CSE 232 Section B: Computer Networks: Programming Assignment 2: TCP Socket Programming Lab

Ritika Thakur (2022408) Swarnima Prasad (2022525)

1 Part 1: Complete Server Code

1.1 About TCP Server and How it is different from UDP

TCP is connection-oriented, meaning a connection must be established between the client and server before any data is transferred. This involves a handshake process. TCP ensures that all data sent is received in the correct order and without errors.

Performance Considerations:

1. TCP: Due to the reliable delivery and connection establishment, TCP is best suited for scenarios where all data must be received correctly and in order, such as web pages or file downloads.
2. UDP: Due to its lower overhead and faster communication, UDP is ideal for applications like video streaming, gaming, or DNS queries, where speed is critical, and some packet loss is acceptable.

Differences in Code:

1. TCP: Requires establishing a connection with accept(), uses send() and recv(), ensures reliable data transfer, and requires closing the connection.
2. UDP: Does not establish a connection, uses recvfrom() and sendto(), is faster but unreliable, and does not require closing the connection.

1.2 Server Code

```
from socket import *
import sys

# Stream instead of datagram in TCP
serverSocket = socket(AF_INET, SOCK_STREAM)

serverPort = 6789
serverSocket.bind(('127.0.0.1', serverPort))
serverSocket.listen(1)


while True:
    print('Ready to serve...')

    connectionSocket, addr = serverSocket.accept()

    try:
        message = connectionSocket.recv(1024).decode()

        if not message:
            connectionSocket.close()
            continue
```

```
23
24          filename = message.split()[1]
25          f = open(filename[1:])
26          outputdata = f.read()
27
28          # for part 1
29          connectionSocket.send("HTTP/1.1 200 OK\r\n\r\n".encode())
30          for i in range(0, len(outputdata)):
31              connectionSocket.send(outputdata[i].encode())
32
33          # For client in part 3 next 2 lines and comment out above code
34          # response = "HTTP/1.1 200 OK\r\n\r\n" + outputdata
35          # connectionSocket.send(response.encode())
36          # print(response)
37
38          connectionSocket.send("\r\n".encode())
39          f.close()
40          connectionSocket.close()
41
42      except IOError:
43          connectionSocket.send("HTTP/1.1 404 Not Found\r\n\r\n".encode())
44          connectionSocket.send("<html><body><h1>404 Not Found</h1></body></html>\r\n".
                encode())
45          connectionSocket.close()
46
47  serverSocket.close()
48  sys.exit()
```

