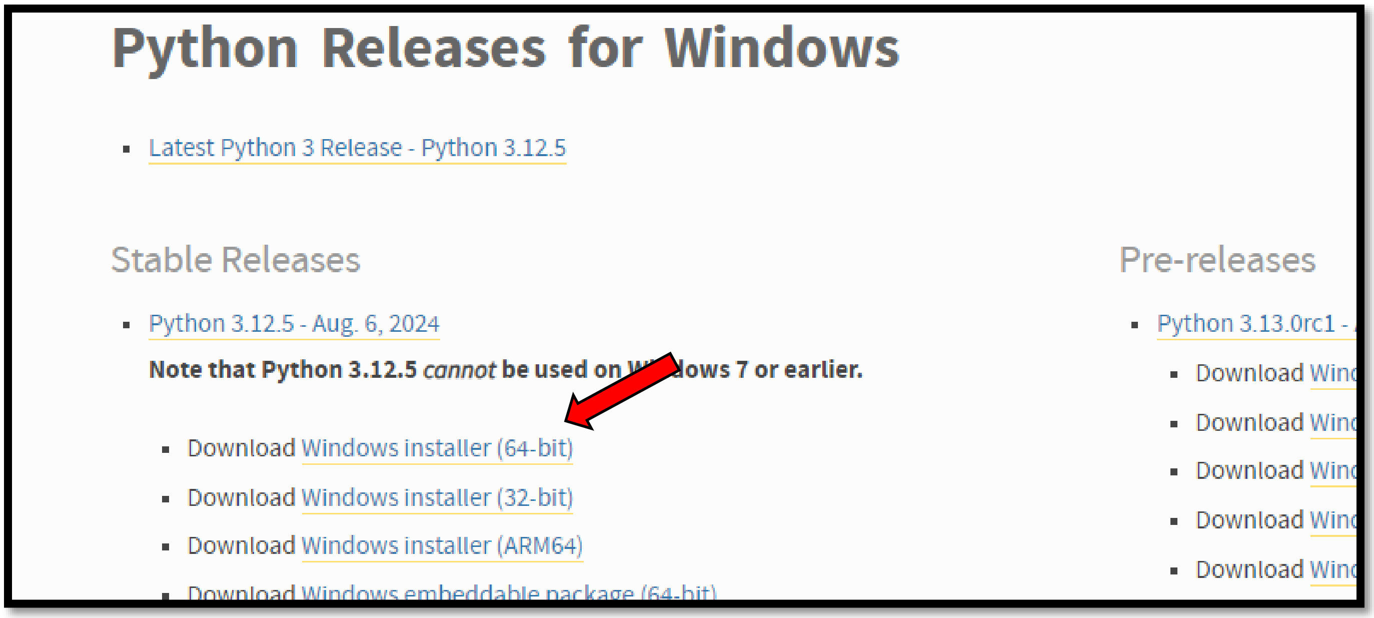
Lab 14-Socket Programming

1.1 Installation of and configuration of Python and Visual Studio Code

Visit this **Link** to download installer of Python. And visit this **Link** to download the Visual Studio Code.



