<u>Chat Program Project Report</u>

Submitted By:

Brandon Hang

Course Title: **ECE 146 Computer Networks**

Date Submitted: **May 5, 2023**

Table of Contents

1. Introduction

An instant messaging application uses technology to enable real-time communication between two or more connected users over the Internet. Users can exchange text messages and these systems are most commonly used for personal communication between friends and family, or even business communications with colleagues. In this project, a client-server chat application will be demonstrated to show the process of connecting to a server, broadcasting messages to connected clients, and disconnecting from the server.

There are two programs the server application, which is hosted on a server, and the client which is on the local PC. The server is to create a socket, binds it, accepts requests, receives and sends messages, and closes the connection. The client, on the other hand, creates a socket, connects to the server, sends and receives messages, and closes the socket. The server will be able to maintain an unlimited number of open client connections, and the client will have a graphical user interface to be able to view and send messages to the server.

The applications will be written in Python and with the help of the many available Python modules, the program will be able to work as intended with the consideration of errors.

2. Objectives

The objective of creating a chat program is to enable 2 or more users to communicate with each other. Understanding how TCP connections are utilized with the help of sockets will provide a reliable connection between the client and server. Threading is also involved in order to maintain multiple connections to the server.

3. Procedure

By creating a Client-Server application written in Python, connected users using the client script will be able to communicate with each other through a graphical user interface. When developing the application, a TCP socket connection will be initiated with the server to ensure a reliable connection. In order to connect, the user will provide a local IP address and a port number of the server. When the user/users are connected, they will be able to send messages by having the server receive the messages and rebroadcast them to the other connected clients. When a user wants to disconnect, they will send a message to the server indicating that they are going to exit the program and closes the socket connection. When the server is done accepting connections, it will close the socket. The steps and procedure of a server-client program can be shown in Figure 1.

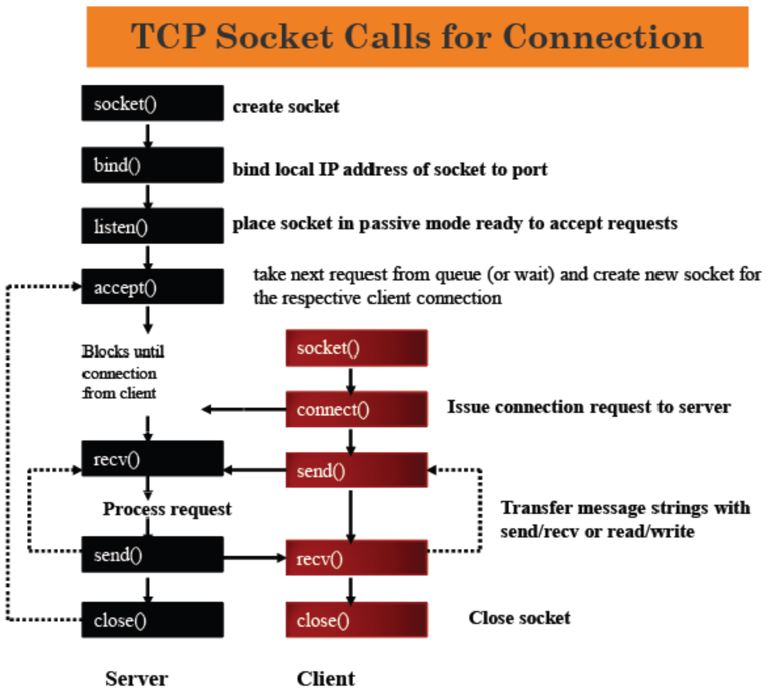

TCP Socket Calls for Connection
socket()
create socket
bind()
bind local IP address of socket to port
listen()
place socket in passive mode ready to accept requests
accept()
take next request from queue (or wait) and create new socket for the respective client connection
Blocks until connection from client
socket()
connect()
Issue connection request to server
recv()
send()
Process request
Transfer message strings with send/recv or read/write
send()
recv()
close()
close()
Close socket
Server
Client

Figure 1. Multi-User Chat Program TCP Socket Procedure

Both the server and the client will also implement multi-threading in order to maintain a connection to the server and when receiving messages so without requiring multiple copies of the program when running. This can all be done with external modules which will be demonstrated in the code section.

4. <u>System Analysis</u>

Chat application establishes a connection between 2 or more systems within a network. For demonstration purposes, the client program will be hosted on a local computer, and the server program will be deployed onto a separate virtual machine.

4.1. <u>System Architecture</u>

There will be at least 1 server and 1 client. While there are other approaches to creating a more reliable and available chatting program architecture, this is what I will be using as it is what I have in my home network. Figure 2.0 shows the current network architecture of the chatting program.

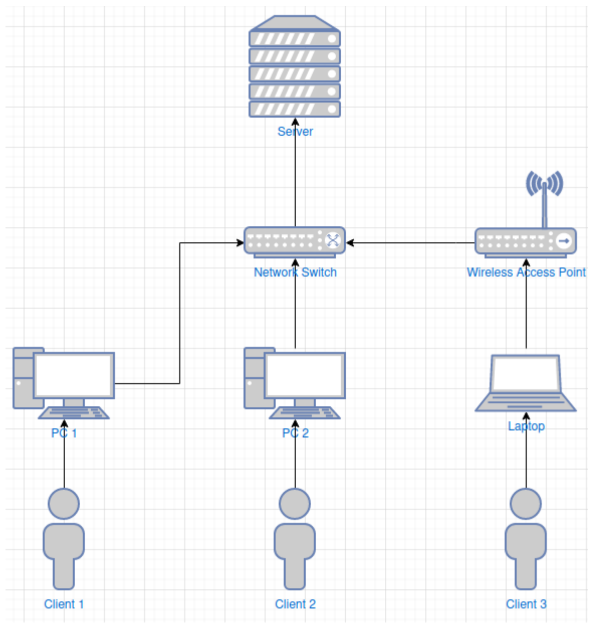


Figure 2. Chatting-Program Home Network Architecture

4.2. System Information

The information provided is the devices themselves and their IP addresses. This will be important when connecting to the server. Hardware and software information isn't necessary as the program is running a single script on the machine and all machines will have Python installed.

