FACULTY OF ENGINEERING TECHNOLOGY

ELECTRIC & COMPUTER ENGINEERING DEPARTMENT

ENCS3320 – Computer Networks

Project #1 Socket Programming

Prepared by:

Student name	ID	Section
Ahmad Owenah	1193117	4
Layan Salem	1221026	4
Tuleen Rimawi	1220211	4

Submission date: 1 December 2024

Table of content

Table of figures

Table of tables

Theory and Procedure

Task 1:

Networking Commands and Wireshark:

Modern computer networks rely on a combination of tools and protocols for diagnosing connectivity, analyzing data flow, and troubleshooting issues. This project uses essential networking commands and Wireshark, a packet analysis tool, to explore network operations.

Command Details

- IPCONFIG:

The ipconfig command displays the current IP configuration of a system, including IP address, subnet mask, default gateway, and DNS server addresses. For instance, running ipconfig /all shows detailed settings. This is particularly useful for verifying network configuration and resolving connectivity issues. [1]

- PING:

  The ping command is a fundamental tool for testing connectivity between two devices on a network. It sends Internet Control Message Protocol (ICMP) echo requests to the target and waits for replies., running ping measures round-trip time (RTT) and determines if the host is reachable. This command helps diagnose network delays and packet loss. [1]

- TRACERT:

  The tracert (or traceroute) command traces the route data packets take to a destination. It displays each hop (router) along the path and their respective IP addresses. executing tracert can reveal delays or failures at specific points in the network. This tool is invaluable for pinpointing network bottlenecks. [1]

- NSLOOKUP:

  The nslookup command queries Domain Name System (DNS) servers to resolve domain names into IP addresses or vice versa.using nslookup retrieves the IP address of the server hosting the domain. This is crucial for identifying DNS misconfigurations. [1]

- TELNET:

  Telnet is a protocol that establishes a text-based session with a remote host over a specified port. For instance, running telnet example.com 80 verifies connectivity to the host's HTTP service. Although largely replaced by SSH for security reasons, Telnet remains useful for basic port checks. [2]

- Wireshark:

  Wireshark is a packet analyzer that captures and displays network traffic for analysis. It enables detailed inspection of packets, such as DNS queries, aiding in troubleshooting advanced issues like dropped packets or protocol mismatches. [3]

Running Network Commands:

- Open a command prompt or terminal.
- Use ipconfig to display the system's IP address, subnet mask, gateway, and DNS server. Record this data for future reference.
- Use ping to test:
  1. A local device (your router’s IP) to ensure LAN connectivity.
  2. A public domain (e.g., google.com) to check internet connectivity.
  3.
- Run tracert for discover.engineering.utoronto.ca to view all intermediary routers. Note any high-latency hops that may indicate network issues.
- Use telnet to connect to discover.engineering.utoronto.ca on port 80. Check if the connection succeeds and the server responds with an HTTP banner or similar.

- Use nslookup for discover.engineering.utoronto.ca to retrieve its associated IP address and DNS server information.

Packet Capture with Wireshark:

- Install Wireshark and start a new capture on my active network interface.
- Perform a DNS query, visit google.com in a browser while Wireshark is running.
- Stop the capture and apply a DNS filter to isolate the relevant packets.
- Analyze the captured packets, noting details such as, DNS.

Task 2 +3:

- Socket Programming:

  Socket programming is the backbone of network communication, enabling applications to exchange data. In this project, both TCP and UDP protocols are explored, demonstrating their unique characteristics.

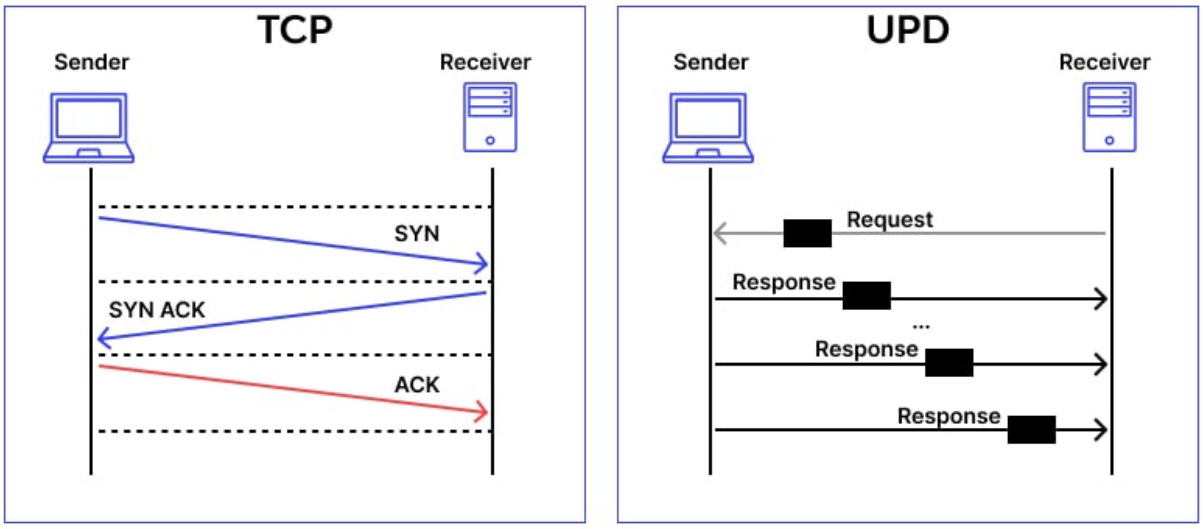


Figure 1 socket programming

- Transmission Control Protocol (TCP):

  TCP is a reliable, connection-oriented protocol that ensures data delivery in the correct order. It is ideal for scenarios requiring accuracy, such as web browsing or file transfers. For example, a TCP server binds to a port, listens for incoming connections, and exchanges data with clients. [4]

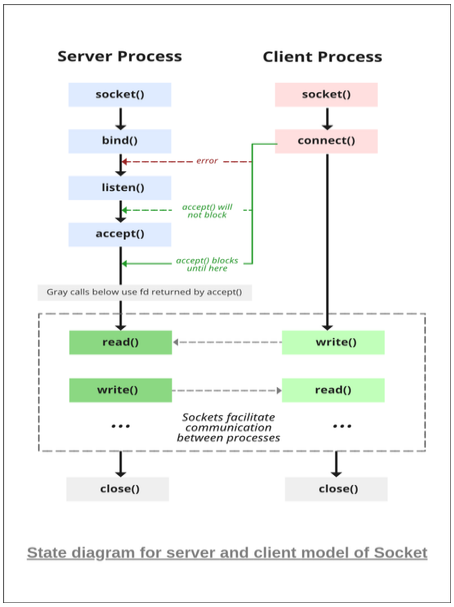


Figure 2 TCP datagram

- User Datagram Protocol (UDP):

  UDP is a connectionless protocol designed for speed and low latency. It is commonly used in real-time applications, such as video streaming or online gaming, where occasional packet loss is acceptable. [5]