Python Releases for Windows

- Latest Python 3 Release - Python 3.12.5

Stable Releases

- Python 3.12.5 - Aug. 6, 2024

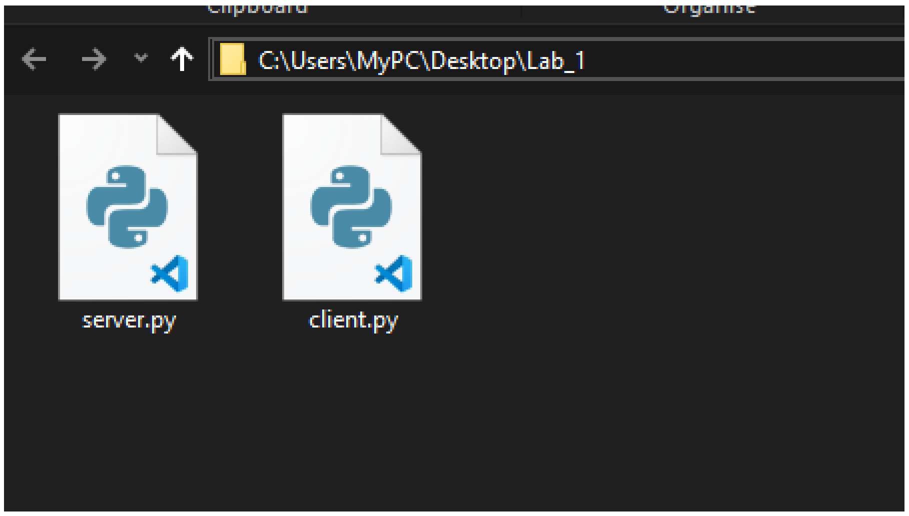
  Note that Python 3.12.5 *cannot* be used on Windows 7 or earlier.

  - Download Windows installer (64-bit)
  - Download Windows installer (32-bit)
  - Download Windows installer (ARM64)
  - Download Windows embeddable package (64-bit)

Pre-releases

- Python 3.13.0rc1 -
  - Download Wind
  - Download Wind
  - Download Wind
  - Download Wind
  - Download Wind

After downloading and installing Python and Visual Studio Code. Now create a folder on Desktop or any other location and create two blank Python files named *client.py* as *server.py* in that folder. Now open the Visual Studio Code and open that folder in the Visual Studio Code.



1.2 Types of network communication protocols

- **Wi-Fi:** An example of a link layer protocol, meaning it sits very close to the hardware and is responsible for physically sending data from one device to another in a wireless environment.
- **IP (Internet Protocol):** IP is a network layer protocol mainly responsible for routing packets and IP addressing.
- **TCP (Transmission Control Protocol):** A reliable, connection-oriented protocol that provides full duplex communication and ensures data integrity and delivery. This is a transport layer protocol, which manages connections, detects errors, and controls information flow.
- **UDP (User Datagram Protocol):** A protocol from the same protocol suite as TCP. The main difference is that UDP is a more simple, fast, but unreliable connectionless protocol that does not perform any delivery checks and follows the paradigm of "fire-and-forget." As TCP, UPD is also located on the transport layer.
- **HTTP (Hypertext Transfer Protocol):** An application layer protocol and the most commonly used protocol for browser-to-server communication on the web, used to serve websites in particular. It goes without saying that this article that you are reading right now was also served via HTTP. HTTP protocol builds on top of TCP and manages and transfers information relevant to web applications like headers, which are used to transfer metadata and cookies, different HTTP methods (GET, POST, DELETE, UPDATE) etc.
- **MQTT (Message Queue Telemetry Transport):** An application-level protocol used for devices with limited processing power and battery life, operating in unreliable network conditions (for example, gas sensors on a mining site or simply a smart light bulb in your house). MQTT is a standard messaging protocol used in IoT (Internet of Things).

Note: An important observation is that all the abovementioned protocols use sockets under the hood but add their own logic and data processing on top. This is due to sockets being a low-level interface for any network communications in modern devices as we will discuss in the next section.

1.3 Socket Background

A socket is an interface (gate) for communication between different processes located on the same or different machines.

The socket is primarily a concept used in the transport layer of the Internet protocol suite (TCP/IP) or session layer of the OSI model. Sockets have a long history. Their use originated with ARPANET (Advanced Research Projects Agency Network) in 1971 and later became an API in the Berkeley Software Distribution (BSD), a Unix based operating system released in 1983 called *Berkeley sockets*.

The most common type of socket applications are client-server applications, where one side acts as the server and waits for connections from clients. This is the type of application that you'll be creating in this lab. More specifically, you'll focus on the socket API for Internet sockets, sometimes called Berkeley or BSD sockets. There are also Unix domain sockets, which can only be used to communicate between processes on the same host.

1.4 Socket API Overview

Python's socket module provides an interface to the Berkeley sockets API. This is the module that you'll use in this lab.