	Client 1	Client 2	Client 3	Server
Device	PC 1	PC 2	Laptop	Virtual Machine
IP address:	10.27.27.206	10.27.27.49	10.27.27.43	10.27.27.131

4.2.1. Local Environment

Initially when creating the program, immediate deployment of the application isn't ideal. So a local environment was used to test the programs. The program was tested on my personal computer (IP: 10.27.27.206 Port: 9898) using multiple terminal instances.

4.2.2. Server Environment

When deploying to the server(IP: 10.27.27.131), Python was installed and the program ran as expected. Something to keep note of is that the firewall blocking all ports was enabled, so allowing port: 9898 allowed the clients to initiate a connection to the server.

4.3. GUI Design

Using a Graphical User Interface for the program on the client side ensures that a user is provided with a good experience when using the chat application. (Figure 3.1) The plan for the GUI for the chat program was as follows:

1. Chatbox (list of sent messages)
2. Message box (Client sends messages)
3. Send button (sends message alternative to pressing return)
4. Disconnect button (Disconnect from the server)
5. A list of connected users to the chat (Show active users)

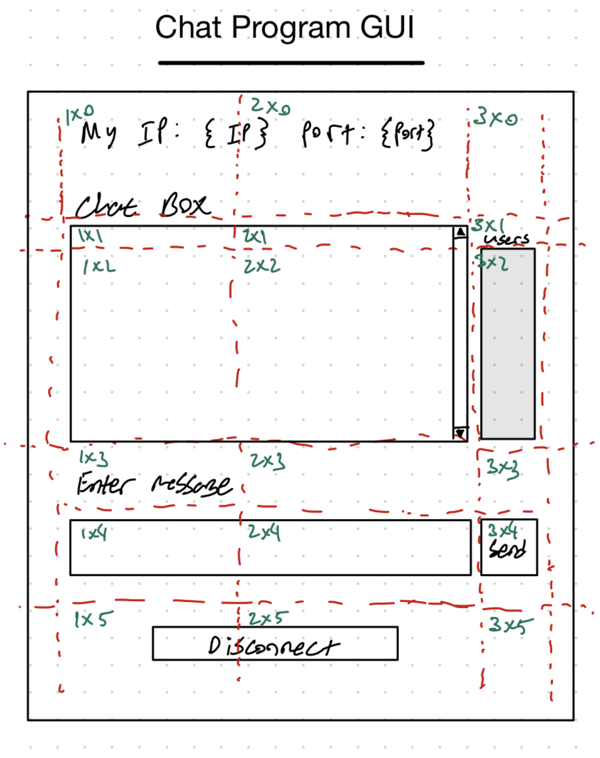


Figure 3.1 Chat Program GUI Diagram

Additionally, an initial GUI of taking in a username, IP address, and port was made so that the client can choose a name to be displayed and know what server to connect to. (Figure 3.2) The plan for the GUI for the login page is as follows:

1. Username input box (Displayed name in the program)
2. IP address input box (Default IP: 10.27.27.131)
3. Port number input box (Default port: 9898)
4. Connect Button (Connect to the server)

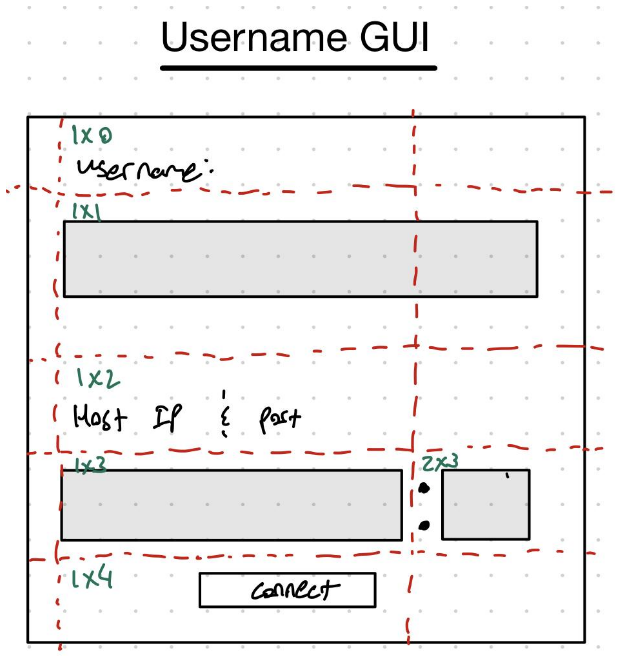


Figure 3.2 Username GUI diagram

The dotted lines represent a grid-like scheme of the diagram. This will help write the program's GUI and show where each attribute belongs.

5. Code

As mentioned before both programs will be written in Python. The client program (client.py) and the server program (server.py) will have their own separate functions performing tasks as intended. Code snippets and a general rundown of the programs will be included in the following. The entire program will be provided along with the submission of this report.

5.1. Modules

Modules are files containing existing Python codes with functions, statements, and classes. Before creating the programs, modules will be required in order to meet the objectives of the tasks. The following modules were used:

1. **Socket** (Creates sockets so that client and server machines can communicate through their endpoints)
2. **Threading** (Allows different parts of the programs to run concurrently)
3. **Tkinter** (Creates Graphical user interfaces)
    a. Python does not have Tkinter installed by default so installing Tkinter is required to run the program. The *client.exe* file provided should have all of the necessary dependencies installed and should work out of the box.
4. **Pickle** (manipulates data in a certain format so it can be sent to the client)

5.2. Client-Side Script

Following the steps shown in Figure 1, we first have to import all of the modules and define a function that will create a socket, issue a connection request to the server and finally send over the username that was inputted to the server.

```python
import socket
import pickle
import threading
from tkinter import *

def connect(name, host, port):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect((host, port))
    client.send(name.encode('utf-8'))
    return client
```