- Last two lines are never executed

1.3 HTML Page Code

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Hello World</title>
8  </head>
9  <body>
10     <h1>This course is called Computer Networks</h1>
11     <p>This course is kind of hard.</p>
12 </body>
13 </html>
```

1.4 Outputs

1.4.1 Local Host

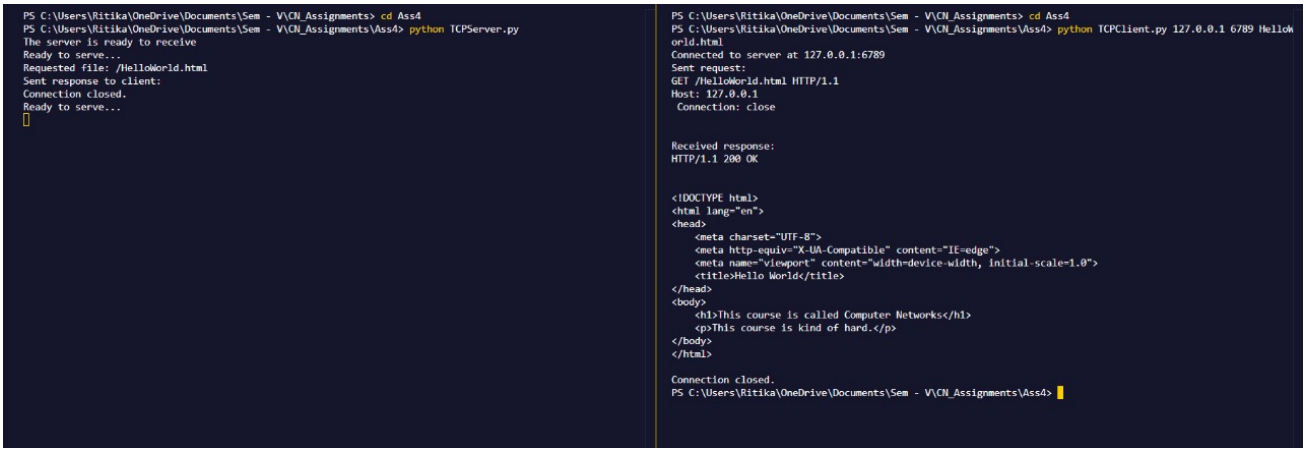

Figure 1: HTTP server for local host

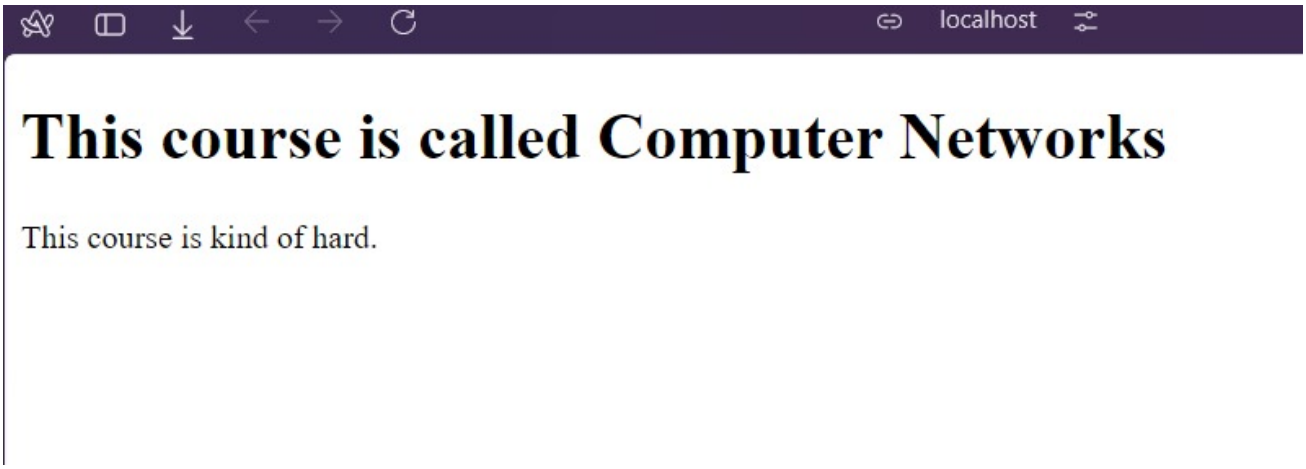

Figure 2: Web page when server is hosted at local host

1.4.2 Not Found Error Output

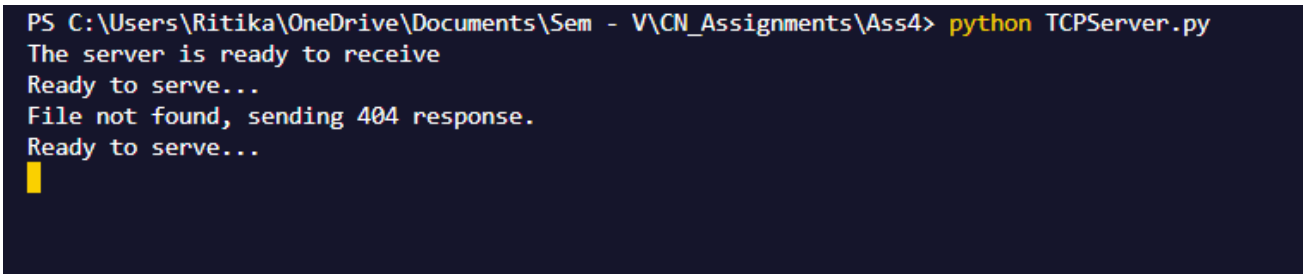

Figure 3: Not found error

404 Not Found

Figure 4: Web page when the requested html file is not present in the directory

1.4.3 Different Machines

Bind the server to the local IP address or 0.0.0.0 (which binds to all available interfaces).