The primary socket API functions and methods in this module are:

```
.socket()
.bind()
.listen()
.accept()
.connect()
.connect_ex()
.send()
.recv()
.close()
```

1.5 TCP Sockets

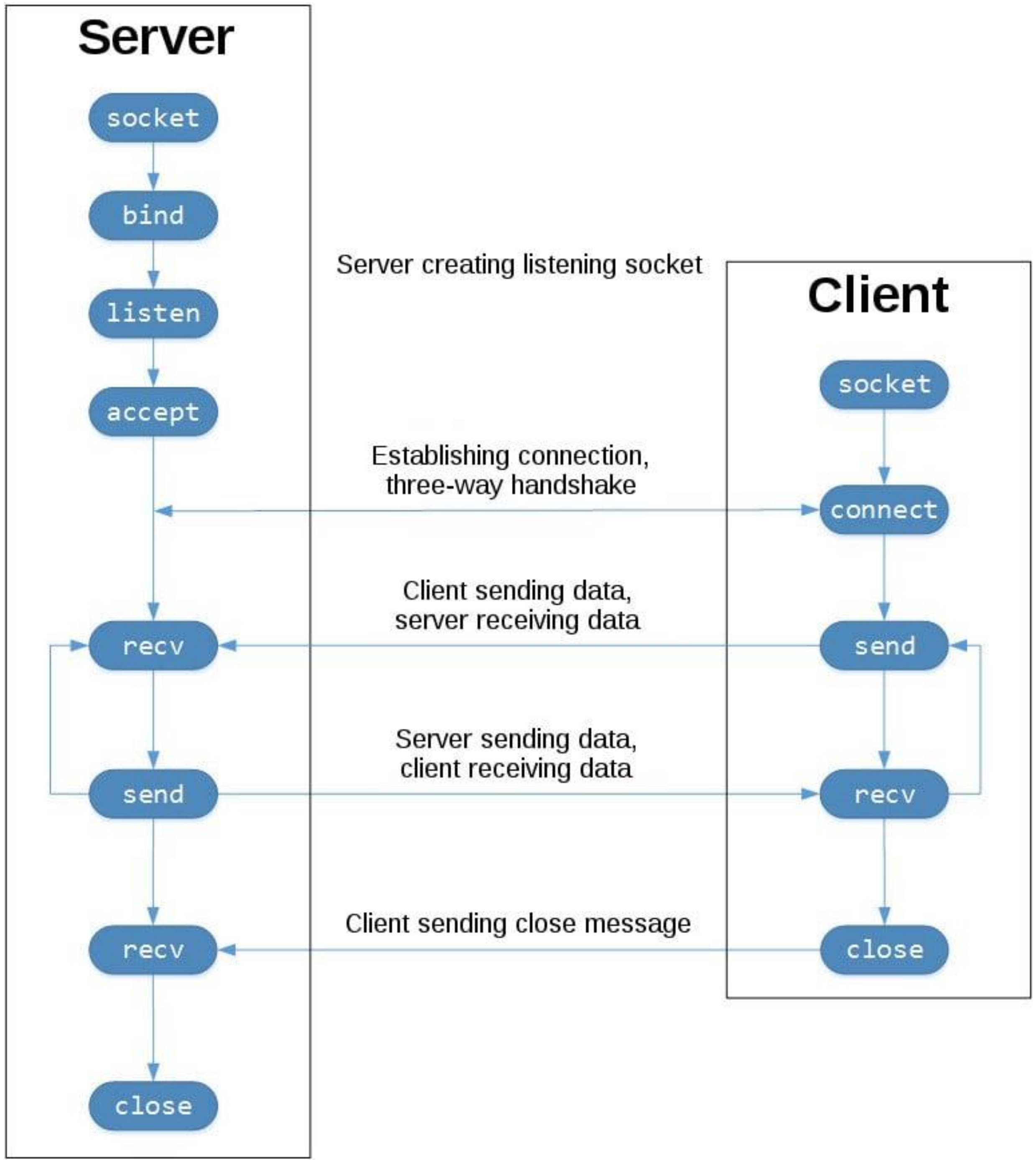
We are going to create a socket object using socket.socket(), specifying the socket type as socket.SOCK_STREAM. When you do that, the default protocol that's used is the Transmission Control Protocol (TCP).

