Advanced Client to be able to both send a receive data at the same time, I simply built a GUI interface for the client which allows the user to enter text at their leisure without blocking the other processes of the program, as was the case in the first iteration of the client. I then implemented a thread in order to continuously listen for incoming messages from the server socket.

GUI Window for Advanced Client



Part 3

Multi-Thread Communication Server



```
MultiThreadServer.py - C:/Users/Damoboyle/Google Drive/UL-Lafayette/Fall 2020/INFX 540/Pro...
File Edit Format Run Options Window Help

from socket import *
from threading import *

serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', 5050))

def handleClient(conn, addr):
    print('New Connection:', addr)
    success = 'Successfully Connected to ClassChat Server'
    conn.send(success.encode())

    while True:
        message = conn.recv(1024).decode()

        if message:
            print(addr, ': ', message)
            conn.send(message.encode())
        else:
            conn.close()

def start():
    serverSocket.listen()
    print('Server is online!')

    while True:
        clientSocket, addr = serverSocket.accept()
        thread = Thread(target = handleClient, args = (clientSocket, addr))
        thread.start()

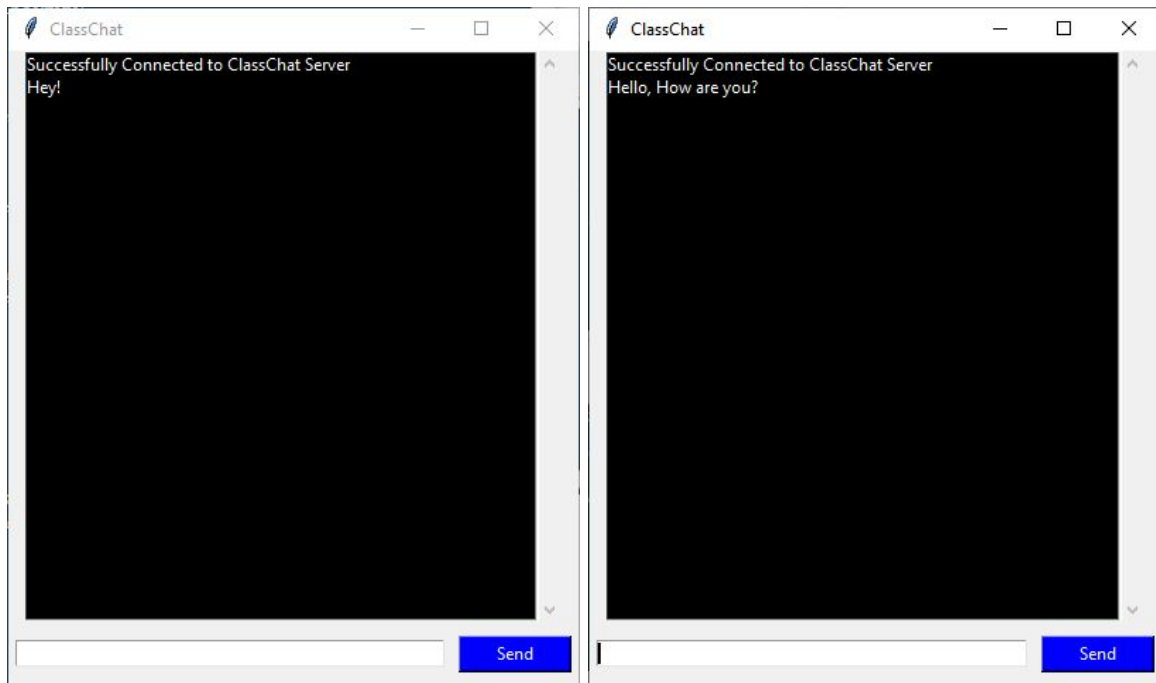
print('Starting Server...')
start()
```

```
*Python 3.8.2 Shell*
File Edit Shell Debug Options Window Help

Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/Damoboyle/Google Drive/UL-Lafayette/Fall 2020/INFX 540/Project/MultiThreadServer.py
Starting Server...
Server is online!
New Connection: ('127.0.0.1', 55292)
New Connection: ('127.0.0.1', 55300)
('127.0.0.1', 55292) : Hey!
('127.0.0.1', 55300) : Hello, How are you?
```

I solved the multiple concurrent connection problem through the use of threads. Now multiple users can all connect the server at the same time.