```
serverSocket.bind(('0.0.0.0', serverPort))
```

Figure 5: Change made in server code

```
Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . :
   IPv6 Address. . . . . . . . . . . : 2401:4900:83ab:9fcd:93:2439:f40e:b323
   Temporary IPv6 Address. . . . . . : 2401:4900:83ab:9fcd:51a0:da38:c42e:446b
   Link-local IPv6 Address . . . . . : fe80::1bcc:e524:39d3:9912%11
   IPv4 Address. . . . . . . . . . . : 192.168.34.232
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : fe80::fcf6:ecff:fefa:c33c%11
                                       192.168.34.112
PS C:\Users\Ritika\OneDrive\Documents\Sem - V\CN_Assignments> cd Ass4
PS C:\Users\Ritika\OneDrive\Documents\Sem - V\CN_Assignments\Ass4> python TCPServer.py
Ready to serve...
Ready to serve...
```

Figure 6: Finding IP Address of server

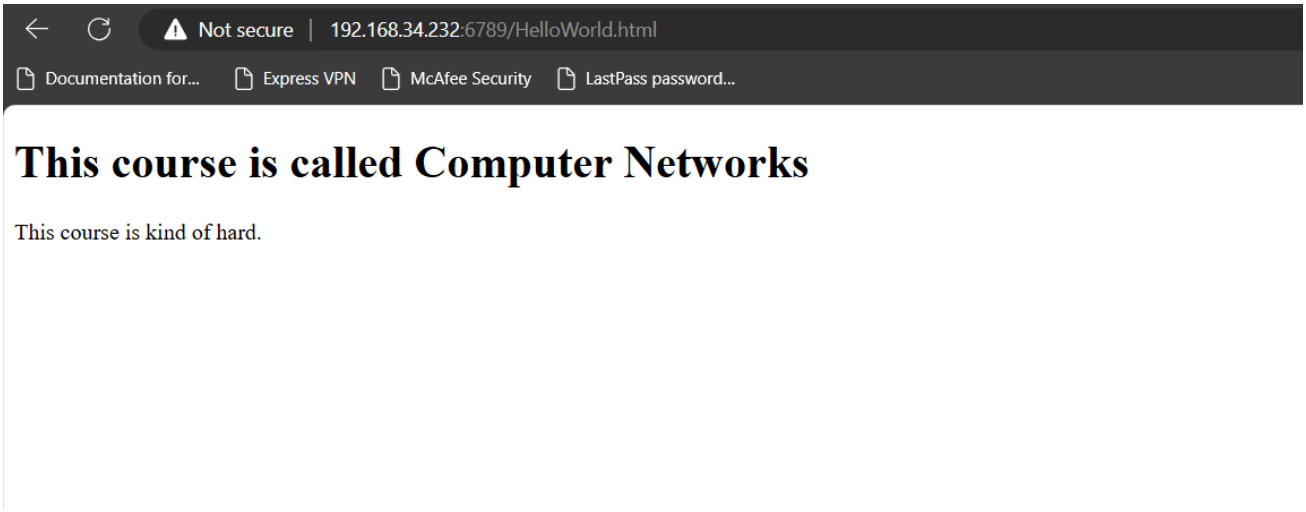

Figure 7: Hosted on other machine by giving IP address of first machine

2 Part 2: MultiThreaded TCP Server

2.1 Server Code