Figure 3 UDP datagram

Task 2: Web server

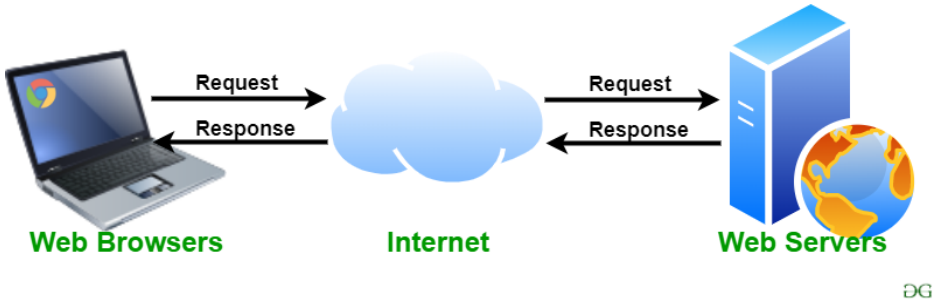
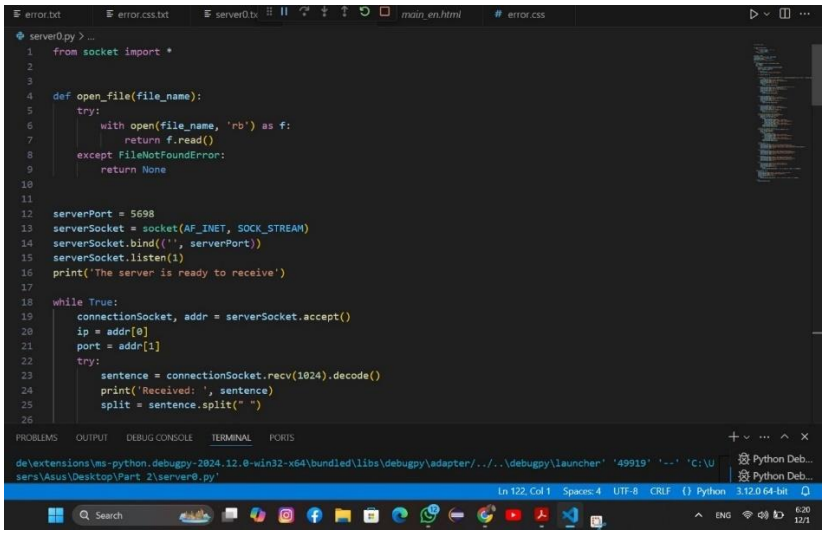


Figure 4 web server

In this part, we used socket programming to create a complete web server on port 5698 using the Python programming language, as well as HTML, CSS. First, we defined a server port which is 2035, then we created a socket instance and pass it two parameters, the first one is AF_INET which means to use IPv4 protocol, and the second one is SOCK_STREAM which means connection-oriented TCP protocol. As shown in figure 5. After creating the socket, we associated it with its local address, allowing clients to connect to the server using that address using bind(). Then we put the socket into listening mode to make it ready for the requests.



```
from socket import *


def open_file(file_name):
    try:
        with open(file_name, 'rb') as f:
            return f.read()
    except FileNotFoundError:
        return None


serverPort = 5698
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
print('The server is ready to receive')

while True:
    connectionSocket, addr = serverSocket.accept()
    ip = addr[0]
    port = addr[1]
    try:
        sentence = connectionSocket.recv(1024).decode()
        print('Received: ', sentence)
        split = sentence.split(" ")
```

Figure 5 task 2 server code 1

The server will accept and complete the connection by using accept(). Then we will receive the http request from the server and decode it to store it in the “sentence” variable.

1. English version of web server If the request is / or /index.html or /main_en.html or / en then the server should send main_en.html file. If the request is / or main_ar html then the main html file will be sent to the client. To send the html file, the server will first open that file then read it. Then it will send the header which contains of the encoded response status. Finally, the server will send the encoded file that it read previously. As shown in figures below :

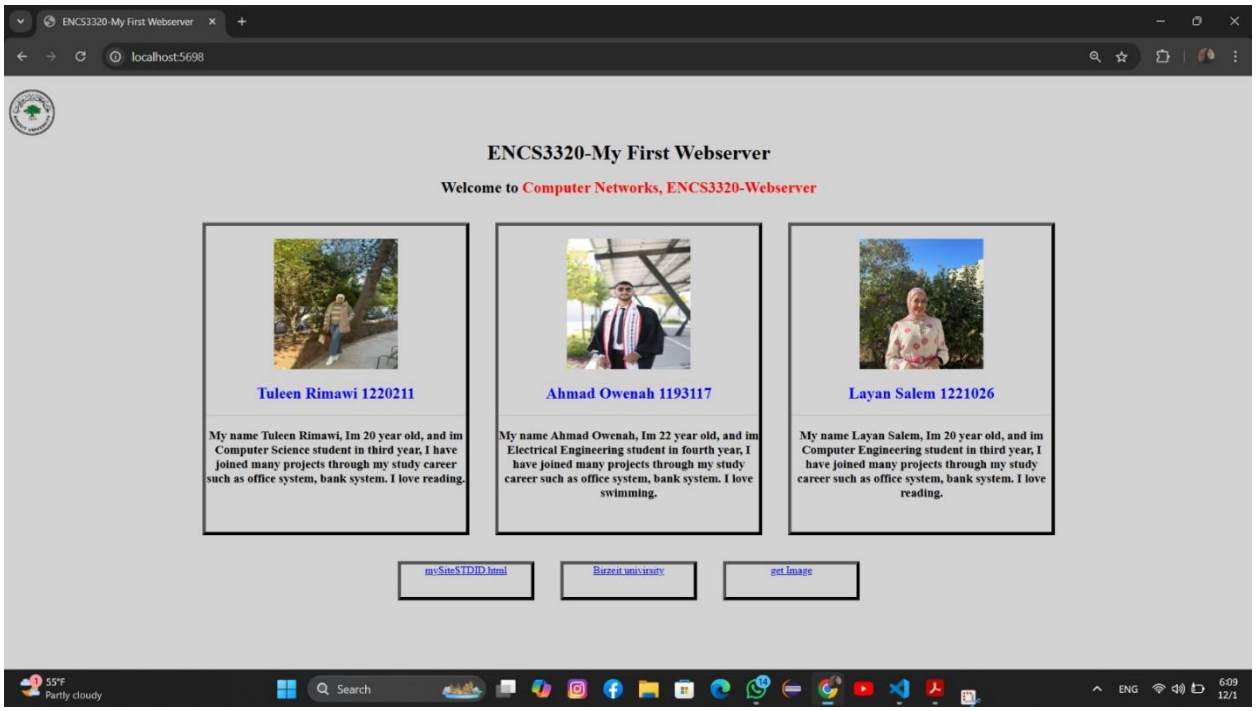


Figure 6 testing html_en

```
32
33  if split[1] == "/" or split[1].startswith("/en") or split[1].startswith("/main_en.html") or split[1].startswith(
34          "/index.html"):
35      connectionSocket.send(b'HTTP/1.1 200 OK\r\n')
36      connectionSocket.send(b'Content-Type: text/html\r\n')
37      connectionSocket.send(b'\r\n')
38      content = open_file('main_en.html')
39      if content:
40          connectionSocket.send(content)
41
```