Respective GUI Interfaces



Part 4

Client-Client Communication

```
server.py - C:/Users/Damoboye/Google Drive/UL-Lafayette/Fall 2020/INFX 540/Project/server.py (3.8.2)
File Edit Format Run Options Window Help
1 from socket import *
2 from threading import *
3 from tkinter import filedialog
4 import time
5
6 BUFSIZE = 1024
7
8 serverSocket = socket(AF_INET, SOCK_STREAM)
9 serverSocket.bind(('', 5050))
10
11 history = []
12 sender = []
13 clientList = []
14 addresses = []
15
16 def handleClient(conn, addr):
17     #Welcome Messages
18     conn.send(bytes('Connecting to Server...', 'utf8'))
19     time.sleep(1)
20     conn.send(bytes('Please Enter Your Username Now!', 'utf8'))
21
22     username = conn.recv(BUFSIZE).decode().lower()
23     addresses.append(conn)
24     clientList.append(username)
25
26     #Handles immediate client quit
27     if username != '{quit}':
28         conn.send(bytes(f'Welcome [{username}]', 'utf8'))
29         conn.send(bytes('You have Successfully Connected to ClassChat Server', 'utf8'))
30         new = f'New Connection: [{username}]'
31         print(new, addr) #Server Record
32         broadcast(new, 'server')
33
34     else:
35         conn.close()
36         addresses.remove(conn)
37         clientList.remove(username)
38
39 while True:
40     try:
```

```
41     message = conn.recv(BUFSIZE).decode()
42
43     if message != '{quit}':
44         history.append(message)
45         sender.append(username)
46
47         chat = message.partition(':')
48         if chat[1] != ':':
49             conn.send(bytes('Message not sent, No recipirent!', 'utf8'))
50             conn.send(bytes(' (Format) username:message ', 'utf8'))
51         else:
52             send(chat[0], chat[2], username, conn)
53             conn.send(message.encode())
54
55             print (f'[{username}]', message)
56
57             #Logs user out of the server
58             else:
59                 addresses.remove(conn)
60                 clientList.remove(username)
61     except:
62         conn.close()
63
64 def send(to, msg, username, you):
65     private = f'[{username}] {msg}'
66
67     for user in clientList:
68         if user == to:
69             addresses[clientList.index(to)].send(private.encode())
70             break
71         else:
72             you.send(bytes('!!Headsup!!', 'utf8'))
73             time.sleep(.1)
74             you.send(bytes(f'User:{to} is not online right now', 'utf8'))
75             time.sleep(.1)
76             you.send(bytes('We will send this message when they come online', 'utf8'))
77
78 def start():
79     serverSocket.listen()
80     print ('Server is online!')
```