```
 import socket
import threading

class WebServer:
    def __init__(self, host='', port=6789):
        self.host = host
        self.port = port
        self.serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.threads = []

        try:
            self.serverSocket.bind((self.host, self.port))
            self.serverSocket.listen(5)
            print(f"Server listening on {self.host}:{self.port}...")
        except socket.error as e:
            print(f"Socket error during binding/listening: {e}")
            self.serverSocket.close()
            raise

    def start(self):
        try:
            while True:
                try:
                    connectionSocket, addr = self.serverSocket.accept()
                    print(f"Connected to: {addr}")

                    thread = threading.Thread(target=self.threaded_client, args=(
                        connectionSocket,))
                    thread.start()
                    self.threads.append(thread)

                except socket.error as e:
                    print(f"Socket error in main loop: {e}")
                    break
                except Exception as e:
```

```
                    print(f"Unexpected error in main loop: {e}")
                    break
        except KeyboardInterrupt:
            print("Server is shutting down...")
        finally:
            self.serverSocket.close()
            for thread in self.threads:
                thread.join()
            print("Server shutting down.")

    def threaded_client(self, connectionSocket):
        try:
            while True:
                try:
                    data = connectionSocket.recv(1024).decode()
                    if not data:
                        print("No data received; closing connection.")
                        break

                    print("Received request.")

                    # Simple HTTP request handling
                    lines = data.splitlines()
                    if len(lines) > 0:
                        # Getting the requested file from the HTTP request
                        filename = lines[0].split()[1]
                        if filename == "/":
                            filename = "/HelloWorld.html"

                        # Attempting to open the requested file
                        try:
                            print(f"Attempting to open file: {filename[1:]}")
                            with open(filename[1:], 'r') as f:
                                outputdata = f.read()

                            # Sending HTTP response header and content
                            response = "HTTP/1.1 200 OK\r\nConnection: keep-alive\r\n\
                                r\n" + outputdata
                            print("File found and response sent.")
                        except FileNotFoundError:
                            print("File not found, sending 404 response.")
                            response = "HTTP/1.1 404 Not Found\r\n\r\n<html><body><h1
                                >404 Not Found</h1></body></html>"
                        except Exception as e:
                            print(f"Error while opening file: {e}")
                            response = "HTTP/1.1 500 Internal Server Error\r\n\r\n<
                                html><body><h1>500 Internal Server Error</h1></body></
                                html>"

                        connectionSocket.send(response.encode())

                except socket.error as e:
                    print(f"Socket error in thread: {e}")
                    break
                except Exception as e:
                    print(f"Error in thread: {e}")
                    break
        finally:
            connectionSocket.close()
            print("Connection closed.\n")
```

```python
if __name__ == "__main__":
    try:
        server = WebServer(host='127.0.0.1', port=6789)
        server.start()
    except Exception as e:
        print(f"Failed to start server: {e}")
```

2.2 Outputs

2.2.1 Local Host

```
PS C:\Users\swarnima prasad\OneDrive\Desktop\CN_ass\CN_Assignments\Ass4> python multithreaded_TCPServer.py
Server listening on 127.0.0.1:6789...

 Connected to: ('127.0.0.1', 57338)
Received request.
Attempting to open file: HelloWorld.html
File found and response sent.

 Connected to: ('127.0.0.1', 57339)

 Connected to: ('127.0.0.1', 57340)
Received request.
Attempting to open file: HelloWorld.html
File found and response sent.
Received request.
Attempting to open file: HelloWorld.html
File found and response sent.
```

Figure 8: When multiple Tabs were opened

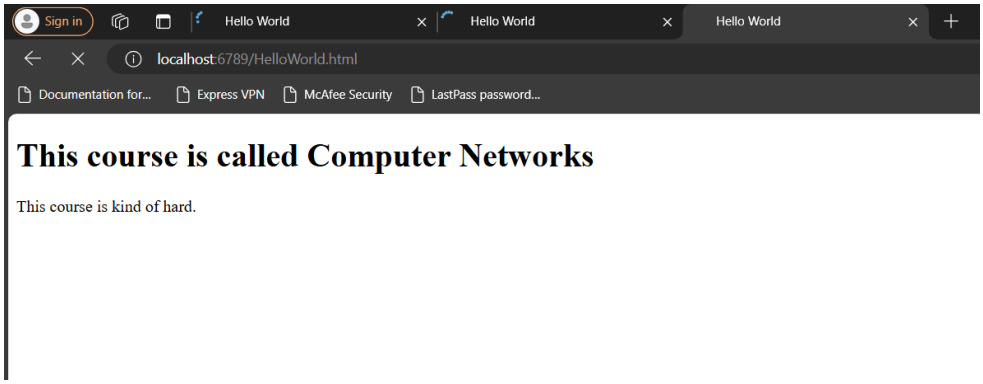

Figure 9: opening multiple tabs