Why should you use the *Transmission Control Protocol* (TCP):

- **Is reliable:** Packets dropped in the network are detected and retransmitted by the sender.
- **Has in-order data delivery:** Data is read by your application in the order it was written by the sender.

In contrast, *User Datagram Protocol* (UDP) sockets created with socket.SOCK_DGRAM aren't reliable, and data read by the receiver can be out-of-order from the sender's writes.

To better understand this, check out the sequence of socket API calls and data flow for TCP:


Server
socket
bind
listen
accept
recv
send
recv
close
Server creating listening socket
Establishing connection, three-way handshake
Client sending data, server receiving data
Server sending data, client receiving data
Client sending close message
Client
socket
connect
send
recv
close

Starting in the top left-hand column, note the API calls that the server makes to set up a "listening" socket:

- socket()
- .bind()
- .listen()
- .accept()

A listening socket does just what its name suggests. It listens for connections from clients. When a client connects, the server calls .accept() to accept, or complete, the connection. The client calls .connect() to establish a connection to the server and initiate the three-way handshake.

1.6 Server Code

Write the following code in the *server.py* file you created in the beginning.

```
import socket

HOST = "127.0.0.1"          # Standard loopback interface address (localhost)
PORT = 65432                # Port to listen on (non-privileged ports are > 1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

- socket.socket() creates a socket object that supports the context manager type, so you can use it in a with statement. There's no need to call s.close().
- The arguments passed to socket() are constants used to specify the address family and socket type. AF_INET is the Internet address family for IPv4. SOCK_STREAM is the socket type for TCP, the protocol that will be used to transport messages in the network.
- The .bind() method is used to associate the socket with a specific network interface and port number.
- The .listen() method has a *backlog* parameter. It specifies the number of unaccepted connections that the system will allow before refusing new connections.
- If your server receives a lot of connection requests simultaneously, increasing the backlog value may help by setting the maximum length of the queue for pending connections.
- The .accept() method blocks execution and waits for an incoming connection. When a client connects, it returns a new socket object representing the connection and a tuple holding the address of the client. The tuple will contain (host, port) for IPv4 connections or (host, port, flowinfo, scopeid) for IPv6
- After .accept() provides the client socket object *conn*, an infinite while loop is used to loop over blocking calls to conn.recv(). This reads whatever data the client sends and echoes it back using conn.sendall().

1.7 Client Code

Write the following code in the *client.py* file:

```
import socket

HOST = "127.0.0.1"          # The server's hostname or IP address
PORT = 65432                # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Hello, world")
    data = s.recv(1024)