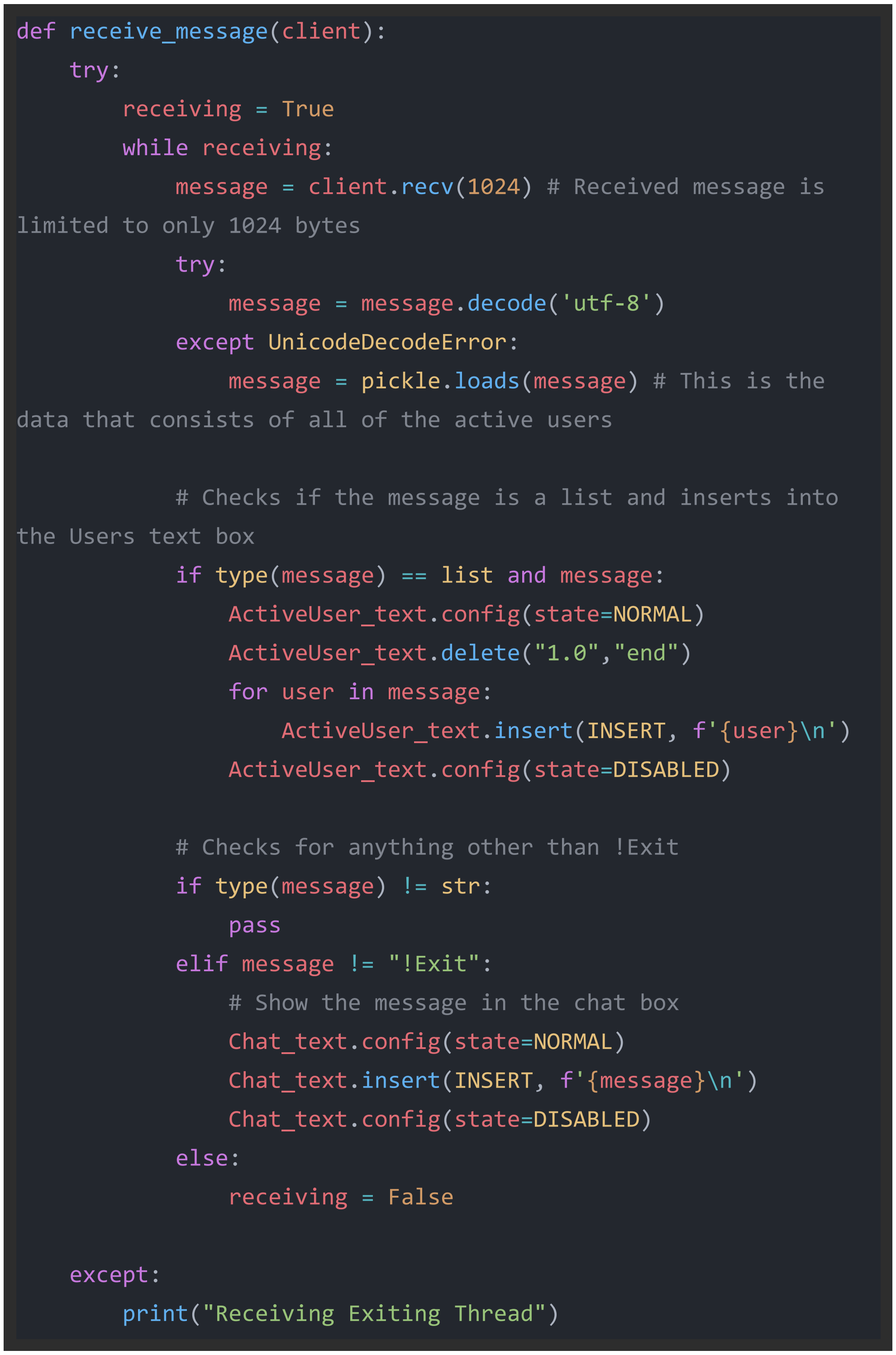
Before we can connect, a thread for receiving the message has to be created to obtain the messages that are being sent from the server.

```
try:
    HOST = Host.get()
    PORT = int(Port.get())

    client = connect(username.get(), HOST, PORT)
    thread = threading.Thread(target=receive_message,
args=(client,))
    thread.start()
except ConnectionRefusedError:
    print("ERROR: Server is not running or not accepting
connections")
    raise SystemExit
```

The user name is sent in the format of UTF-8 because the socket will only be able to transfer over bytes. The function to send over a message is a matter of encoding the message and sending it out to the server.

```
def send_message(client, msg):
    message = msg.encode('utf-8')
    client.send(message)
```

In order to receive the messages that were broadcasted back to the server, a thread is required so that we can obtain the message that isn't "!Exit" and display it. The message received is limited to only 1024 bytes which should be more than enough for the chat program. One thing to keep in mind is that if a new user is connected, the message sent will be in the pickle data or a list being sent over the TCP socket and will display those users in the GUI program. Finally, the socket will close and the program ends if the receiving is set to false.



```python
def receive_message(client):
    try:
        receiving = True
        while receiving:
            message = client.recv(1024) # Received message is
limited to only 1024 bytes
            try:
                message = message.decode('utf-8')
            except UnicodeDecodeError:
                message = pickle.loads(message) # This is the
data that consists of all of the active users

            # Checks if the message is a list and inserts into
the Users text box
            if type(message) == list and message:
                ActiveUser_text.config(state=NORMAL)
                ActiveUser_text.delete("1.0","end")
                for user in message:
                    ActiveUser_text.insert(INSERT, f'{user}\n')
                ActiveUser_text.config(state=DISABLED)

            # Checks for anything other than !Exit
            if type(message) != str:
                pass
            elif message != "!Exit":
                # Show the message in the chat box
                Chat_text.config(state=NORMAL)
                Chat_text.insert(INSERT, f'{message}\n')
                Chat_text.config(state=DISABLED)
            else:
                receiving = False

    except:
        print("Receiving Exiting Thread")
        client.close()
```