Figure 7 server code 2

2. Arabic version of web server

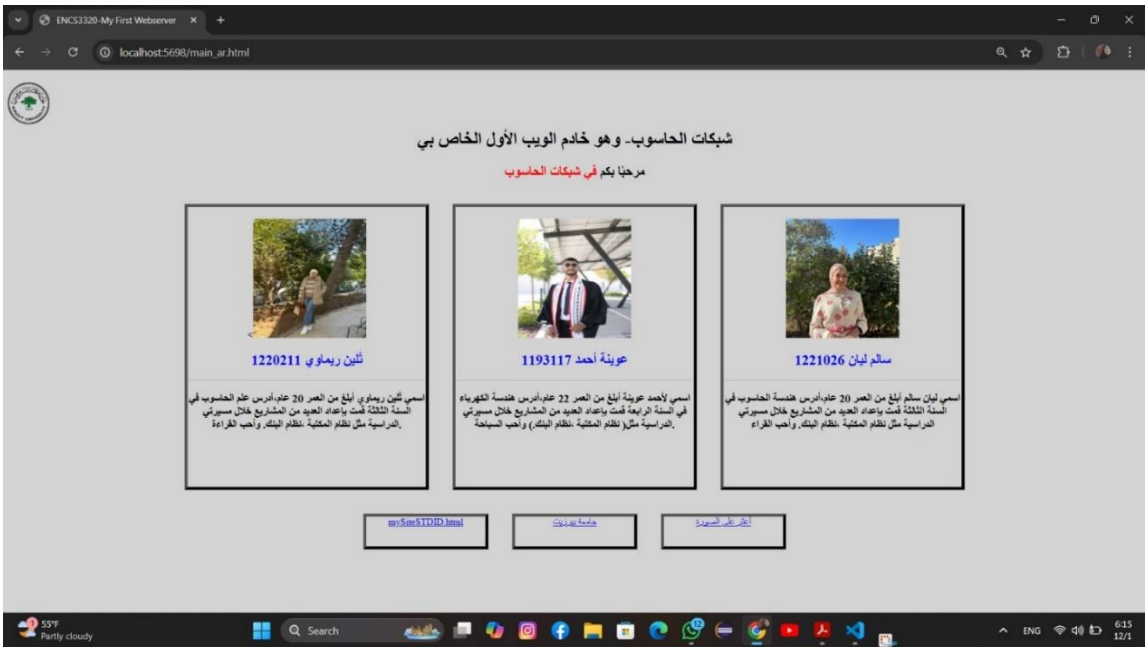


Figure 8 testing html_ar

```
41
42      elif split[1].startswith("/ar") or split[1].startswith("/main_ar.html"):
43          connectionSocket.send(b'HTTP/1.1 200 OK\r\n')
44          connectionSocket.send(b'Content-Type: text/html\r\n')
45          connectionSocket.send(b'\r\n')
46          content = open_file('main_ar.html')
47          if content:
48              connectionSocket.send(content)
49
50
51      elif split[1].endswith(".css"):
52          connectionSocket.send(b'HTTP/1.1 200 OK\r\n')
```

Ln 122, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.12.0 64-bit

Figure 9 server code 3

3. Request of html file:

```
58
59
60      elif split[1].endswith(".html"):
61          connectionSocket.send(b'HTTP/1.1 200 OK\r\n')
62          connectionSocket.send(b'Content-Type: text/html\r\n')
63          connectionSocket.send(b'\r\n')
64          content = open_file(split[1].lstrip("/"))
65          if content:
66              connectionSocket.send(content)
67
68      # Handle image requests via query parameter
69      elif split[1].startswith("/get_image?image-in="):
```

Ln 122, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.12.0 64-bit

Search ENG 6:17 12/1

Figure 10 request html file

If the request is /html then the main html file will be sent to the client. To send the html file, the server will first open that file then read it. Then it will send the header which contains of the encoded response status (which is OK), the encoded content type. Finally, the server will send the encoded file and open the local html file which called (main_en.html).

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Python Deb...
Python Deb...

```
de\extensions\ms-python.debugpy-2024.12.0-win32-x64\bundled\libs\debugpy\adapter/../..\debugpy\launcher' '49919' '--' 'C:\U
sers\Asus\Desktop\Part 2\server0.py'
The server is ready to receive
Received:  GET /main_ar.html HTTP/1.1
Host: localhost:5698
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: "Google Chrome";v="131", "Chromium";v="131", "Not_A Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/131.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed
-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: ar-AE,ar;q=0.9,en-GB;q=0.8,en;q=0.7,he-IL;q=0.6,he;q=0.5,en-US;q=0.4


Received:  GET /main_CSS.css HTTP/1.1
Host: localhost:5698
Connection: keep-alive
```

Ln 122, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.12.0 64-bit

Search ENG 6:18 12/1

Figure 11 Response of html request

4. Request the CSS file: This code handles the case where the file name contains ".css". If true, it sends an HTTP response header indicating success (200 OK) and specifies that the content type is CSS. After that, it sends the contents of the specified CSS file as the response body.