print(f"Received {data!r}")
```

1.8 Viewing Socket State

After executing the *server.py* file, you see the current state of sockets on your host, use netstat. Type the command in Shell:

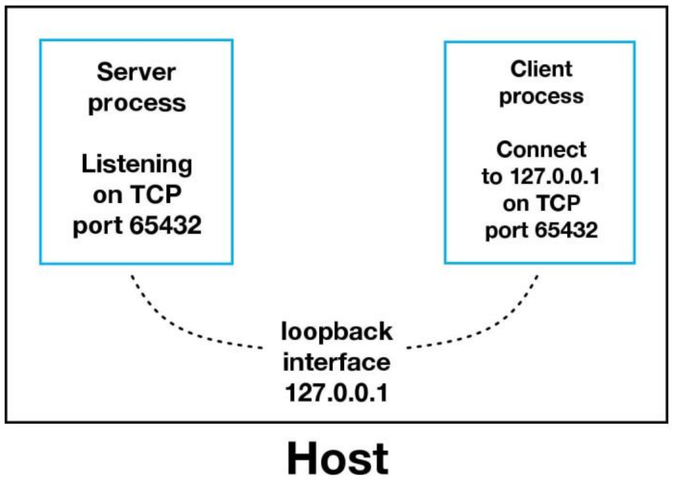
```
$ netstat -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  127.0.0.1.65432        *.*                    LISTEN
```

Here's a common error that you'll encounter when a connection attempt is made to a port with no listening socket:

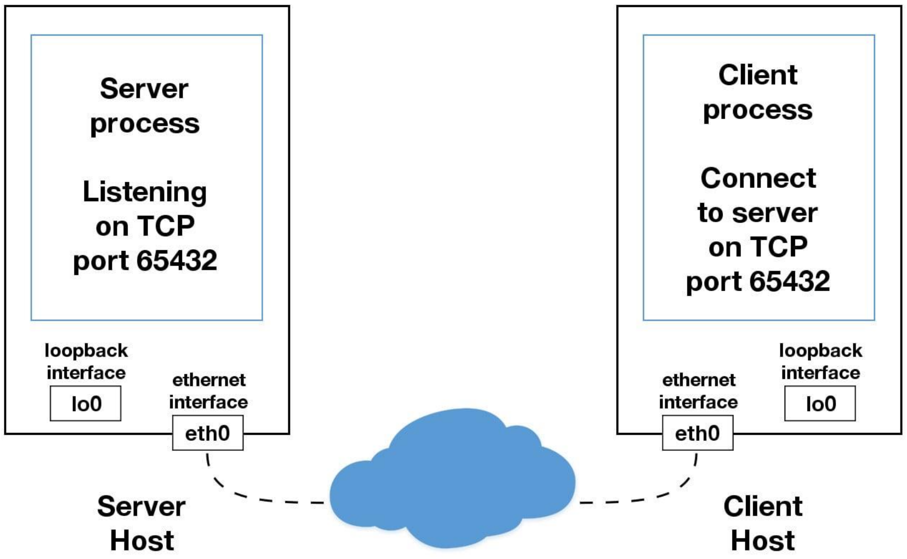
```
$ python client.py
Traceback (most recent call last):
  File "./echo-client.py", line 9, in <module>
    s.connect((HOST, PORT))
ConnectionRefusedError: [Errno 61] Connection refused
```

Either the specified port number is wrong, or the server isn't running. Or maybe there's a firewall in the path that's blocking the connection, which can be easy to forget about. You may also see the error Connection timed out. Get a firewall rule added that allows the client to connect to the TCP port!

1.9 Communication Breakdown


Server process
Listening on TCP port 65432
Client process
Connect to 127.0.0.1 on TCP port 65432
loopback interface 127.0.0.1
Host

In case your server and client are located on different location and connected through WAN or Internet:



1.10 More Detailed Example for Client and Server

Enter the following code in *server.py*:

```
import socket

def run_server():
    # create a socket object
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    server_ip = "127.0.0.1"        # Standard loopback interface address (localhost)
    port = 8000                    # Port to listen on (non-privileged ports are > 1023)

    server.bind((server_ip, port))  # bind the socket to a specific address and port
    server.listen(0)                # listen for incoming connections
    print(f"Listening on {server_ip}:{port}")

    client_socket, client_address = server.accept() # accept incoming connections
    print(f"Accepted connection from {client_address[0]}:{client_address[1]}")

    # receive data from the client
    while True:
        request = client_socket.recv(1024)
        request = request.decode("utf-8") # convert bytes to string

        # if we receive "close" from the client, then we break
        # out of the loop and close the conneciton
        if request.lower() == "close":
            # send response to the client which acknowledges that the
            # connection should be closed and break out of the loop
            client_socket.send("closed".encode("utf-8"))
            break

        print(f"Received: {request}")

        response = "accepted".encode("utf-8") # convert string to bytes
        client_socket.send(response)    # convert and send accept response to the client

    client_socket.close()   # close connection socket with the client
    print("Connection to client closed")
    server.close()   # close server socket

run_server()
```

And enter the following code in *client.py*:

```
import socket

def run_client():
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)   # create a socket object

    server_ip = "127.0.0.1"  # replace with the server's IP address
    server_port = 8000  # replace with the server's port number

    client.connect((server_ip, server_port))    # establish connection with server

    while True:
        msg = input("Enter message: ")      # input message and send it to the server
        client.send(msg.encode("utf-8")[:1024])    # String to Bytes

        response = client.recv(1024)    # receive message from the server
        response = response.decode("utf-8") # Convert bytes to String

