**3D3 – Computer Networks**

**Project 1**

Submitted by: Preetam Singhvi (18335173)

**Real Time GUI Based Chat Application**

Abstract:

Chat refers to the process of communicating, interacting and/or exchanging messages over the Internet. It involves two or more individuals that communicate through a chat-enabled service or software. Chat may be delivered through text, audio or video communication via the Internet.

A chat application has basic two components, viz server and client. A server is a computer program or a device that provides functionality for other programs or devices. Clients who want to chat with each other connect to the server.

The chat application we are going to make will be more like a chat room, rather than a peer to peer chat. So this means that multiple users can connect to the chat server and send their messages. Every message is broadcasted to every connected chat user.

Preferred Used Case:

The client-server communication model is used in a wide variety of software applications. Where normally the server side is sufficiently protected and sealed from public access, but client applications running on devices like notebooks and desktops are considered insecure and exposed to security threats.

The main weakness of client-server chat application is that there is no security provided to data which is transferred between clients. Any unauthorized client can hack the client account and can change the data. This is the main objective of this project (To develop a secured Client-Server Chat Application).

Protocol Overview:

We will be using TCP sockets for this purpose, and therefore we use AF\_INET and SOCK\_STREAM flags. We use them over UDP sockets because they’re more telephonic, where the recipient has to approve the incoming connection before communication begins, and UDP sockets are more post-mail sort of thing (anyone can send a mail to any recipient whose address s/he knows), so they don’t really require an establishment of connection before communication can happen. Clearly, TCP suit more to our purpose than UDP sockets, therefore we use them.

We’ll be using an infinite loop since, we will be receiving messages quite non-deterministically, and independent of how and when we send the messages. We don’t want this to be a walkie-talkie chat app which can only either send or receive at a time; we want to receive messages when we can, and send them when we want. The functionality within the loop is pretty straightforward; the recv() is the blocking part. It stops execution until it receives a message, and when it does, we move ahead and append the message to msg\_list. We will soon define msg\_list, which is basically a Tkinter feature for displaying the list of messages on the screen.

Module Descriptions:

To accommodate the requirements presented in the above section, the chat room is modelled around the two-tier architecture: client tier & server tier.

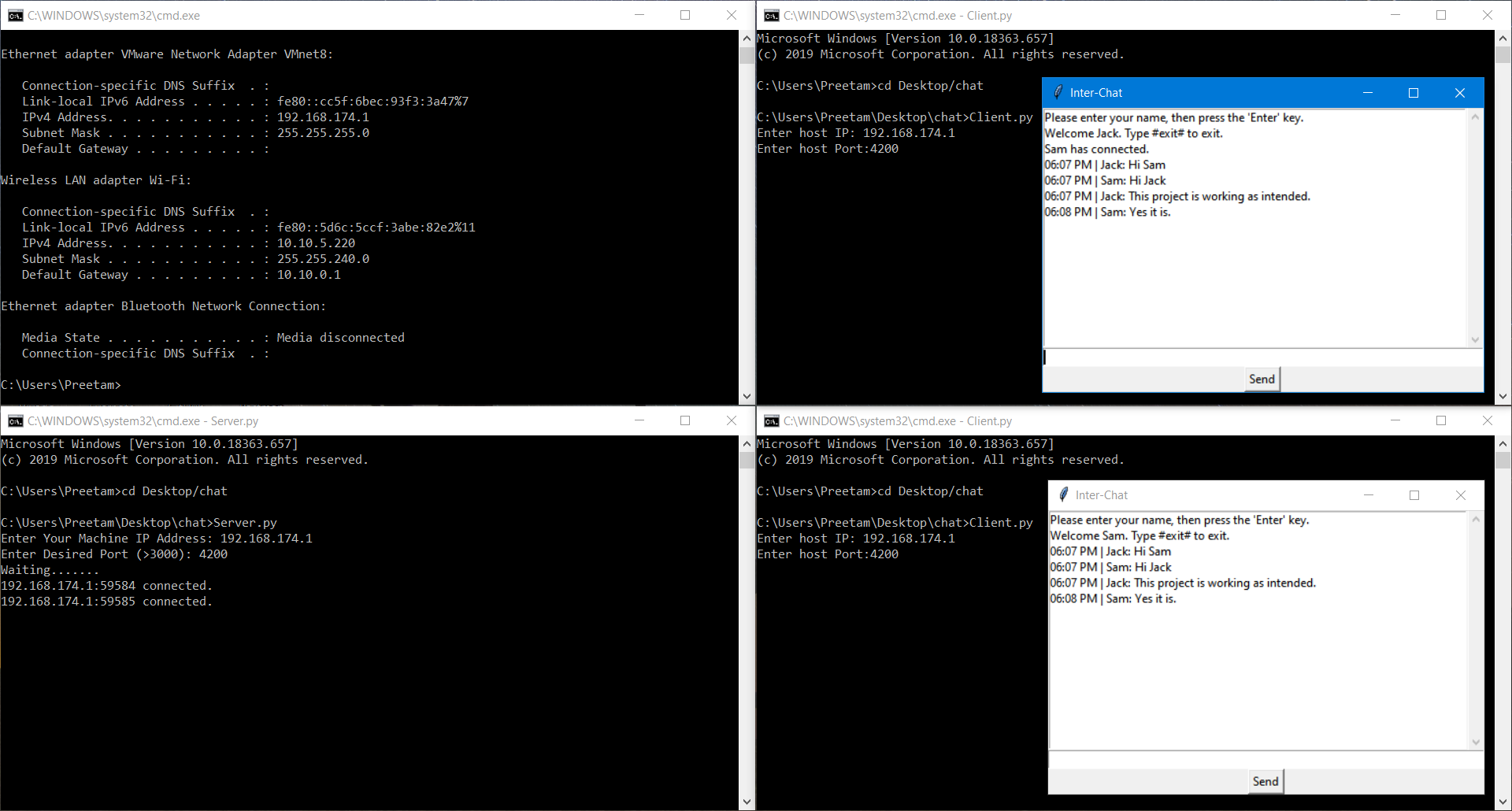
Client tier is made up of the following components:

* Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.
* Client Socket module is used to manage any client willing to connect on a specific host and port. It authorizes the client to send and receive messages.
  + def receive():
    - Why an infinite loop again?
    - Because we’ll be receiving messages quite non-deterministically, and independent of how and when we send the messages. We don’t want this to be a walkie-talkie chat app which can only either send or receive at a time; we want to receive messages when we can and send them when we want. The functionality within the loop is pretty straightforward; the recv() is the blocking part. It stops execution until it receives a message, and when it does, we move ahead and append the message to msg\_list. We will soon define msg\_list, which is basically a Tkinter feature for displaying the list of messages on the screen.
  + def send(event=None):
    - We’re using event as an argument because it is implicitly passed by Tkinter when the send button on the GUI is pressed. my\_msg is the input field on the GUI, and therefore we extract the message to be sent usin g msg = my\_msg.get(). After that, we clear the input field and then send the message to the server, which, as we’ve seen before, broadcasts this message to all the clients (if it’s not an exit message). If it is an exit message, we close the socket and then the GUI app (via top.close())
* Client Socket Methods
  + s.connect() : This method actively initiates TCP server connection.

Application server tier is made up of:

* We will be using TCP sockets for this purpose, and therefore we use AF\_INET and SOCK\_STREAM flags. We use them over UDP sockets because they’re more telephonic, where the recipient has to approve the incoming connection before communication begins, and UDP sockets are more post-mail sort of thing (anyone can send a mail to any recipient whose address s/he knows), so they don’t really require an establishment of connection before communication can happen. Clearly, TCP suit more to our purpose than UDP sockets, therefore we use them
  + def accept\_incoming\_connections():
    - This is just a loop that waits forever for incoming connections and as soon as it gets one, it logs the connection (prints some of the connection details) and sends the connected client a welcome message. Then it stores the client’s address in the addresses dictionary and later starts the handling thread for that client. Of course, we haven’t yet defined the target function handle\_client().
  + def handle\_client(client):
    - Naturally, after we send the new client the welcoming message, it will reply with the name s/he wants to use for further communication. In the handle\_client() function, the first task we do is we save this name, and then send another message to the client, regarding further instructions. After this comes the main loop for communication: here we receive further messages from the client and if a message doesn’t contain instructions to quit, we simply broadcast the message to other connected clients (we’ll be defining the broadcast method in a moment). If we do encounter a message with exit instructions (i.e., the client sends a {quit}), we echo back the same message to the client (it triggers close action on the client side) and then we close the connection socket for it. We then do some clean up by deleting the entry for the client, and finally give a shout out to other connected people that this person has left the conversation.
  + def broadcast(msg, prefix="")
    - This is pretty much self-explanatory; it simply sends the msg to all the connected clients and prepends an optional prefix if necessary. We do pass a prefix to broadcast() in our handle\_client() function, and we do it so that people can see exactly who the sender of a particular message is. That was all the required functionalities for our server.
* Server Socket Methods
  + s.bind() : This method binds address (hostname, port number pair) to socket.
  + s.listen() : This method sets up and start TCP listener.
  + s.accept() : This passively accept TCP client connection, waiting until connection arrives (blocking)

Working Model



Summary of Algorithms:

In this project we have used the basics of networking in python. We also learned how to make a GUI for our application. This project teaches us how to create a basic chat application by creating a local network and using TCP sockets.