CLIENT:

import tkinter as tk

from tkinter import ttk

import threading

import socket

This code imports the necessary libraries and modules required for building a GUI application, creating a thread, and establishing a socket connection.

**tkinter** is a Python standard library that provides GUI functionalities to Python applications. **ttk** is a themed widget library extension for tkinter, which provides more customizable and modern-looking widgets.

**threading** is a Python module that provides classes to create and manage threads.

**socket** is a Python module that provides a low-level interface for network communication. It allows two different devices to communicate over a network.

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

This line creates a new socket object using the **socket()** function from the **socket** module.

The **AF\_INET** argument specifies the address family of the socket. In this case, it specifies that the socket will use IPv4 addressing.

The **SOCK\_STREAM** argument specifies the type of socket. In this case, it specifies that the socket will be a TCP socket, which provides a reliable, stream-oriented connection between two devices.

The returned **client\_socket** object is an instance of the **socket** class, which is used to establish network connections with other devices, send and receive data over the network, and manage socket-related errors and exceptions.

*# Define host and port for server*

HOST = 'localhost'

PORT = 5000

*# Connect to the server*

client\_socket.connect((HOST, PORT))

These lines define the host and port number for the server that the client will connect to, and then establishes a connection to the server using the **connect()** method of the **client\_socket** object.

In this code, the server is located on the same machine as the client, and is identified by the hostname **'localhost'**. This means that the client and server are both running on the same computer, and the server can be accessed using the loopback IP address **127.0.0.1**.

The **PORT** variable specifies the port number on which the server is listening for incoming connections. In this case, the server is listening on port number **5000**.

The **connect()** method is used to initiate a connection to the server. It takes a tuple containing the server's hostname and port number as its argument. When the connection is successful, the client can send and receive data to and from the server over the established network connection.

*# Define function to send message to server*

def send\_message():

    message = message\_entry.get()

    sender = sender\_entry.get()

    data = {'message': message, 'sender': sender}

    client\_socket.sendall(str(data).encode())

    message\_entry.delete(0, tk.END)

    if not sender\_entry['state'] == 'disabled':

        sender\_entry.config(state='disabled')

This code defines a function called **send\_message()** that is used to send a message from the client to the server over the network connection.

Here is a step-by-step explanation of what the code does:

1. The **message** and **sender** variables are assigned the values entered by the user in the **message\_entry** and **sender\_entry** widgets, respectively.
2. A dictionary called **data** is created with two key-value pairs: **'message'** and **'sender'**. The values of these keys are the **message** and **sender** variables, respectively.
3. The **sendall()** method of the **client\_socket** object is used to send the **data** dictionary to the server. The **str()** function is used to convert the dictionary to a string, and the **encode()** method is used to convert the string to bytes so that it can be sent over the network.
4. The **message\_entry** widget is cleared by deleting the contents of the entry field.
5. The **if** statement checks if the **sender\_entry** widget is enabled (i.e., its **state** attribute is not **'disabled'**). If it is, the **state** attribute of the **sender\_entry** widget is set to **'disabled'** so that the user cannot change their username after sending a message.

*# Define function to update chat box with received messages*

def update\_chat\_box(message):

    chat\_box.config(state='normal')

    chat\_box.insert(tk.END, message + '\n')

    chat\_box.config(state='disabled')

Top of Form

This is a function named **update\_chat\_box** that updates the chat box with received messages. Here's what each line of the function does:

1. **chat\_box.config(state='normal')**: This line sets the state of the **chat\_box** to normal, which allows for the text in the widget to be edited.
2. **chat\_box.insert(tk.END, message + '\n')**: This line inserts the message received from the server at the end of the **chat\_box** widget. The **tk.END** argument specifies that the message should be inserted at the end of the widget. The **'\n'** character is added to ensure that each message is displayed on a new line.
3. **chat\_box.config(state='disabled')**: This line sets the state of the **chat\_box** widget to disabled, which ensures that the text in the widget cannot be edited by the user.