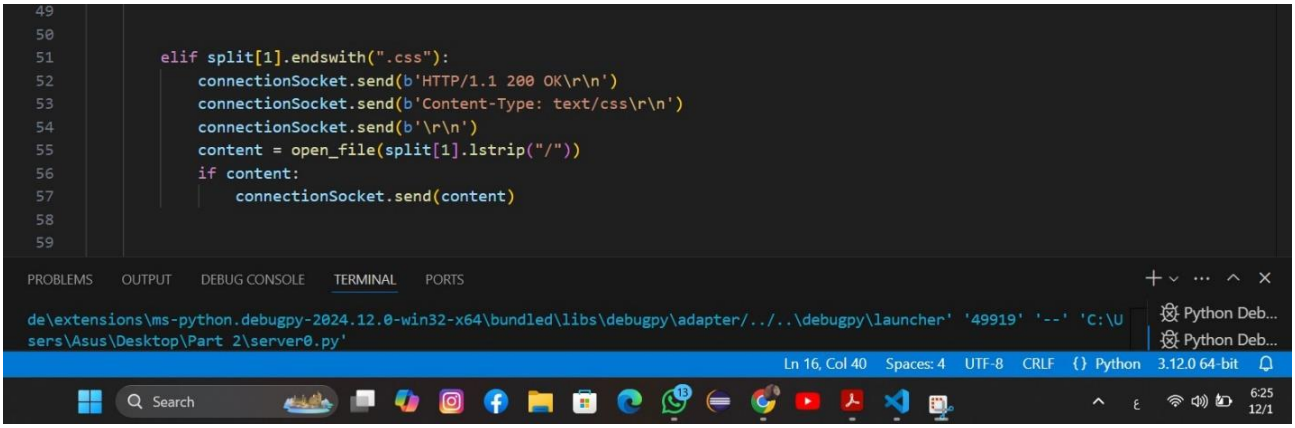


Figure 12 Request of CSS request

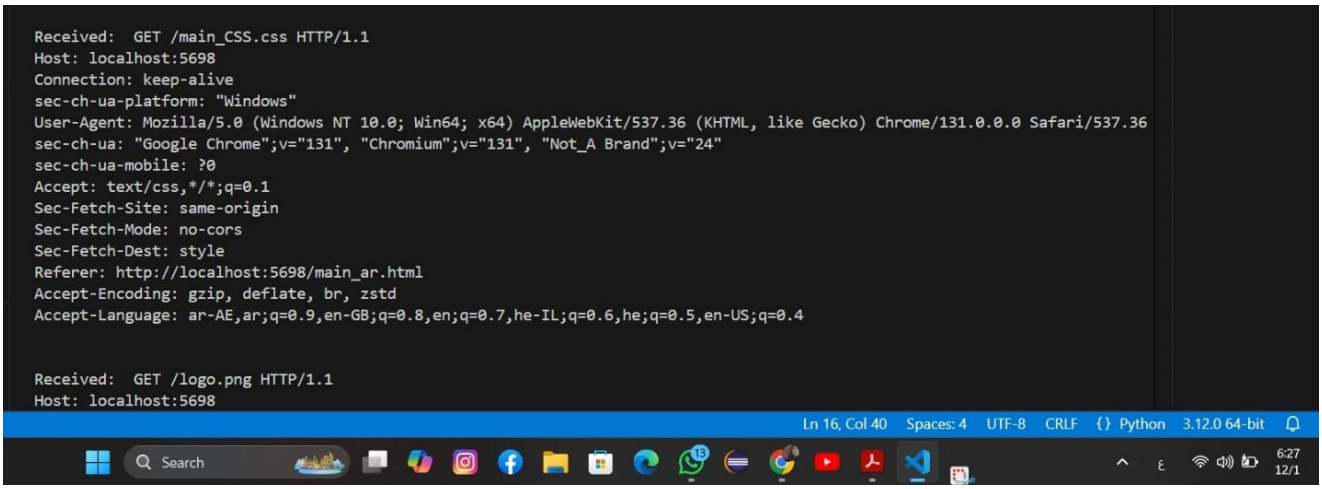
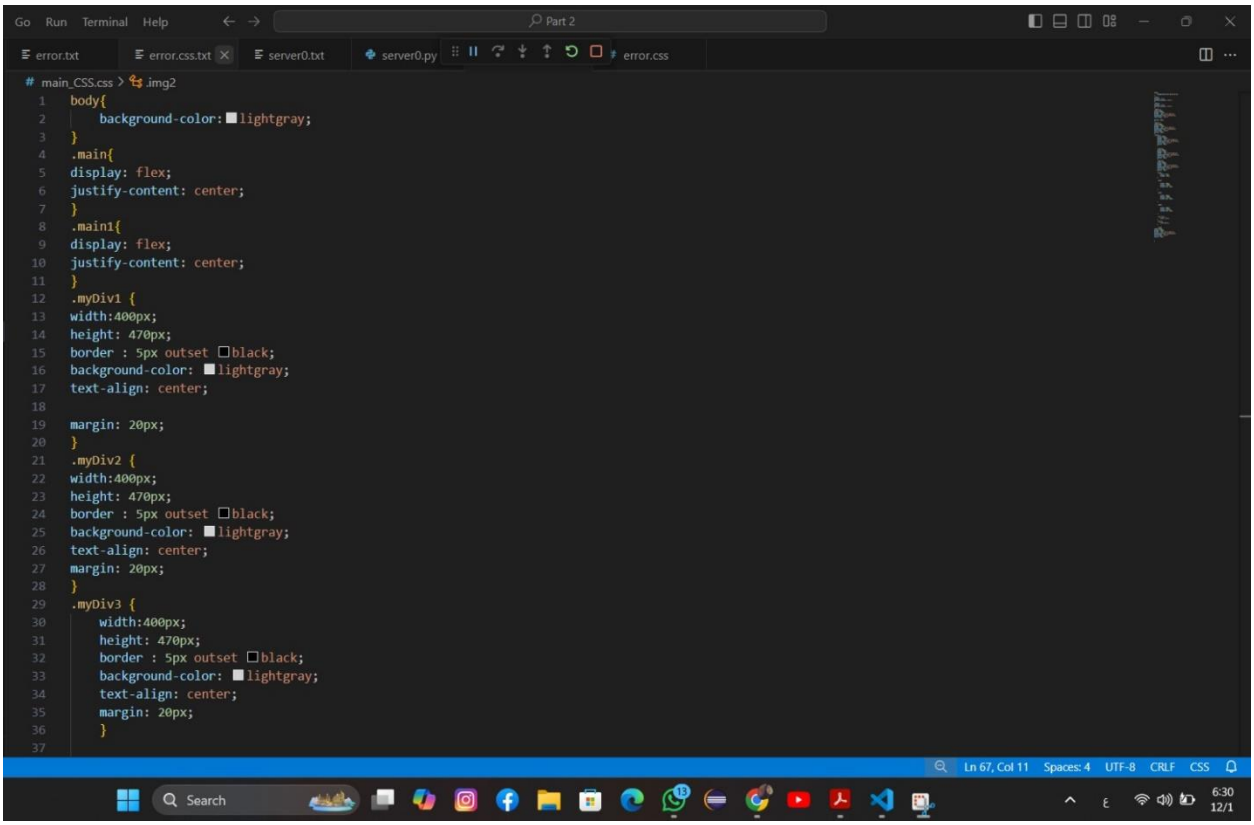


Figure 13 Response of CSS request

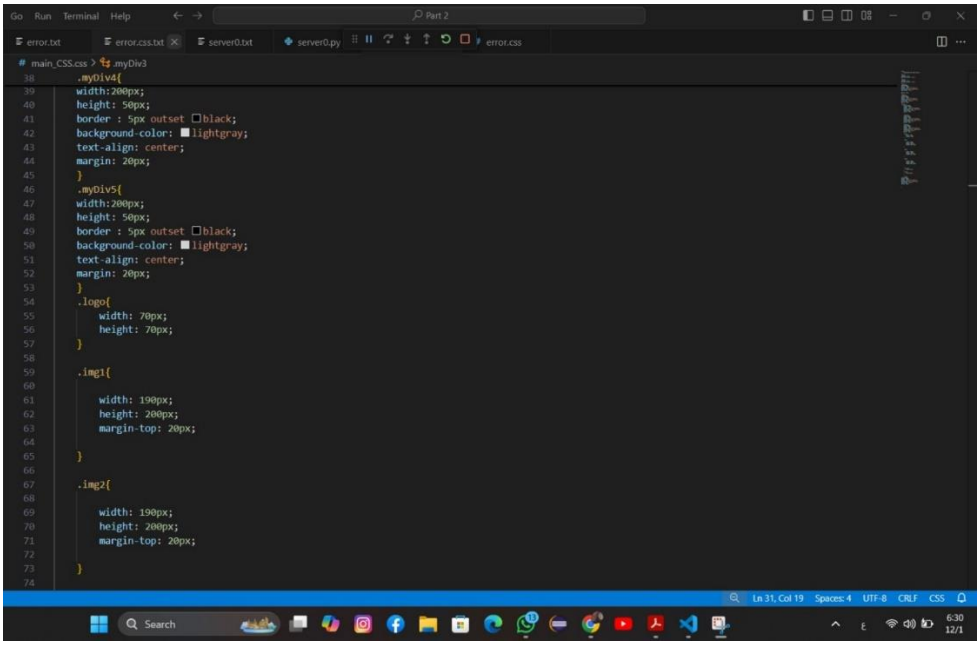
The Style of CSS:



```
body{
    background-color: lightgray;
}
.main{
display: flex;
justify-content: center;
}
.main1{
display: flex;
justify-content: center;
}
.myDiv1 {
width:400px;
height: 470px;
border : 5px outset black;
background-color: lightgray;
text-align: center;

margin: 20px;
}
.myDiv2 {
width:400px;
height: 470px;
border : 5px outset black;
background-color: lightgray;
text-align: center;
margin: 20px;
}
.myDiv3 {
    width:400px;
    height: 470px;
    border : 5px outset black;
    background-color: lightgray;
    text-align: center;
    margin: 20px;
    }
```