        # if server sent us "closed" in the payload, we break out of the loop and close our socket
        if response.lower() == "closed":
            break

        print(f"Received: {response}")

    client.close()  # close client socket (connection to the server)
    print("Connection to server closed")

run_client()
```

1.10.1 Tasks for Student:

1. After successfully completing the above task. Now configure your computer as Server by running the *server.py* file on your computer and configure your friend's laptop as client by running the *client.py* code on that computer.
2. Also try to perform the task by changing the port to different values. Also try to change the port to some value less than 1023 (privileged port). Also edit the Windows Firewall if the system refuses the port.
3. Also try to perform the above tasks by changing the protocol from TCP to UDP.

Important: What's the difference between the actual IP of the Server/Client or 127.0.0.1 or 0.0.0.0? Try to change the IP of the server to these three variants and then try the same code. And express your findings.

1.11 Working with Multiple Clients – Multithreading

We have seen how a server responds to requests from a single client in the previous examples, however, in many practical situations, numerous clients may need to connect to a single server at once. This is where multithreading comes in. Multithreading is used in situations where you need to handle several tasks (e.g. execute multiple functions) concurrently (at the same time).

1.11.1 Multithreaded Server and Client Code Example

Create a python file e.g. named *multithreading_server.py* and place the following code in it and execute it.

```
import socket
import threading

def handle_client(client_socket, addr):
    try:
        while True:
            # receive and print client messages
            request = client_socket.recv(1024).decode("utf-8")
            if request.lower() == "close":
                client_socket.send("closed".encode("utf-8"))
                break
            print(f"Received: {request}")
            # convert and send accept response to the client
            response = "accepted"
            client_socket.send(response.encode("utf-8"))
    except Exception as e:
        print(f"Error when hanlding client: {e}")
    finally:
        client_socket.close()
        print(f"Connection to client ({addr[0]}:{addr[1]}) closed")


def run_server():
    server_ip = "127.0.0.1"
    port = 8000
    try:
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind((server_ip, port))
        server.listen()
        print(f"Listening on {server_ip}:{port}")

        while True:
            client_socket, addr = server.accept()
            print(f"Accepted connection from {addr[0]}:{addr[1]}")
            # start a new thread to handle the client
            thread = threading.Thread(target=handle_client, args=(client_socket, addr,))
            thread.start()
    except Exception as e:
        print(f"Error: {e}")
    finally:
        server.close()

run_server()
```

Create a python file e.g. named *multithreading_client.py* and place the following code in it and execute it.

```
import socket

def run_client():
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    server_ip = "127.0.0.1"
    server_port = 8000
    client.connect((server_ip, server_port))
    try:
        while True:
            # get input message from user and send it to the server
            msg = input("Enter message: ")
            client.send(msg.encode("utf-8")[:1024])

            # receive message from the server
            response = client.recv(1024)
            response = response.decode("utf-8")

            # if server sent us "closed" in the payload, we break out of
            # the loop and close our socket
            if response.lower() == "closed":
                break

            print(f"Received: {response}")
    except Exception as e:
        print(f"Error: {e}")
    finally:
        # close client socket (connection to the server)
        client.close()
        print("Connection to server closed")

run_client()
```

If you want to test the multi-client implementation, open several terminal windows for clients and one for the server. First start the server with `python multithreading_server.py`. After that start a couple of clients using `python multithreading_client.py`.

Task for Student:

With the help of this multithreading example, now repeate the Task 1 again.

1.12 Example of using UDP Protocol

UDP does not use .connect(), .sendall(), .recv() and .close() functions. It simply use .sendto() and .recvfrom() functions.

Here's simple code for server to receive UDP messages:

```
import socket

HOST = "127.0.0.1"
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.bind((HOST, PORT))
    while True:
        data, addr = s.recvfrom(1024)
        print("received message: %s" % data)
```

And this is the code for client to send UDP message to the server:

```
import socket

HOST = "127.0.0.1"
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.sendto((b"Hello, world"), (HOST,PORT))
```