Overall, this function updates the chat box with received messages by first enabling editing, then inserting the message, and finally disabling editing again to prevent the user from accidentally modifying the displayed messages

*# Define function to receive messages from server*

def receive\_messages():

    while True:

        data = client\_socket.recv(1024)

        message = data.decode()

        update\_chat\_box(message)

his code defines a function **receive\_messages()** that runs an infinite loop and continuously receives messages from the server. Here's what each line does:

while True:

This creates an infinite loop which runs until the program is terminated.

data = client\_socket.recv(1024)

This line receives the data from the server with a maximum size of 1024 bytes and stores it in the **data** variable.

kotlinCopy code

message = data.decode()

This line decodes the received data from bytes to a string using the **decode()** method and stores it in the **message** variable.

update\_chat\_box(message)

This line calls the **update\_chat\_box()** function with the received message as an argument to update the chat box with the message.

Overall, this function listens for messages from the server, receives them, and updates the chat box with the received message. It runs continuously until the program is terminated.

Top of Form

*# Create a new thread to receive messages from server*

receive\_thread = threading.Thread(target=receive\_messages)

receive\_thread.start()

These lines of code create a new thread called **receive\_thread** that runs the **receive\_messages()** function. The **target** parameter specifies the function to run in the new thread, and the **start()** method starts the new thread.

The purpose of creating a new thread is to allow the program to receive messages from the server while still being able to send messages and interact with the GUI. If the **receive\_messages()** function were called in the main thread, it would block the program and prevent it from responding to user input or updating the GUI until a message was received. By running it in a separate thread, the program can continue running normally while also receiving messages from the server in the background.

**SERVER:**

import socket

from threading import Thread

from flask import Flask, render\_template

These lines of code are used to import necessary modules for creating a web application with Flask and implementing a chat functionality using sockets and threading:

* socket module provides the necessary functions for creating and using sockets, which are used to establish a network connection for sending and receiving data.
* Thread class is used to create and manage threads, which are used to execute functions concurrently.
* Flask is a micro web framework written in Python that is used for building web applications. It provides a variety of tools and libraries for handling web-related tasks such as routing, rendering templates, and handling requests and responses.
* render\_template is a function from Flask that is used to render HTML templates for web pages.

Overall, these lines of code are necessary for building a web-based chat application using Flask and sockets with threaded communication.

*# Create a Flask app instance*

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'secret!'

*# Define host and port for server*

HOST = 'localhost'

PORT = 5000

The code creates an instance of a Flask application by calling the Flask constructor with the argument **\_\_name\_\_**, which is a special Python variable that is set to the name of the current module. The instance is stored in the variable **app**.

The line **app.config['SECRET\_KEY'] = 'secret!'** sets the **SECRET\_KEY** configuration variable of the Flask app to the string "secret!". This key is used by Flask to secure sessions and can be any random string of characters.

The variables **HOST** and **PORT** define the host address and port number for the server. In this case, the server will be bound to the local host (**localhost**) and the port number **5000**.

*# Create a socket object*

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

*# Set socket option to allow re-use of the address and port*

server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

*# Bind the socket to the host and port*

server\_socket.bind((HOST, PORT))

*# Listen for incoming client connections*

server\_socket.listen()

These lines of code are used to create and set up a socket object for the server.

**socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)** creates a socket object with the given address family (**AF\_INET** for IPv4) and socket type (**SOCK\_STREAM** for TCP/IP).

**server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)** sets the socket option **SO\_REUSEADDR** to 1, which allows the address and port to be re-used immediately after the server is closed.

**server\_socket.bind((HOST, PORT))** binds the socket to the host and port specified in **HOST** and **PORT**.

**server\_socket.listen()** makes the server start listening for incoming client connections.

*# Define function to broadcast message to all connected clients*

def broadcast\_message(sender, message):

    for client in clients:

        if client != sender:

            try:

*# Send the message to the client*

                client.sendall(f' {message}'.encode())

            except socket.error:

*# If there's an error sending the message, remove the client from the list of connected clients*

                clients.remove(client)

                print(f'Client {client} disconnected')

The **broadcast\_message** function is defined to send a message to all connected clients except for the sender of the message.

The function takes two arguments:

* **sender**: The client that sent the message.
* **message**: The message that was sent by the sender.

The function uses a **for** loop to iterate through all connected clients stored in the **clients** list. If the client being iterated over is not the same as the sender of the message, the function attempts to send the message to that client using the **sendall()** method of the client's socket object. The message is encoded using the **encode()** method before being sent.