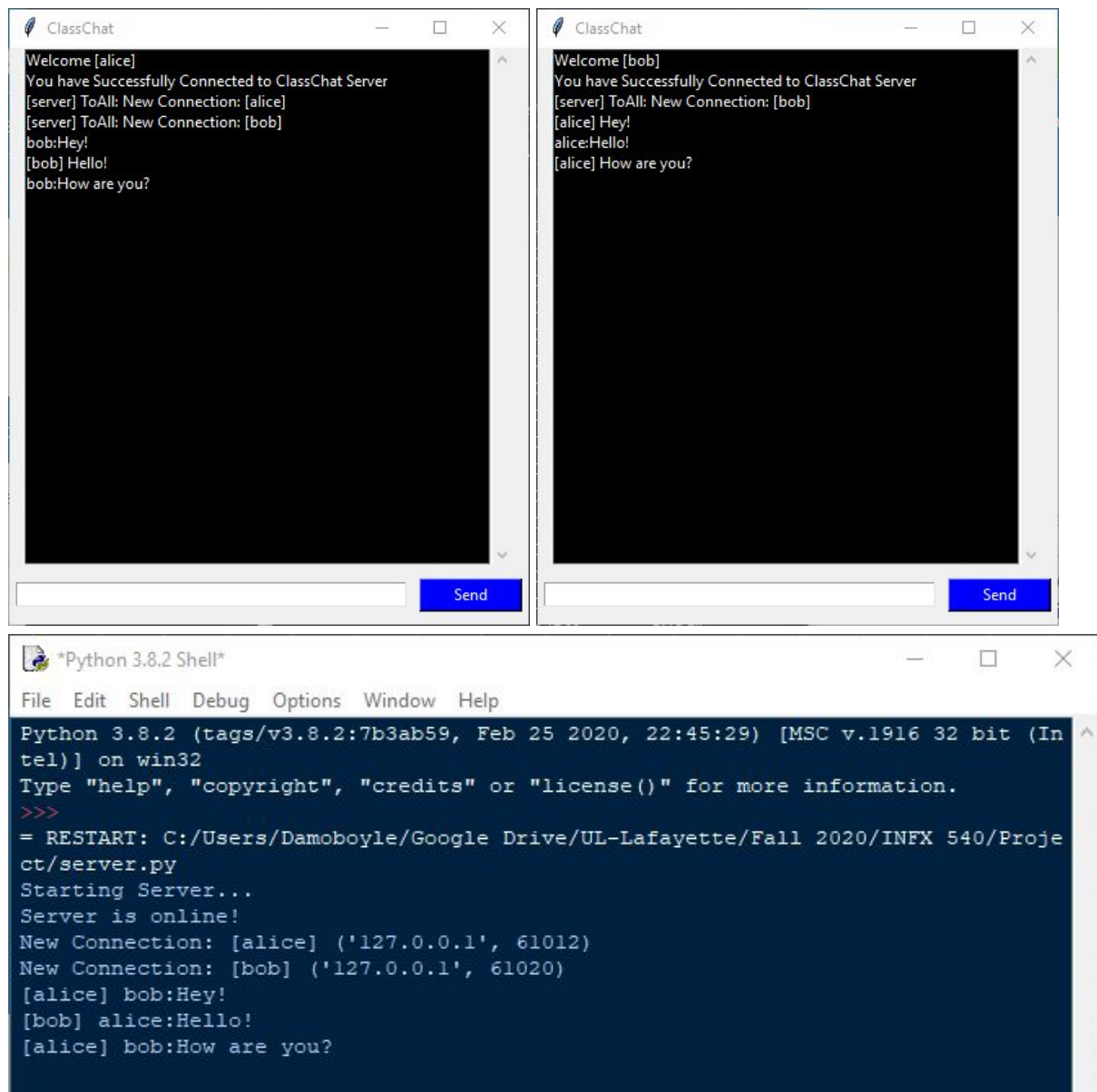
```
82     while True:
83         clientSocket, addr = serverSocket.accept()
84         thread = Thread(target = handleClient, args = (clientSocket, addr))
85         thread.start()
86
87 print ('Starting Server...')
88 start()
```

Ln: 42 Col: 0

The issues of Client Management, Receiving Client Messages and Forwarding Client Messages are all solved in this updated iteration of the server.

Clients are managed through the use of lists to keep track of new users logging in and logging off.

Formatting rules were applied based on expected and required input in order to facilitate receiving and then allowing for forwarding of messages to the desired recipient.



Json

Capturing data for json formatting with the client is pretty simple. In my program the sender's information is all stored and handled on the server side so the only relevant information to format is receiver and message. For example;

```
{"To":username, "message":a message to a client}
```

In the code it was easy to switch my original working program over to a json, rather than just sending the exact user input for the server to handle.

```
msgjson = json.loads(json.dumps({"To":msg.partition(":")[0],  
    "message":msg.partition(":")[2]}))
```

This converts the data to json format, unfortunately json objects can't be passed using sockets in python but this actually makes the job of changing the code easier.