5.2.1. GUI Script

Python can provide a GUI interface using TKinter, below is the code for the username GUI where the window is the GUI window itself, Labels are texts, entries are places where the user can input text, and finally buttons.

```
UsernameInput_window = Tk()
    UsernameInput_window.eval('tk::PlaceWindow . center')
    UsernameInput_window.title("Enter Username")
    UsernameInput_window.config(padx=30, pady=10)

    Username_label = Label(text="Username: ", font=("Arial", 14,
"bold"))
    Username_label.grid(column=1, row=0, pady=(30,5), sticky="W")

    Colon_label = Label(text=":", font=("Arial", 14, "bold"))
    Colon_label.grid(column=2, row=3, sticky="W", padx=(15))

    Host_label = Label(text="Host IP & Port: ", font=("Arial", 14,
"bold"))
    Host_label.grid(column=1, row=2, pady=(30,5),columnspan=2,
sticky="W")

    username = StringVar()
    Host = StringVar()
    Port = StringVar()

    Username_entry = Entry(width = 30)
    Username_entry.grid(column=1, row=1, columnspan=2, ipady=5,
sticky="W")
    Username_entry.focus()
    Host_entry = Entry(width = 20)
    Host_entry.grid(column=1, row=3, columnspan=2, ipady=5, sticky="W")
    Host_entry.insert(0,"localhost")

    Port_entry = Entry(width = 5)
    Port_entry.grid(column=2, row=3, columnspan=1, ipady=5, sticky="E")
    Port_entry.insert(0,"9898")

    Send_button = Button(text="Connect", command=lambda:
Send_button_clicked_Username(), width=8)
    UsernameInput_window.bind('<Return>', lambda event:
Send_button_clicked_Username())
    Send_button.grid(column=1, row=4, columnspan=2, pady=25)

    UsernameInput_window.protocol("WM_DELETE_WINDOW",
Disconnect_button_clicked_Username)

    UsernameInput_window.mainloop()
```

The program already has pre-inputted data such as “localhost” and port “9898” which makes it easier for the client to connect rather than typing the same information.

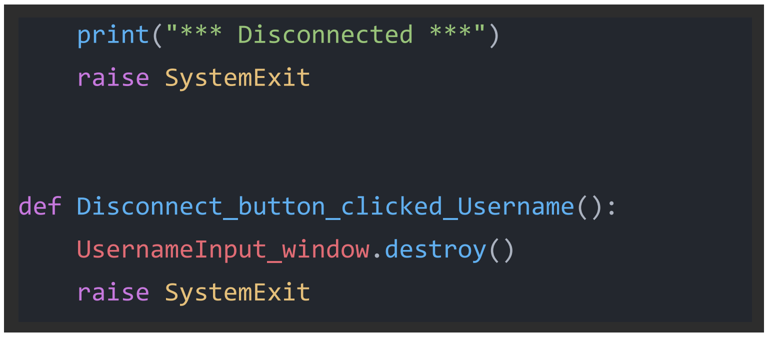
As for the Chat Program GUI itself, the code can be seen below similar to how the username GUI was described.

```
ChatProgram_window = Tk()
    ChatProgram_window.eval('tk::PlaceWindow . center')
    ChatProgram_window.title("Chatting Program")
    ChatProgram_window.config(padx=40, pady=10)

    # Label

    ChatBox_label = Label(text=f"(My IP: {client.getsockname()[0]} Port:
```

```
{client.getsockname()[1]})\n\nChat Box", font=("Arial", 18,
"bold"),justify=LEFT)
    ChatBox_label.grid(column=1, row=0, pady = 10, sticky="W")

    EnterMessage_label = Label(text="Enter Message", font=("Arial", 18,
"bold"))
    EnterMessage_label.grid(column=1, row=3, pady=(30,5), sticky="W")

    ConnectedUsers_label = Label(text="Users", font=("Arial", 12, "bold"))
    ConnectedUsers_label.grid(column=3, row=1, pady=5)

    # Text Box

    Chat_text = Text(ChatProgram_window, font=("Arial", 12, "bold"),
width=60)
    Chat_text.grid(column=1, row=1,  columnspan=2, rowspan=2)
    send_message(client, "!ShowUserHasConnected")
    Chat_text.config(state=DISABLED)

    Chat_Scroll=Scrollbar(ChatProgram_window, orient='vertical')
    Chat_Scroll.grid(column=2, row=1, sticky="nse", rowspan=2)
    Chat_Scroll.config(command=Chat_text.yview)
    Chat_text.configure(yscrollcommand=Chat_Scroll.set)

    ActiveUser_text = Text(ChatProgram_window, font=("Arial", 12, "bold"),
width=15, height=22)
    ActiveUser_text.grid(column=3, row=2, padx=15,sticky="S")

    send_message(client, "!GetAllActiveUsers") # Get a list of all active
users connected to the server
    ActiveUser_text.config(state=DISABLED)

    # Entry

    Chat_input = Entry(width = 83)
    Chat_input.grid(column=1, row=4, columnspan=2, ipady=5, sticky="W")
    Chat_input.focus()

    #Button
    Send_button = Button(text="Send", command=lambda:
Send_button_clicked_Chat(), width=15)
    ChatProgram_window.bind('<Return>', lambda event:
Send_button_clicked_Chat())
    Send_button.grid(column=3, row=4)

    Disconnect_button = Button(text="Disconnect", command=lambda:
Disconnect_button_clicked_Chat(), width = 45)
    Disconnect_button.grid(column=0, row=5, columnspan=5, pady = 50)

    ChatProgram_window.protocol("WM_DELETE_WINDOW",
Disconnect_button_clicked_Chat)

    ChatProgram_window.mainloop()
```

There are also some helper functions so that whenever a button is pressed, it will perform the following operations.