Figure 14 style of css 1



```
.myDiv4{
width:200px;
height: 50px;
border : 5px outset black;
background-color: lightgray;
text-align: center;
margin: 20px;
}
.myDiv5{
width:200px;
height: 50px;
border : 5px outset black;
background-color: lightgray;
text-align: center;
margin: 20px;
}
.logo{
    width: 70px;
    height: 70px;
}

.img1{

    width: 190px;
    height: 200px;
    margin-top: 20px;

}

.img2{

    width: 190px;
    height: 200px;
    margin-top: 20px;

}
```

Figure 15 style of css 2

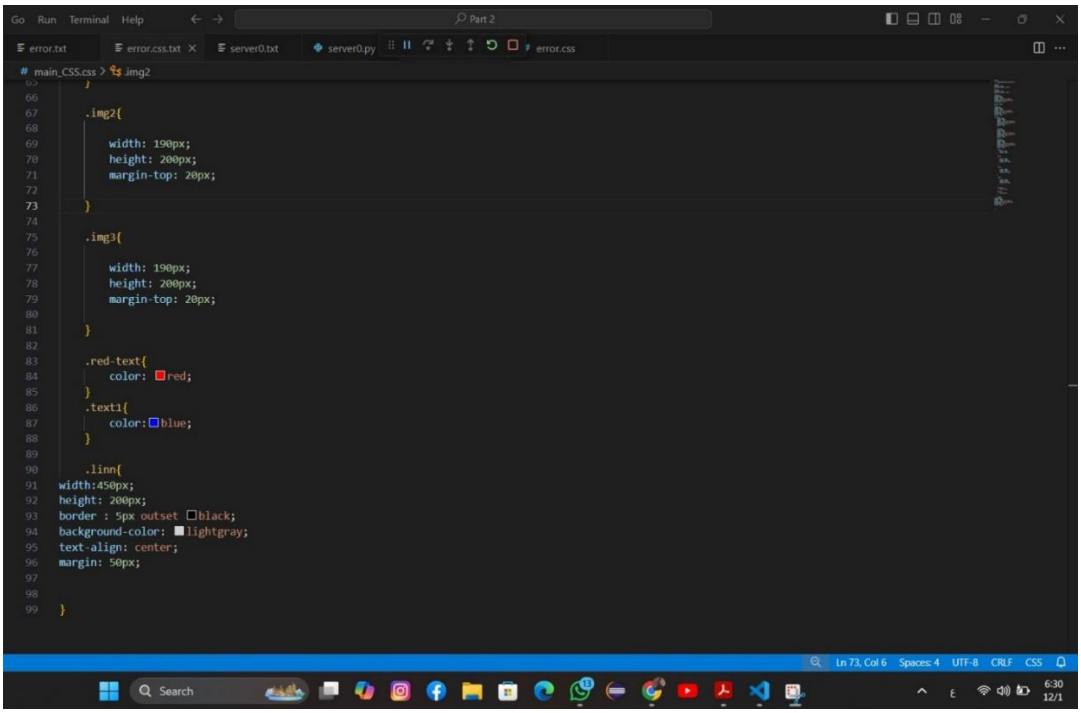


Figure 16 style of css 3

-

5. Request an image with PNG format This code checks if the file name contains ".png". If it does, it opens the file, sends an HTTP response header indicating success (200 OK), and specifies that the content type is a PNG image. After that, it sends the contents of the PNG file. As code bellow:

```
# Handle direct image file requests
elif split[1].endswith(".jpg") or split[1].endswith(".png"):
    file = split[1].lstrip("/")
    content = open_file(file)
    if content:
        if file.endswith(".jpg"):
            connectionSocket.send(b'HTTP/1.1 200 OK\r\n')
            connectionSocket.send(b'Content-Type: image/jpeg\r\n')
        elif file.endswith(".png"):
            connectionSocket.send(b'HTTP/1.1 200 OK\r\n')
            connectionSocket.send(b'Content-Type: image/png\r\n')
        connectionSocket.send(b'\r\n')
        connectionSocket.send(content)
```