See line 27 in AdvancedClient.py

```
clientSocket.send(msg.encode())
```

Is changed to;

```
msgjson = {"To":msg.partition(":")[0],  
    "message":msg.partition(":")[2]}  
  
clientSocket.send(msgjson)
```

There isn't actually any significant gain from capturing and passing the information in this fashion in my program. In fact on the server side it just caused more headaches than it solved, due to the way my code was implemented to receive and handle user input. While json formatted data would allow for easier saving and exportation of data as well as aid scalability, this is of no concern to me for this program as it is hosted locally on my personal computer. For these reason I decided to revert to my original code as it does the exact same job and loses nothing in processing power as the extra workarounds that would have been required to make the json data functional would have negated any advantage this data format would have had over the current operational state of my program.

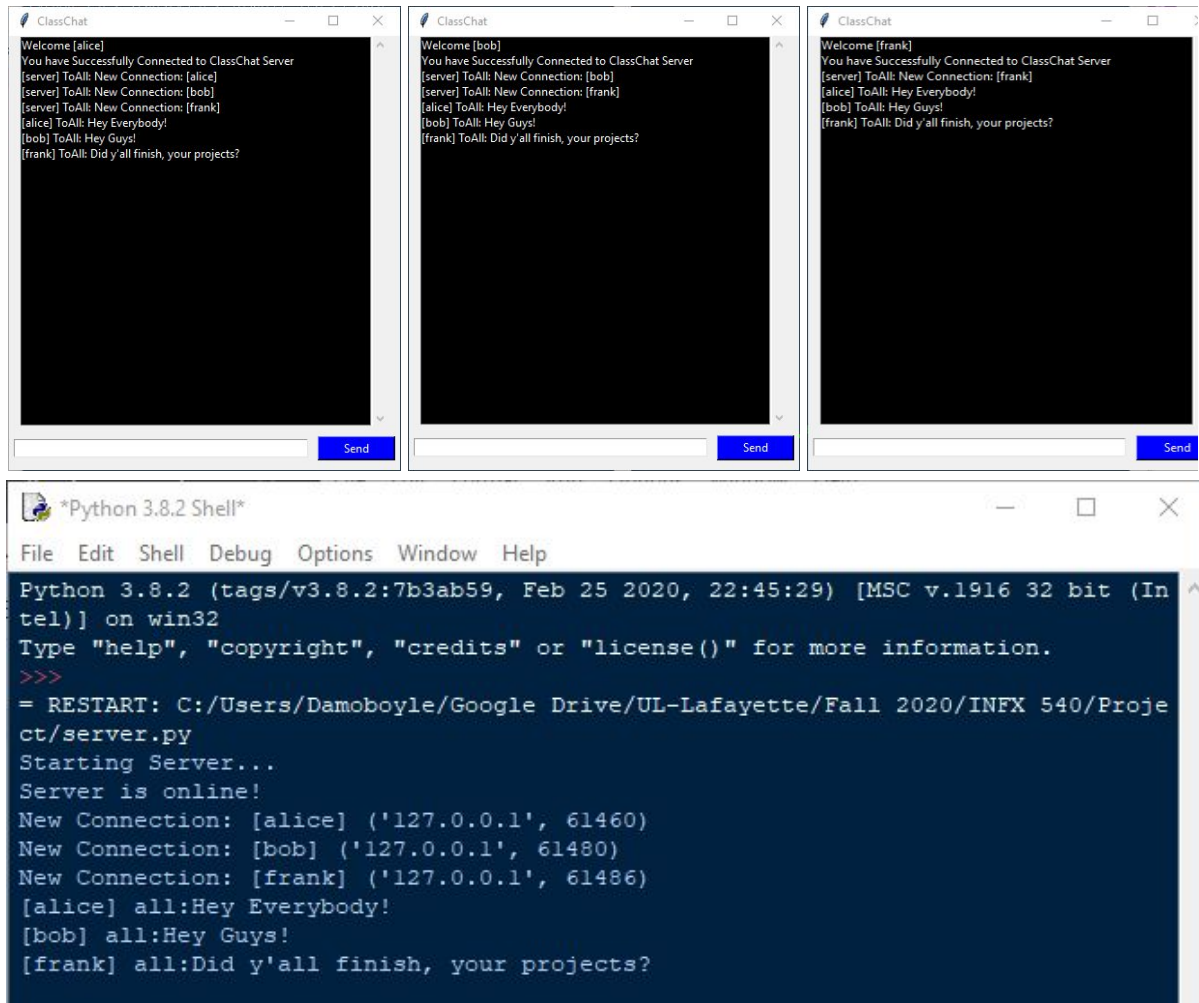
On the server side rather than partitioning the intact formatted message from the user, json would have allowed for quicker distribution of this data to its relevant definitions, however the exceptions to the typically expected input made it very difficult to handle the json formatted data compared to fully intact message from the user.