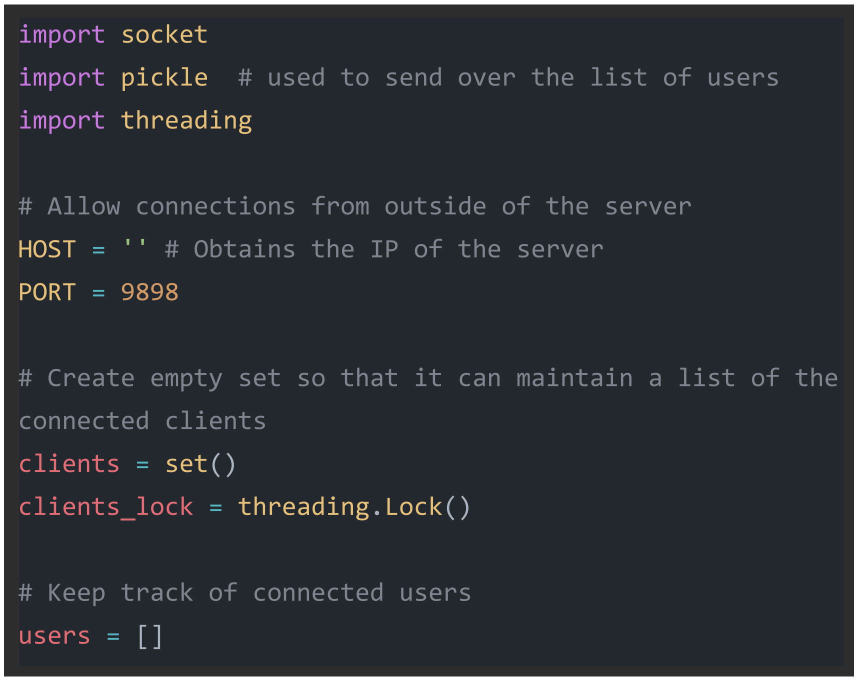
```
def Send_button_clicked_Chat():
    message = Chat_input.get()
    Chat_text.config(state=NORMAL)
    send_message(client, message)
    Chat_text.configure(state='disabled')
    Chat_input.delete(0, 'end')


def Send_button_clicked_Username():
    username.set(Username_entry.get())
    Host.set(Host_entry.get())
    Port.set(Port_entry.get())
    UsernameInput_window.destroy()


def Disconnect_button_clicked_Chat():
    ChatProgram_window.destroy()
    send_message(client, "!Exit")
```



```python
    print("*** Disconnected ***")
    raise SystemExit


def Disconnect_button_clicked_Username():
    UsernameInput_window.destroy()
    raise SystemExit
```

5.3. Server-Side Script

Before starting the server program modules have to be imported and global variables are set to maintain the data of the clients and commonly accessed variables.



```python
import socket
import pickle  # used to send over the list of users
import threading

# Allow connections from outside of the server
HOST = '' # Obtains the IP of the server
PORT = 9898

# Create empty set so that it can maintain a list of the
connected clients
clients = set()
clients_lock = threading.Lock()

# Keep track of connected users
users = []
```

Again, following the steps shown in Figure 1 and importing the modules, a socket is created, then bound to its IP and port, and will begin to allow connections. Threads are also created in order to maintain the active users that are connected to the server, giving them different attributes such as their own usernames.

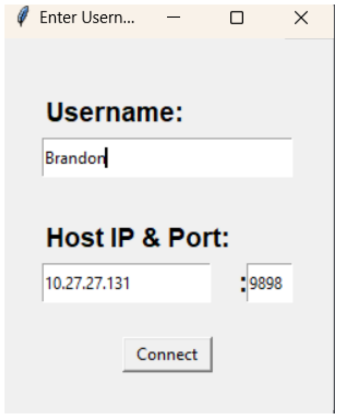
```python
if __name__ == "__main__":
    # Create Socket object for server
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind Local IP (Server IP) of the socket to the port
    server.bind((HOST, PORT))

    print("Allowing Connections...")

    # Ready to accept requests
    server.listen()

    while True:
        try:
            # When a new client connects, accept the connection and create
a new thread to handle it
            client, addr = server.accept()
            thread = threading.Thread(target=add_client, args=(client,
addr))
            thread.start()

            # Add new clients to the set
            with clients_lock:
                clients.add(client)
        except KeyboardInterrupt:
            print("Close Connections...")
            client.close()
            raise SystemExit
        except:
            print("Error accepting new client connection")
            break
```

After the client is accepted to the server, the function "add_client" is called and creates a new user with their username. The thread then sits in a constant while loop which will receive the message and if the message isn't a set command, then it will rebroadcast the message to the other connected clients. Finally, when the server is done accepting connections, it will close the socket and end the program.

```
# Handling new client connections
def add_client(client, addr):
    # Get Clients username and show that they have been connected
    username = client.recv(1024).decode("utf-8")
    users.append(username)

    print(
        f"*** {username} Connected (IP: (IP: {addr[0]} | Remote addr:
{addr[1]}) ***")

    try:
        active_connection = True

        while active_connection:
            # Recever the message that was sent from the client
            msg = client.recv(1024).decode("utf-8")

            if not msg:
                break

            # Condition to Exit the chatting program (Disconnect from
server)
            if msg == "!Exit":
                active_connection = False
                users.remove(username)
                list_active_users = pickle.dumps(users)
                print(f"*** <{username}> (IP: {addr[0]} | Remote addr:
{addr[1]}) Disconnected ***")

            # Get all of the active users that are currently connected
to the server
            elif msg == "!GetAllActiveUsers":
                # sends the user list over tcp
                list_active_users = pickle.dumps(users)

            # Display in GUI and show everyone that the user has
connected to server
            elif msg == "!ShowUserHasConnected":
                message = (f"*** {username} Connected
***".encode("utf-8"))

            else:
                message = (f"<{username}> {msg}".encode("utf-8"))

            # Loop through all connected clients and send the message to
each one
            with clients_lock:
                for c in clients:
                    try:
                        if msg == "!GetAllActiveUsers" or msg ==
"!Exit":
                            c.sendall(list_active_users)
                        else:
                            c.sendall(message)
                    except:
                        # Remove messages from the clients set, if there
is an error sending
                        clients.remove(c)
    except:
        # Show that there is an error with the client
        print(f"Error with client (IP: {addr[0]} | Remote addr:
{addr[1]})")
    finally:
        # Remove the client from the clients set and close the socket
        with clients_lock:
            clients.remove(client)
        client.close()
```

6. Result & Demonstration

Before connecting to the server, the client is prompted with a login GUI shown in Figure 4.1 where they are able to provide a username, IP, and port.