```
 Connected to: ('127.0.0.1', 57672)
Received request.
Attempting to open file: HelloWorld.html
File found and response sent.
Received request.
Attempting to open file: HelloWorld.html
File found and response sent.
No data received; closing connection.
Connection closed.

No data received; closing connection.
Connection closed.

No data received; closing connection.
Connection closed.

Server is shutting down...
Server shutting down.
```

Figure 10: Graceful shutdown if CTRL + C is pressed and all tabs are closed

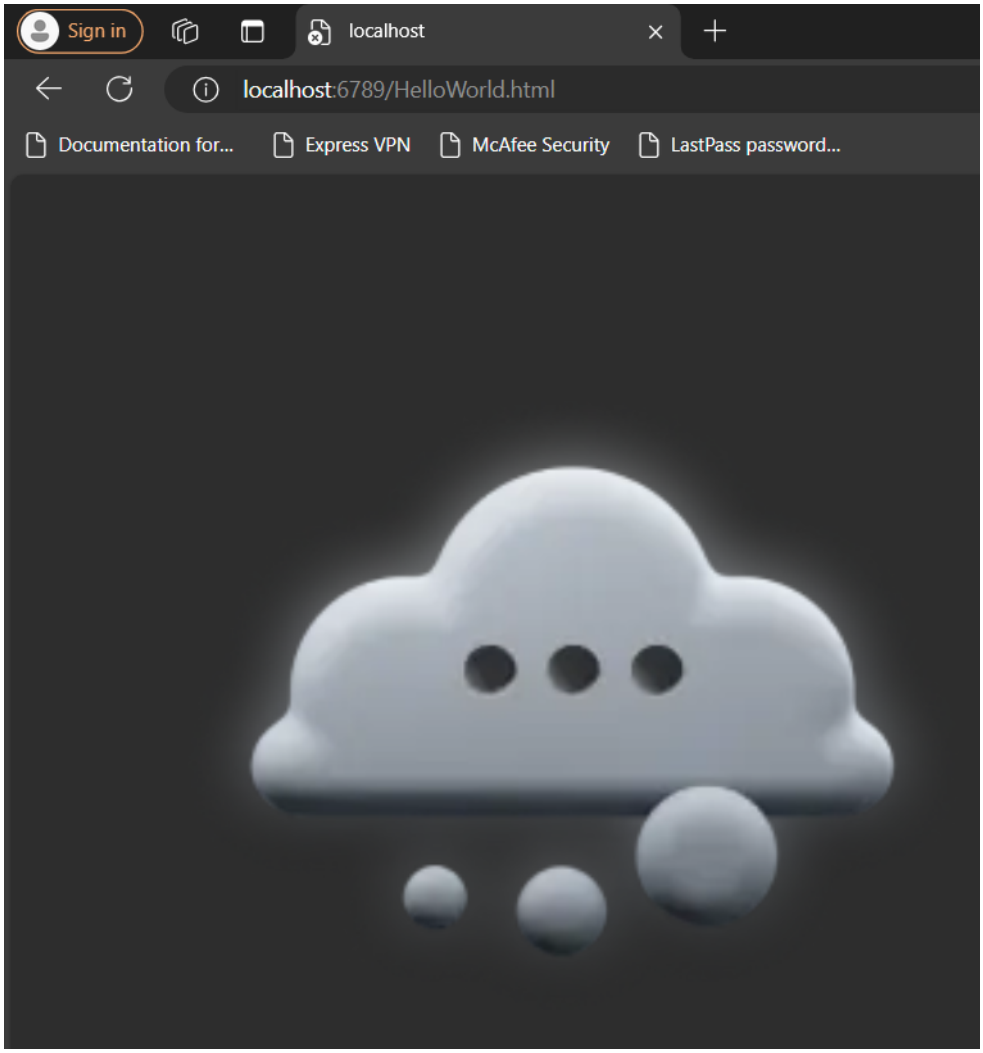

Figure 11: After Server Shut Down

2.2.2 Different Machines