If there is an error sending the message to a particular client (e.g., the client has disconnected), the function removes that client from the **clients** list using the **remove()** method and prints a message to the console indicating that the client has disconnected.

Note that the **clients** list is assumed to be defined elsewhere in the code and contains socket objects for all currently connected clients.

*# Define function to handle client connection*

def handle\_client(client\_socket, addr):

    print(f'Client connected from {addr}')

*# Add the client socket to the list of connected clients*

    clients.append(client\_socket)

    while True:

*# Receive data from the client*

        data = client\_socket.recv(1024)

*# If there's no data, the client has disconnected*

        if not data:

*# Remove the client from the list of connected clients and close the connection*

            clients.remove(client\_socket)

            client\_socket.close()

            break

*# Decode the received data into a string*

        message = data.decode()

*# Broadcast the message to all connected clients*

        broadcast\_message(addr, message)

Top of Form

This code defines a function **handle\_client** that will be used to handle client connections to the server. When a client connects to the server, this function is called in a new thread to handle the client's requests.

The function takes two parameters: **client\_socket**, which is the socket object representing the client's connection, and **addr**, which is the address of the client.

The first thing the function does is print a message to the console indicating that a client has connected from the given address.

Next, the client socket is added to the list of connected clients. The list **clients** is a global variable that is initialized as an empty list at the beginning of the script.

The function then enters into an infinite loop that listens for incoming data from the client. It uses the **recv()** method of the client socket to receive data from the client, with a buffer size of 1024 bytes.

If there is no data received, it means that the client has disconnected, so the function removes the client socket from the list of connected clients, closes the client socket, and breaks out of the loop.

If there is data received, it is decoded from bytes to a string using the **decode()** method of the socket object. The decoded message is then broadcast to all connected clients using the **broadcast\_message()** function, passing the **addr** and **message** as parameters.

The **broadcast\_message()** function sends the message to all clients in the **clients** list except the sender, using a loop to iterate over the list and the **sendall()** method of each socket object to send the message. If there is an error sending the message, the client is removed from the list of connected clients and the connection is closed.

Overall, the **handle\_client()** function handles client connections by adding the client to the list of connected clients, listening for incoming data, and broadcasting any received messages to all other clients.

*# Define a route for the index page*

@app.route('/')

def index():

    return render\_template('index.html')

if \_\_name\_\_ == '\_\_main\_\_':

    print(f'Server running on {HOST}:{PORT}')

*# Listen for incoming client connections*

    while True:

*# Accept an incoming client connection and create a new thread to handle the connection*

        client\_socket, addr = server\_socket.accept()

        client\_thread = Thread(target=handle\_client, args=(client\_socket, addr))

        client\_thread.start()

Top of Form

This section of code sets up a Flask web application and defines a route for the index page. It then starts a loop that listens for incoming client connections and creates a new thread to handle each connection.

**@app.route('/')** is a decorator that creates a route for the root URL of the web application. When a user visits the root URL, the **index()** function is called, which renders the **index.html** template using the **render\_template()** function from Flask.

The **if \_\_name\_\_ == '\_\_main\_\_'** statement is a Python convention that allows the script to be used both as a standalone program and as a module in another program. When the script is run as a standalone program, the code block following this statement is executed. This block starts the server by calling the **run()** method on the Flask app instance, which starts the development server.

The **while True** loop listens for incoming client connections using the **accept()** method on the server socket object. When a connection is accepted, the method returns a new client socket object and the address of the client. A new thread is then created to handle the connection by calling the **handle\_client()** function with the client socket and address as arguments. The loop then goes back to listening for incoming connections.

The **handle\_client()** function is responsible for receiving messages from the client and broadcasting them to all connected clients. It first adds the client socket object to the list of connected clients. It then enters a loop that receives data from the client using the **recv()** method on the client socket object. If the received data is empty, it means the client has disconnected, so the function removes the client from the list of connected clients and closes the connection. Otherwise, the received data is broadcast to all connected clients using the **broadcast\_message()** function.

Overall, this section of code sets up a server that listens for incoming client connections, receives and broadcasts messages, and renders an index page for the web application.

Top of Form