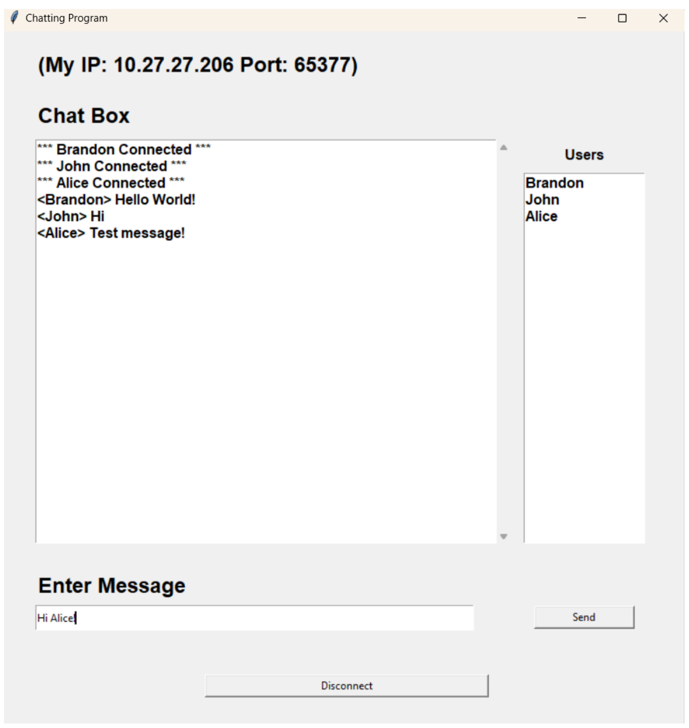


Brandon

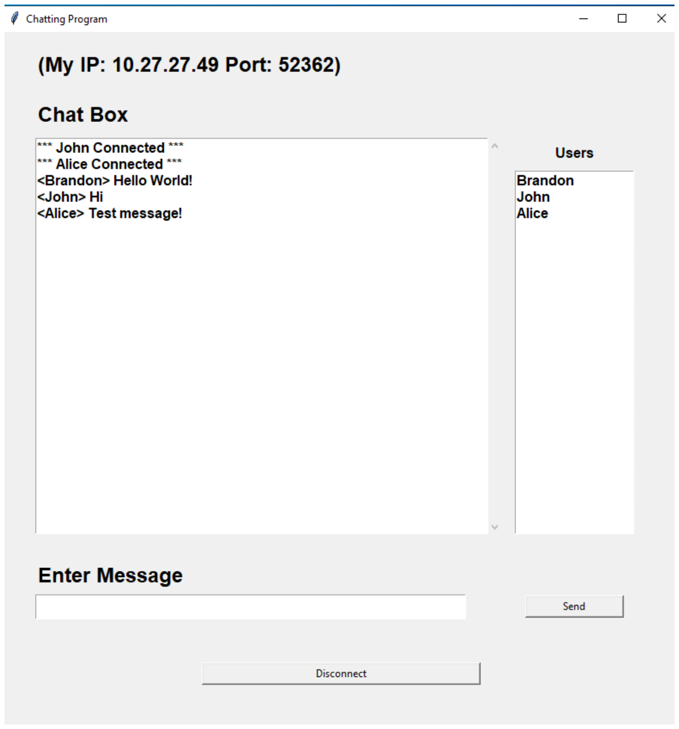
10.27.27.131 : 9898

Figure 4.1 Username, IP, and Port input GUI

The graphical user interface can be presented in the following screenshots below. Where Figure 4.1 is PC1 with an IP of 10.27.27.206, figure 2 is PC2 with an IP of 10.27.27.49, and finally, the laptop has an IP of 10.27.27.43. As mentioned in the GUI design, there is a chat box, message box, list of users, and buttons. There are also sample messages sent from each computer where the username is enclosed with greater-than and less-than symbols, followed by the message itself.



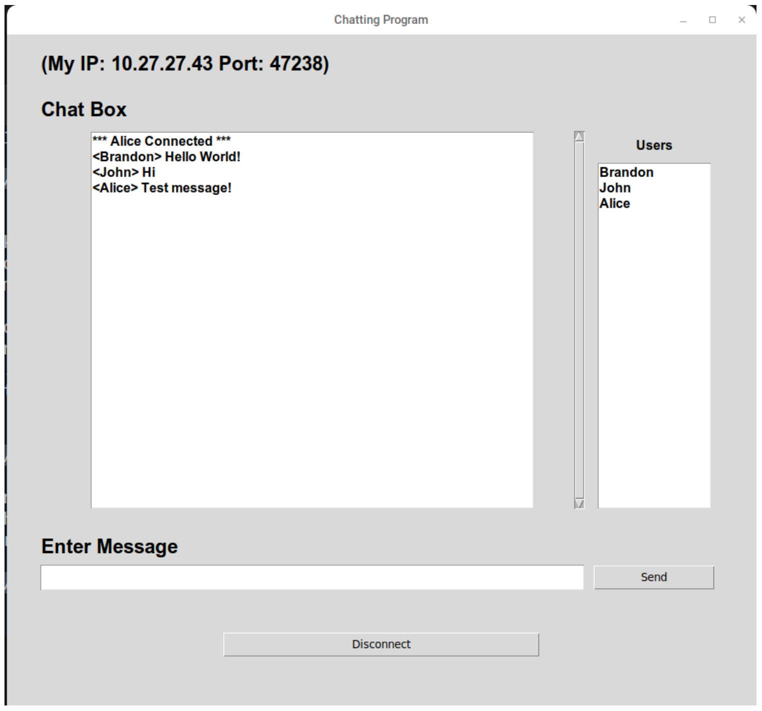
```
(My IP: 10.27.27.206 Port: 65377)

*** Brandon Connected ***
*** John Connected ***
*** Alice Connected ***
<Brandon> Hello World!
<John> Hi
<Alice> Test message!

Brandon
John
Alice

Hi Alice
```



```
(My IP: 10.27.27.49 Port: 52362)

*** John Connected ***
*** Alice Connected ***
<Brandon> Hello World!
<John> Hi
<Alice> Test message!

Brandon
John
Alice
```