Part 5 Bonus

A) Group Chatting

This feature was easily added on the server program with a definition to handle sending messages to every user rather than just one individual in particular. For ease of formatting I simply designed this feature to teach the username 'all' indicator to distribute the message to all users on the chat server. Private and group messaging occurs in the same window.

```
60     while True:
61         try:
62             message = conn.recv(BUFSIZE).decode()
63
64             if message != '{quit}':
65                 history.append(message)
66                 sender.append(username)
67
68                 chat = message.partition(':')
69                 if chat[1] != ':':
70                     conn.send(bytes('Message not sent, No recipirent!', 'utf8'))
71                     conn.send(bytes('(Format) username:message ', 'utf8'))
72
73                     elif chat[0].lower() == 'all':
74                         broadcast(message['message'], username)
75
76                     else:
77                         send(chat[0], chat[2], username, conn)
78                         conn.send(message.encode())
79
80                         print (f'[{username}]', message)
81
82                         #Logs user out of the server
83                     else:
84                         addresses.remove(conn)
85                         clientList.remove(username)
86         except:
87             conn.close()
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106 #BONUS#
107 #Group Chat
108 def broadcast(msg, you):
109     group = f'[{you}] ToAll: {msg}'
110
111     for client in addresses:
112         client.send(group.encode())
113
114
115
116
117
118
119
120
121
122
123
```



The image displays three separate 'ClassChat' application windows and a 'Python 3.8.2 Shell' window. Each 'ClassChat' window shows a chat log for a specific user (alice, bob, or frank) and a 'Send' button at the bottom. The 'Python 3.8.2 Shell' window shows the command prompt output for running 'server.py', including connection logs for the three users and their messages.

```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/Damoboyale/Google Drive/UL-Lafayette/Fall 2020/INFX 540/Project/server.py
Starting Server...
Server is online!
New Connection: [alice] ('127.0.0.1', 61460)
New Connection: [bob] ('127.0.0.1', 61480)
New Connection: [frank] ('127.0.0.1', 61486)
[alice] all:Hey Everybody!
[bob] all:Hey Guys!
[frank] all:Did y'all finish, your projects?
```