```
PS C:\Users\Ritika\OneDrive\Documents\Sem - V\CN_Assignments> cd Ass4
PS C:\Users\Ritika\OneDrive\Documents\Sem - V\CN_Assignments\Ass4> python multithreaded_TCPServer.py
Server listening on 0.0.0.0:6789...
Connected to: ('192.168.34.67', 55355)
Received request.
Attempting to open file: HelloWorld.html
File found and response sent.
No data received; closing connection.
Connection closed.

Connected to: ('192.168.34.67', 55356)
Received request.
Attempting to open file: HelloWorld.html
File found and response sent.
No data received; closing connection.
Connection closed.

Connected to: ('192.168.34.67', 55370)
Received request.
Attempting to open file: HelloWorld.html
File found and response sent.
Connected to: ('192.168.34.67', 55371)
```

Figure 12: Multithreaded TCP Server hosted on 192.168.34.67

3 Client Code

3.1 Client Code

```python
import socket
import sys

class TCPClient:
    def __init__(self, host, port, filename):
        self.host = host
        self.port = port
        self.filename = filename
        self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    def connect(self):
        try:
            self.client_socket.connect((self.host, self.port))
            print(f"Connected to server at {self.host}:{self.port}")
        except socket.error as e:
            print(f"Error connecting to server: {e}")
            sys.exit(1)

    def create_request(self):
        request_line = f"GET /{self.filename} HTTP/1.1\r\n"
        headers = f"Host: {self.host}\r\n Connection: close\r\n\r\n"
        return request_line + headers

    def send_request(self):
        try:
            request = self.create_request()
            self.client_socket.sendall(request.encode())
            print("Sent request:")
            print(request)
```