Figure 4.1 PC1(Brandon) chat program

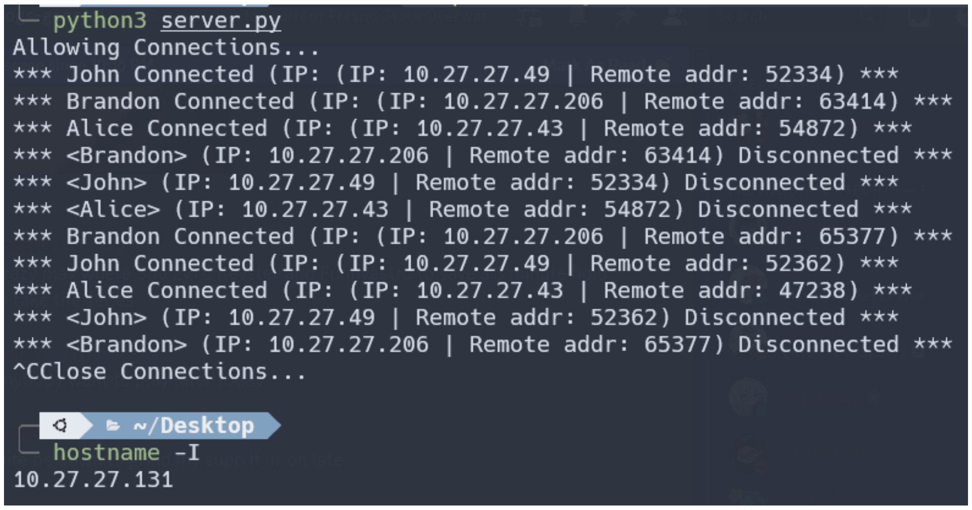
Figure 4.2 PC2 (John) Chat Program



```
(My IP: 10.27.27.43 Port: 47238)

*** Alice Connected ***
<Brandon> Hello World!
<John> Hi
<Alice> Test message!

Brandon
John
Alice
```

Figure 4.3 Laptop (Alice) Chat Program

The server does not have a GUI, instead everything is operated through the terminal. When running the program, the server shows that it is allowing connections and when a client connects to the server, it prints out the user name and their computer address information. The same goes when the user disconnects from the server which can be demonstrated in Figure 4.4.



```
└ python3 server.py
Allowing Connections...
*** John Connected (IP: (IP: 10.27.27.49 | Remote addr: 52334) ***
*** Brandon Connected (IP: (IP: 10.27.27.206 | Remote addr: 63414) ***
*** Alice Connected (IP: (IP: 10.27.27.43 | Remote addr: 54872) ***
*** <Brandon> (IP: 10.27.27.206 | Remote addr: 63414) Disconnected ***
*** <John> (IP: 10.27.27.49 | Remote addr: 52334) Disconnected ***
*** <Alice> (IP: 10.27.27.43 | Remote addr: 54872) Disconnected ***
*** Brandon Connected (IP: (IP: 10.27.27.206 | Remote addr: 65377) ***
*** John Connected (IP: (IP: 10.27.27.49 | Remote addr: 52362) ***
*** Alice Connected (IP: (IP: 10.27.27.43 | Remote addr: 47238) ***
*** <John> (IP: 10.27.27.49 | Remote addr: 52362) Disconnected ***
*** <Brandon> (IP: 10.27.27.206 | Remote addr: 65377) Disconnected ***
^CClose Connections...

┌  ~/Desktop
└ hostname -I
10.27.27.131
```

Figure 4.4 Server Terminal and IP address

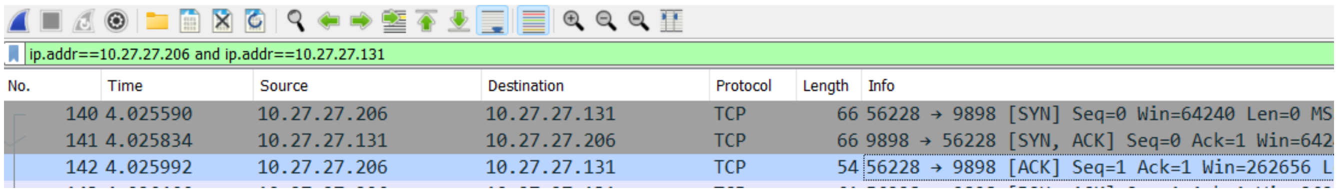
Since the server is hosted on a Linux server, the command `hostname -I` will show the server's local IP address which is “10.27.27.131”.

7. <u>**Wireshark**</u>

Wireshark is a program that allows analyzing of packets that are sent through a network. Since the chatting program uses TCP socket connections, we can understand how the messages are sent to the server and client by using Wireshark.