Figure 17 request image

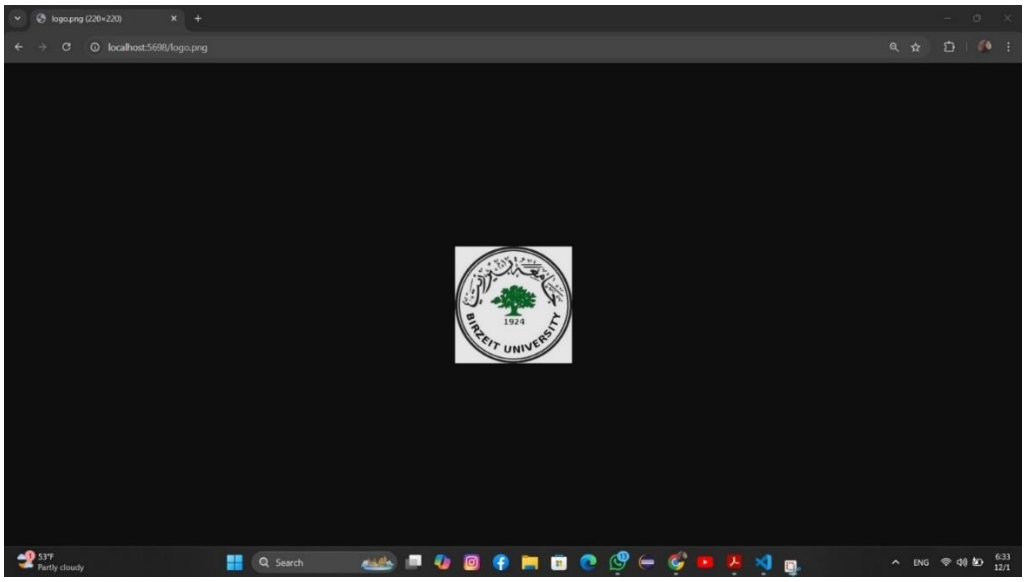


Figure 18 image

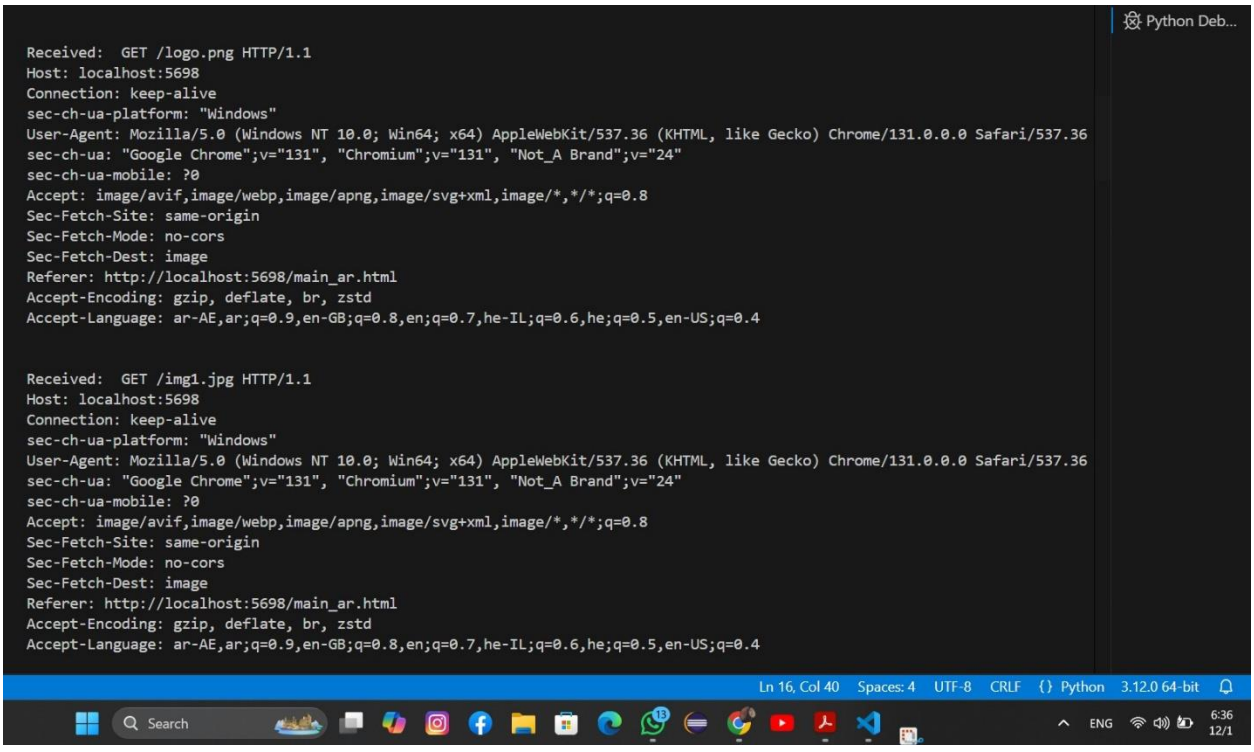


Figure 19 Response of png

6. Request an image with JPG format: This code checks if the file name contains ".jpg". If it does, it opens the file, sends an HTTP response header indicating success (200 OK), and specifies that the content type is a JPEG image. After that, it sends the contents of the JPEG file as the response body. As code bellow:

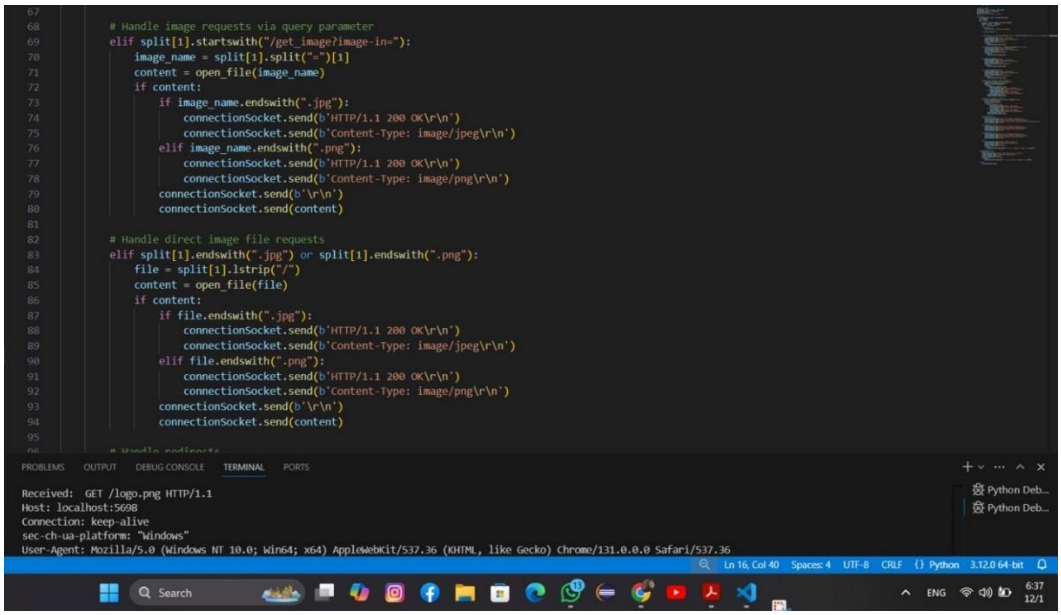


Figure 20 request JPG

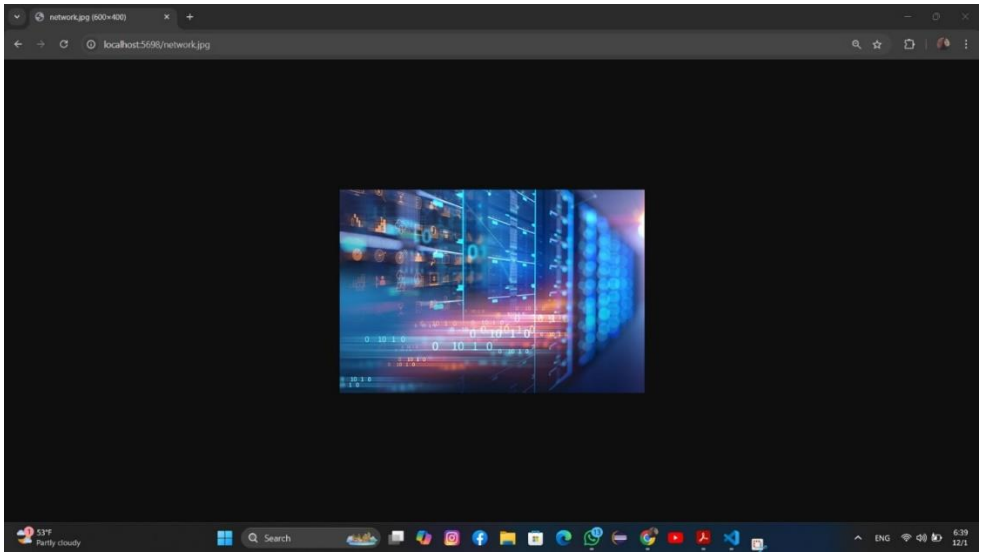


Figure 21 JPG image

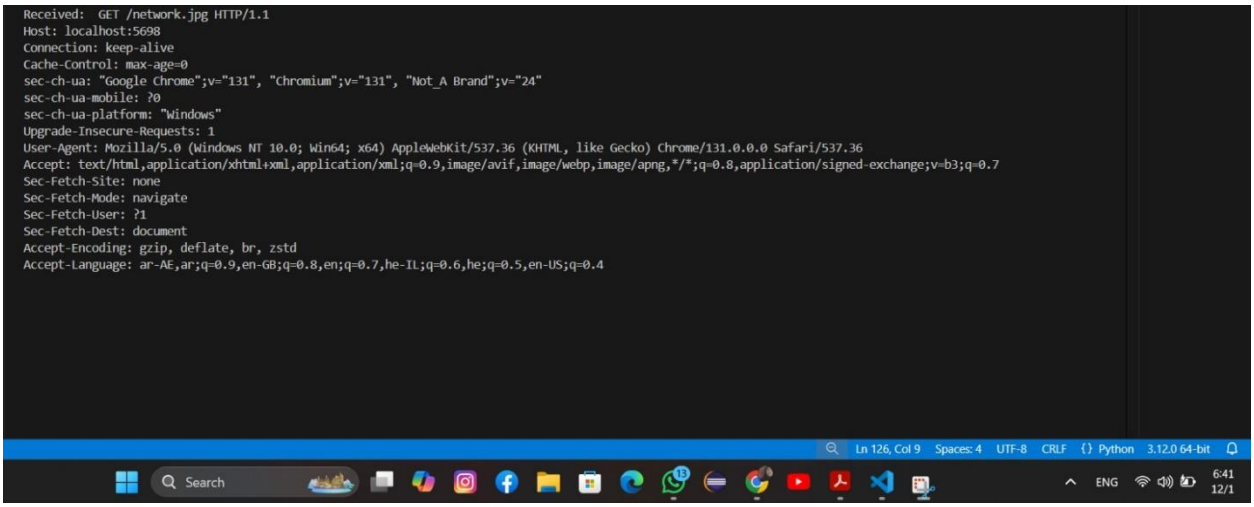


Figure 22 Response of image.jpg request

7. Store some images in a folder

8. Display image :

This site made for getting an images by writing the name of the image in box then select the button (get image) to display.