```
30          except socket.error as e:
31              print(f"Error sending request: {e}")
32              sys.exit(1)
33
34      def receive_response(self):
35          try:
36              response = self.client_socket.recv(4096)
37              if not response:
38                  raise ValueError("No response received or server closed connection
                        unexpectedly.")
39              print("Received response:")
40              print(response.decode())
41          except socket.error as e:
42              print(f"Error receiving response: {e}")
43          except ValueError as ve:
44              print(f"Error: {ve}")
45          except Exception as e:
46              print(f"Unexpected error while receiving response: {e}")
47
48      def close(self):
49          try:
50              self.client_socket.close()
51              print("Connection closed.")
52          except socket.error as e:
53              print(f"Error closing the connection: {e}")
54
55  def main():
56      if len(sys.argv) != 4:
57          print("Usage: client.py <server_host> <server_port> <filename>")
58          sys.exit(1)
59
60      server_host = sys.argv[1]
61      server_port = int(sys.argv[2])
62      filename = sys.argv[3]
63
64      client = TCPClient(server_host, server_port, filename)
65
66      try:
67          client.connect()
68          client.send_request()
69          client.receive_response()
70      except Exception as e:
71          print(f"Error: {e}")
72      finally:
73          client.close()
74
75  if __name__ == "__main__":
76      main()
```

3.2 Outputs

3.2.1 Local Host

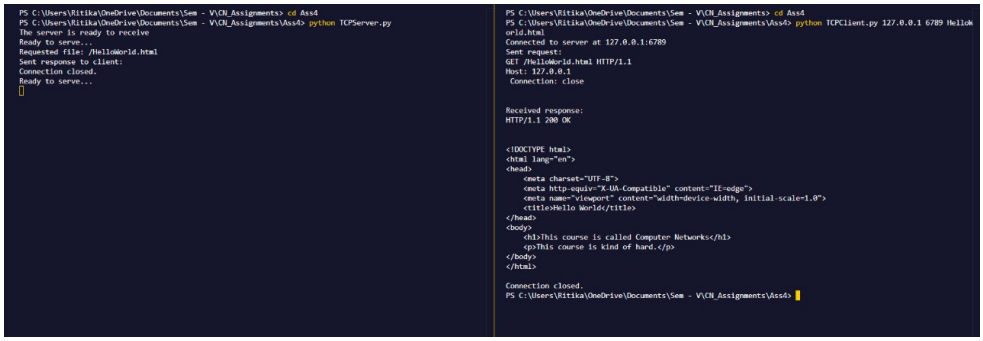

Figure 13: TCP Server and TCP client

Figure 14: Multithreaded server and TCP client

3.2.2 Different Machines

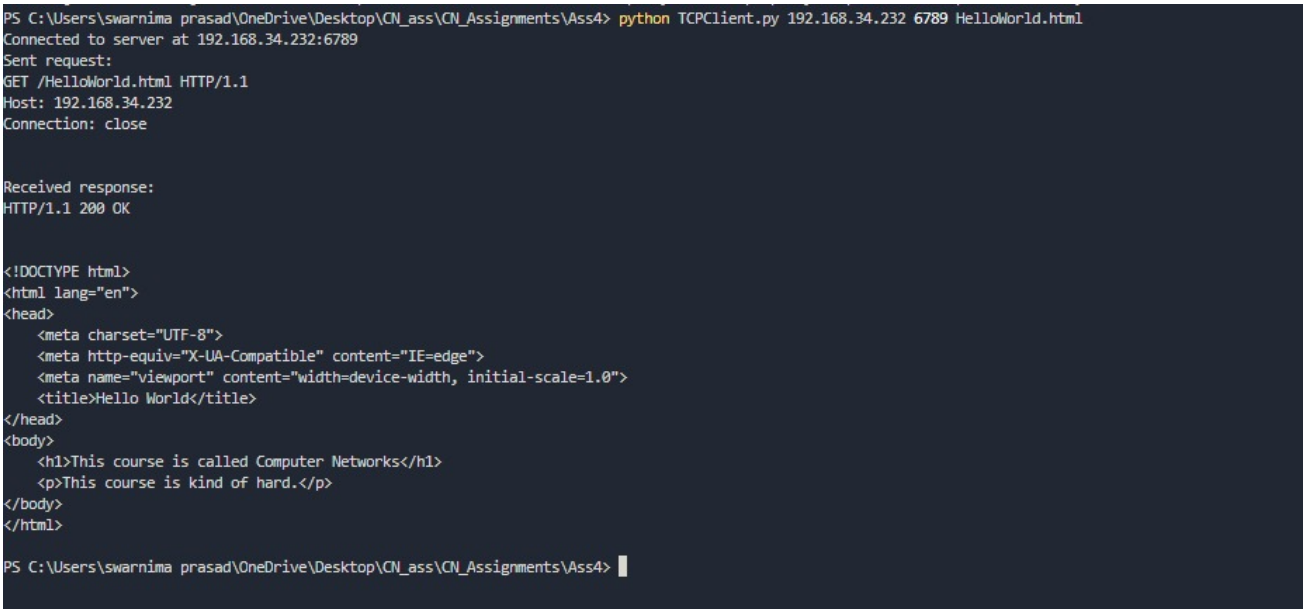

Figure 15: TCP Client when server is hosted on different machine

Figure 16: When the socket is binded to 'localhost' instead of 0.0.0.0 we notice that our client-server connection does not work across different machines

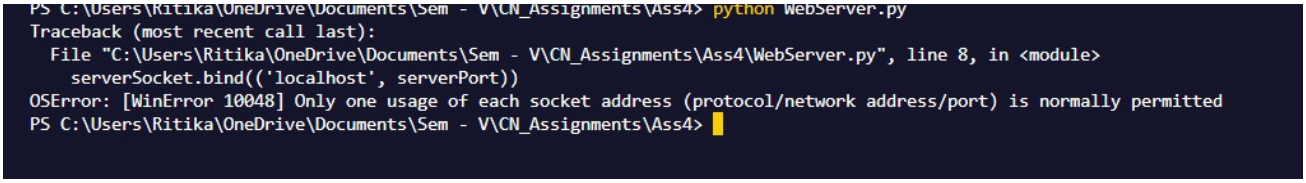

Figure 17: When we tried to host Multithreaded TCP Server after TCP Server was already hosted on the same port

4 Observations and Conclusions

For TCP Server:

1. Connection Establishment: TCP is connection-oriented, requiring a handshake between the client and server. The server listens for incoming connections (serverSocket.accept()), ensuring reliable communication once a client connects.

2. Local IP Address (127.0.0.1): The server is hosted locally using the loopback IP 127.0.0.1, allowing for internal testing. For remote access, the server's network IP would be used.

3. TCP Stream: The server uses SOCK_STREAM to establish a reliable, continuous data stream via TCP, ensuring that data is transferred in the correct order without errors.

4. Error Handling: The server returns a "404 Not Found" message if a requested file is missing. The connection is closed after sending the error, preventing the last two lines from executing.

5. Reliability: TCP ensures reliable, ordered delivery of data. If data is lost, TCP retransmits it, making it ideal for web servers. In contrast, UDP would be faster but lacks error handling and data integrity guarantees.