B) File Transfer

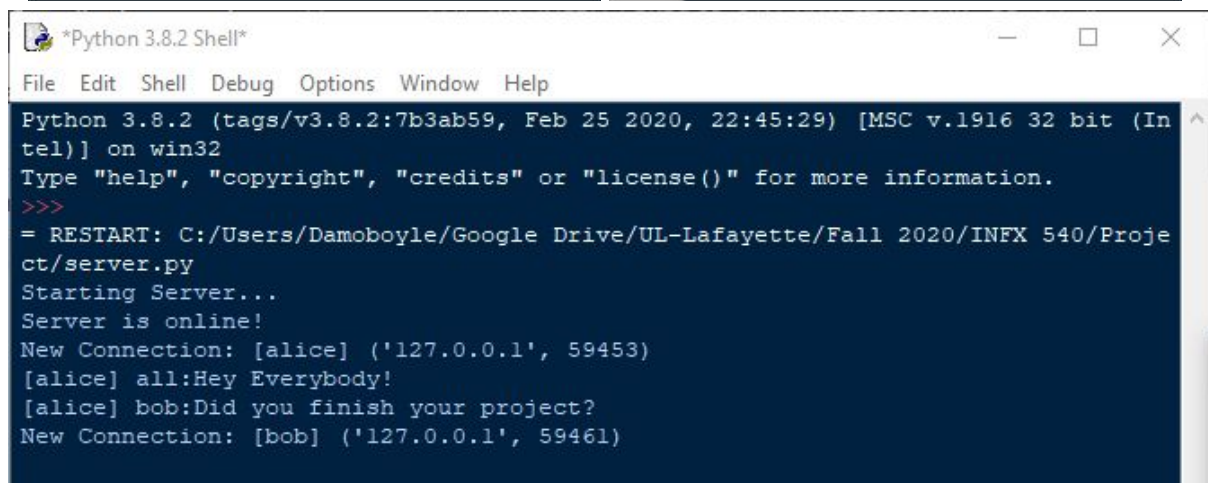
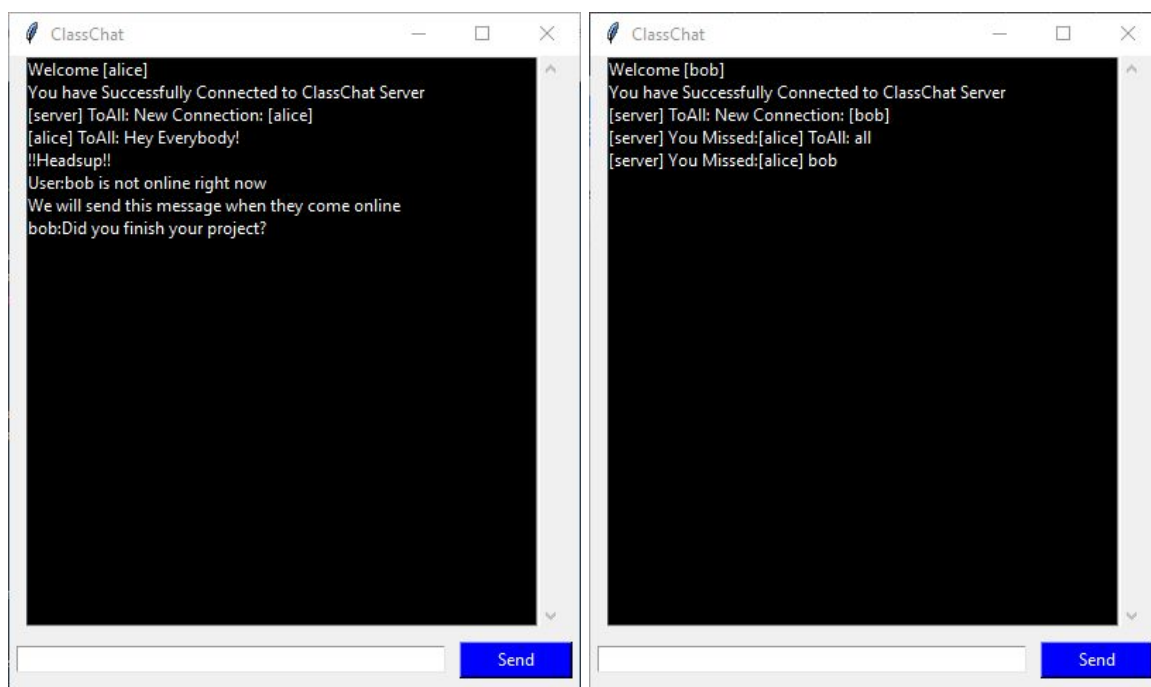
C) Offline Messaging

This feature was easily implemented using lists to store messages and their senders/recipients as they are sent. Then when a new user joins the chat server this list is probed using a for loop to determine if any messages were intended for the new user. They are also sent all of the messages they missed in the group chat.

```

49 #BONUS#
50 #Receive Missed Messages
51 for missed in history:
52     if username == missed.partition(':')[0]:
53         missedMSG = f"[server] You Missed: [{sender[history.index(missed)]]} {missed.rpartition(':')[0]}"
54         conn.send(missedMSG.encode())
55
56     elif missed.partition(':')[0] == 'all':
57         missedToAll = f"[server] You Missed: [{sender[history.index(missed)]]} ToAll: {missed.rpartition(':')[0]}"
58         conn.send(missedToAll.encode())
59

```



D) Encryption/Decryption