7.1. <u>**Packet Trace and TCP**</u>

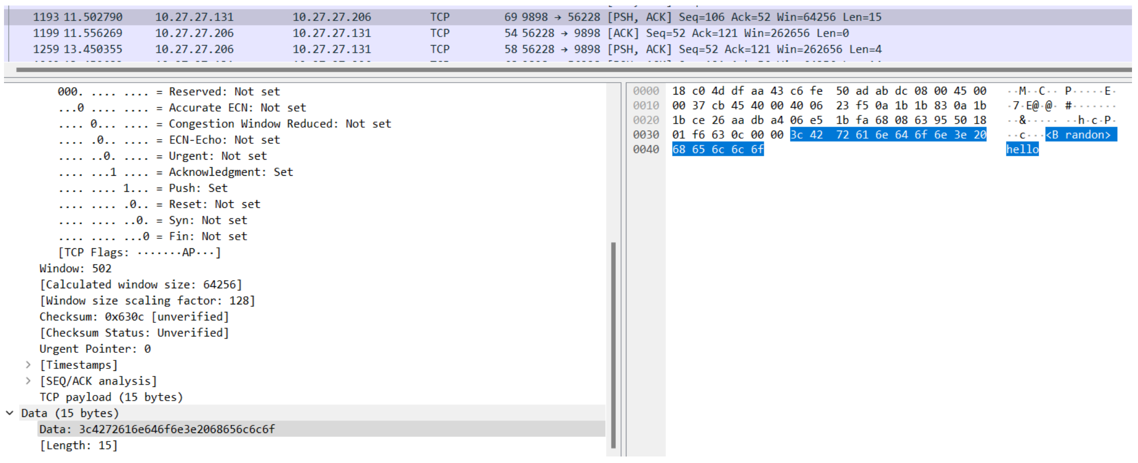
In the packet trace, a filter will be set to only show the communication between PC1 and the server. The first 3 packets are what are known as the 3-way handshake for TCP connections, ensuring that a connection between the server and client has been properly initiated.



```
ip.addr==10.27.27.206 and ip.addr==10.27.27.131

No.   Time      Source        Destination   Protocol  Length  Info
  140 4.025590  10.27.27.206  10.27.27.131  TCP         66 56228 → 9898 [SYN] Seq=0 Win=64240 Len=0 MS
  141 4.025834  10.27.27.131  10.27.27.206  TCP         66 9898 → 56228 [SYN, ACK] Seq=0 Ack=1 Win=642
  142 4.025992  10.27.27.206  10.27.27.131  TCP         54 56228 → 9898 [ACK] Seq=1 Ack=1 Win=262656 L
```

Figure 5.1 First 3 TCP packets from PC1 to the Server

As mentioned before when a message is sent, the server will rebroadcast it to the other connected clients. In packet number 1193 (Figure 5.2), the source IP is the server and the destination is PC1 where the PSH flag is set, indicating that the server is sending a message.



```
1193 11.502790  10.27.27.131  10.27.27.206  TCP  69 9898 → 56228 [PSH, ACK] Seq=106 Ack=52 Win=64256 Len=15
1199 11.556269  10.27.27.206  10.27.27.131  TCP  54 56228 → 9898 [ACK] Seq=52 Ack=121 Win=262656 Len=0
1259 13.450355  10.27.27.206  10.27.27.131  TCP  58 56228 → 9898 [PSH, ACK] Seq=52 Ack=121 Win=262656 Len=4
```

```
      000. .... .... = Reserved: Not set
      ...0 .... .... = Accurate ECN: Not set
      .... 0... .... = Congestion Window Reduced: Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 1... = Push: Set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·······AP···]
   Window: 502
   [Calculated window size: 64256]
   [Window size scaling factor: 128]
   Checksum: 0x630c [unverified]
   [Checksum Status: Unverified]
   Urgent Pointer: 0
 > [Timestamps]
 > [SEQ/ACK analysis]
   TCP payload (15 bytes)
v Data (15 bytes)
   Data: 3c4272616e646f6e3e2068656c6c6f
   [Length: 15]
```

```
0000  18 c0 4d df aa 43 c6 fe  50 ad ab dc 08 00 45 00   ··M··C·· P·····E·
0010  00 37 cb 45 40 00 40 06  23 f5 0a 1b 1b 83 0a 1b   ·7·E@·@· #·······
0020  1b ce 26 aa db a4 06 e5  1b fa 68 08 63 95 50 18   ··&····· ··h·c·P·
0030  01 f6 63 0c 00 00 3c 42  72 61 6e 64 6f 6e 3e 20   ··c···<B randon> 
0040  68 65 6c 6c 6f                                     hello
```

Figure 5.2 Packet 1193 PSH flag set and data sent.

Something to keep in mind is that when sending the messages to the server, and broadcasting it back, the data can be seen in plaintext. This poses a major security issue as the messages are not encrypted and anyone who is intercepting these packets can see the messages being transmitted.

8. Conclusion

Although this is a simple chatting program between server and clients, it has still shown how communicating through sockets endpoints can be completed with a TCP connection. The simple chat room server allows multiple clients and while the program has its limitations, it still shows a clear demonstration of creating one-to-one or one-to-many conversations.

Some future work may include implementing security measures within the chat program. The messages can be seen in plaintext if captured by a packet sniffer but by adding some form of encryption, messages will be difficult to read and it ensures the privacy of the connected clients. Also adding some more features to the program such as commands, authentication, message retention and more could be possibly implemented within the near future.

Overall, by building a real-time chat application, you can provide users with a medium to communicate with each other. In addition, you have full control over what features you want to add to the program and how the client's graphical user interface would look. It is also important to understand how communication is transmitted between the devices that are connected within a network and how TCP sockets ensure that connection. The main goal of a chat application is to collaborate with other users making it engaging and environment-friendly for everyone connected.

9. References

Work Cited and Resources

1. Amos, David. (2019). Python GUI Programming With Tkinter. *realpython.com*. https://realpython.com/python-gui-tkinter/
2. Anderson, Jim (2019). An Intro to Threading in Python. *realpython.com*. https://realpython.com/intro-to-python-threading/
3. Davide, Mastromatteo. (2020). The Python pickle Module: How to Persist Objects in Python. *realpython.com*. https://realpython.com/python-pickle-module/
4. McMillan Gordon (2023). Socket Programming HOWTO. Python Documentation. https://docs.python.org/3/howto/sockets.html
5. sentdex. (2014, August 16). *Python 3 Programming Tutorial - Sockets intro* [Video]. YouTube. https://www.youtube.com/watch?v=wzrGwor2veQ