6. Overhead and Efficiency: TCP has higher overhead due to its connection management and error-checking processes. UDP has lower overhead but lacks reliability, making it unsuitable for this use case where data integrity is critical.

For TCP Multi-threaded Server:

1. Multithreading for Concurrent Clients: The server is designed to handle multiple clients simultaneously using threads. Each client connection runs on its own thread, allowing the server to manage multiple requests concurrently without blocking.

2. Socket Binding and Listening: The server binds to the host IP 0.0.0.0 and port 6789, meaning it listens on all available network interfaces. The listen(5) call allows the server to queue up to 5 incoming connection requests before refusing additional ones.

3. Error Handling: The server is robust against socket errors and exceptions, providing proper error messages and closing connections gracefully in the event of unexpected behavior (e.g., file not found, internal errors).

4. Graceful Shutdown: The server handles interruptions (like Ctrl+C) gracefully by terminating threads and closing sockets properly, ensuring no resources are left hanging.

5. Handling of Client Requests: For each client, the server processes HTTP GET requests, retrieves the requested file, and sends the appropriate HTTP response. If the file is not found, a "404 Not Found" error is sent back.

For TCP Client:

1. Object-Oriented Design: The client is structured using an object-oriented approach with a TCPClient class, making the code modular and easy to manage. The class encapsulates the client's behavior, including connecting to the server, creating and sending HTTP requests, receiving responses, and closing the connection.

2. TCP Connection Management: The client establishes a TCP connection using the socket.AF_INET and socket.SOCK_STREAM to the server at a given IP address and port. Proper error handling is implemented for connection failures, ensuring that the client can gracefully exit if the server is unreachable.

3. HTTP Request Handling: The client sends an HTTP GET request to the server, requesting the file specified in the command-line arguments. It constructs a simple HTTP request with headers, following the HTTP/1.1 protocol, including the host information.

4. Robust Error Handling: The code handles multiple exceptions such as socket errors (for connection, sending, and receiving), unexpected server disconnection, and general errors. This ensures that the client handles common issues gracefully and provides clear feedback to the user.

5. Response Handling: The client waits for the server's response using recv(4096) and prints the received data. It raises an error if no data is received or if the connection is closed unexpectedly, which helps detect issues with the server or network.

6. Connection Closure: The client ensures that the socket is properly closed after communication, with error handling for any issues during the closing process. This ensures that resources are released correctly.

7. Command-Line Interface: The client takes three command-line arguments (server host, port, and filename) and validates the input. If incorrect input is provided, it displays a usage message and exits, which improves user interaction.

5 References

- socket-programming-multi-threading-python
- Threading
- Python-tcp-server