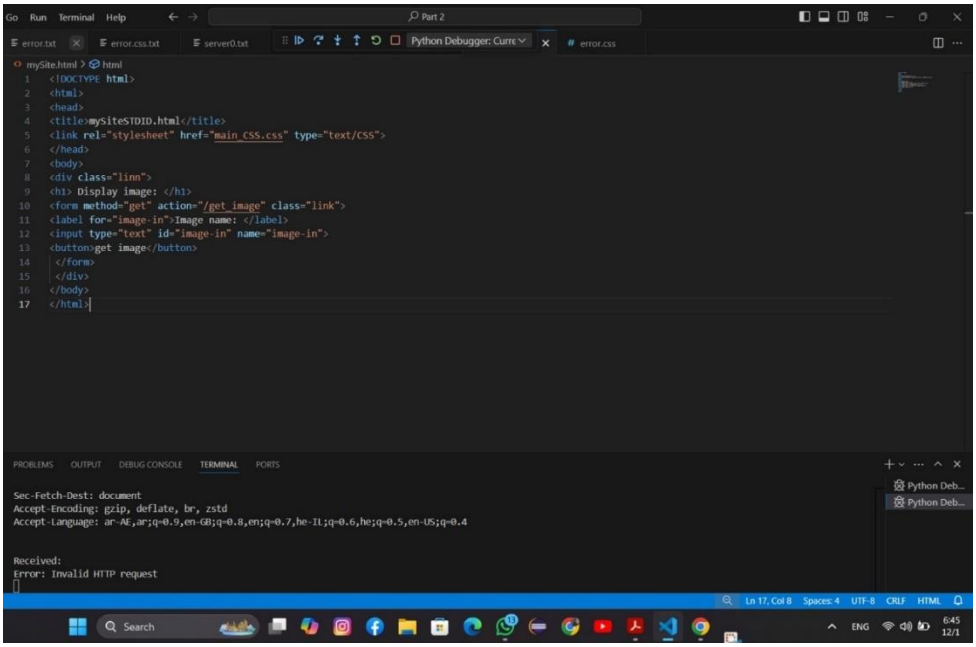


Figure 23 code to display image in english

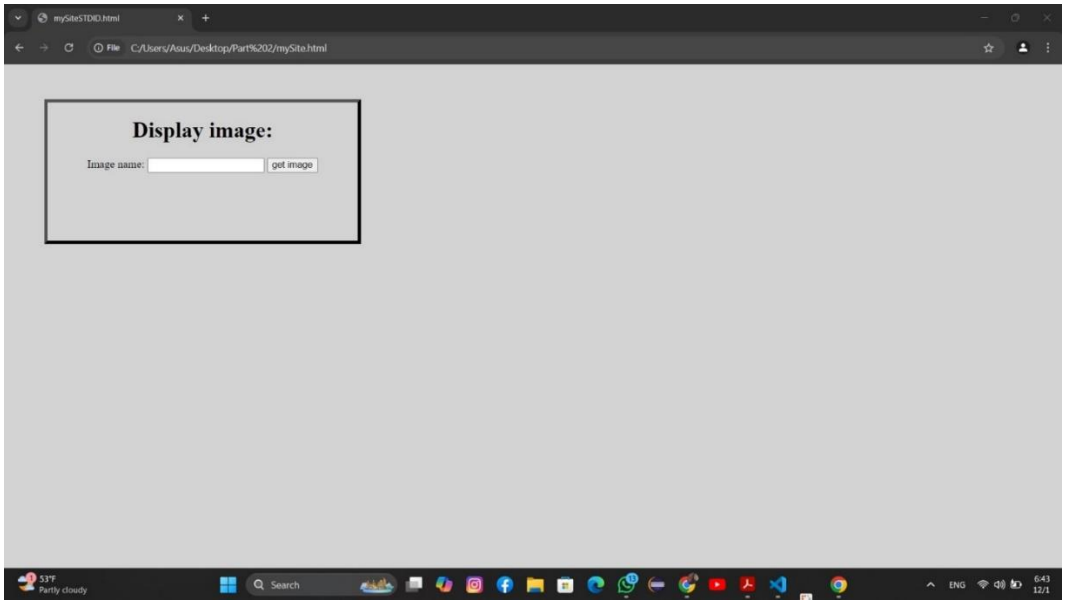


Figure 24 display image in English

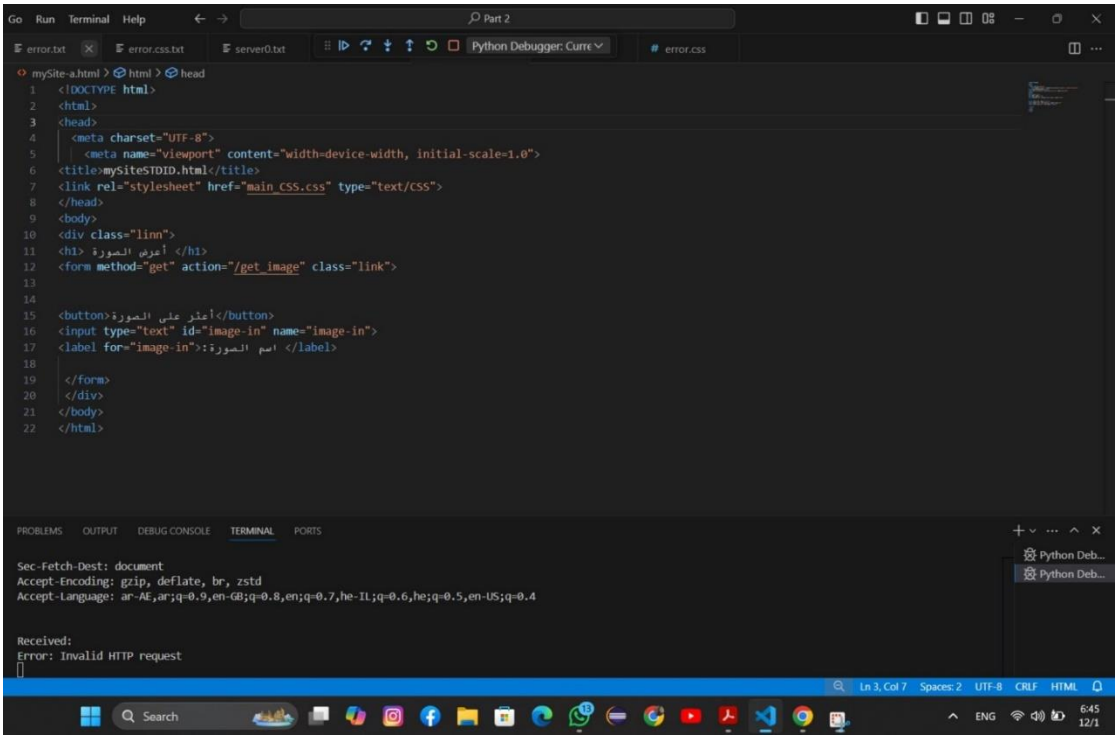


Figure 25 code to display image (Arabic)

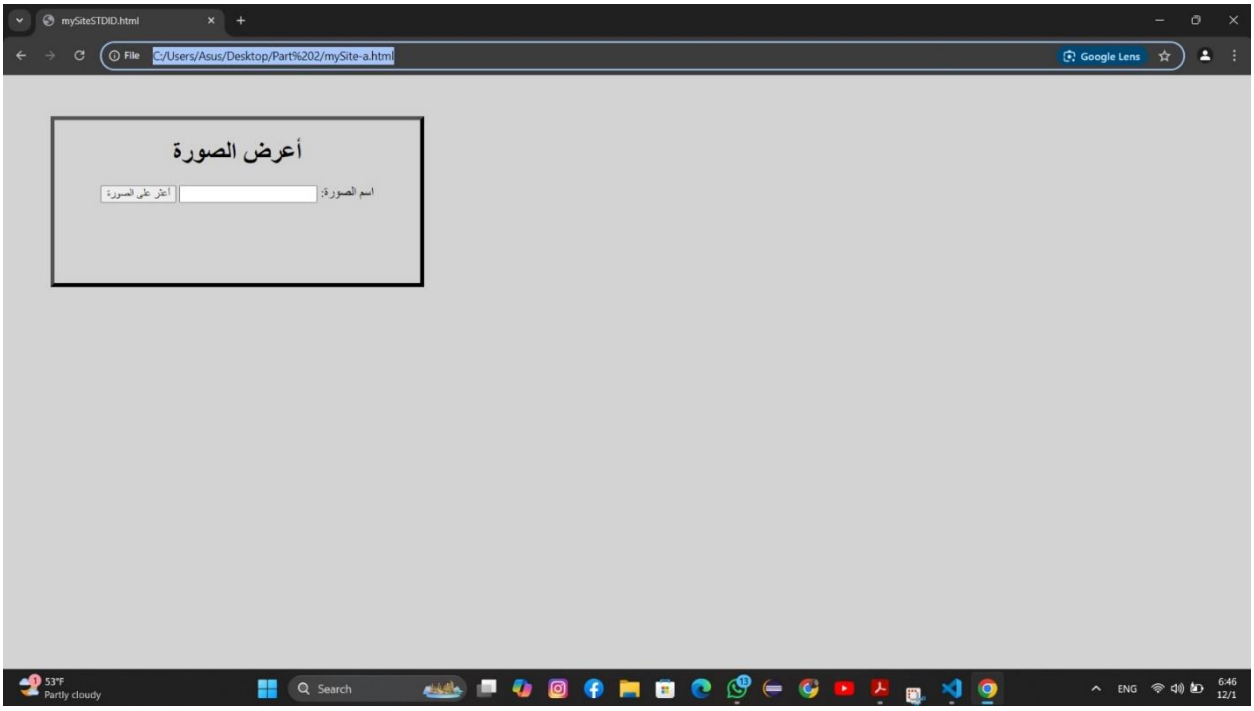


Figure 26 display image in Arabic

9. 307 Temporary Redirect By using the status code 307 Temporary Redirect to redirect the following: a) /bo request : This code checks if the file is equal to "so". If it is, the server sends an HTTP response with a 307 Temporary Redirect status. It indicates that the requested resource is temporarily located at a different URL. The server also sets the content type to HTML and redirects the client to "https://gaia.cs.umass.edu/kurose_ross/index.php " by specifying the Location header, as bellow:

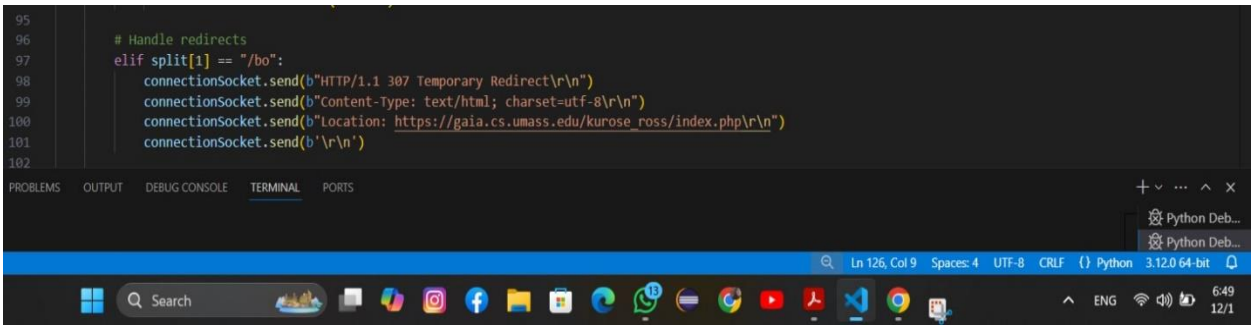


Figure 27 handle redirects

Directly transmitted to Computer Networking after sending

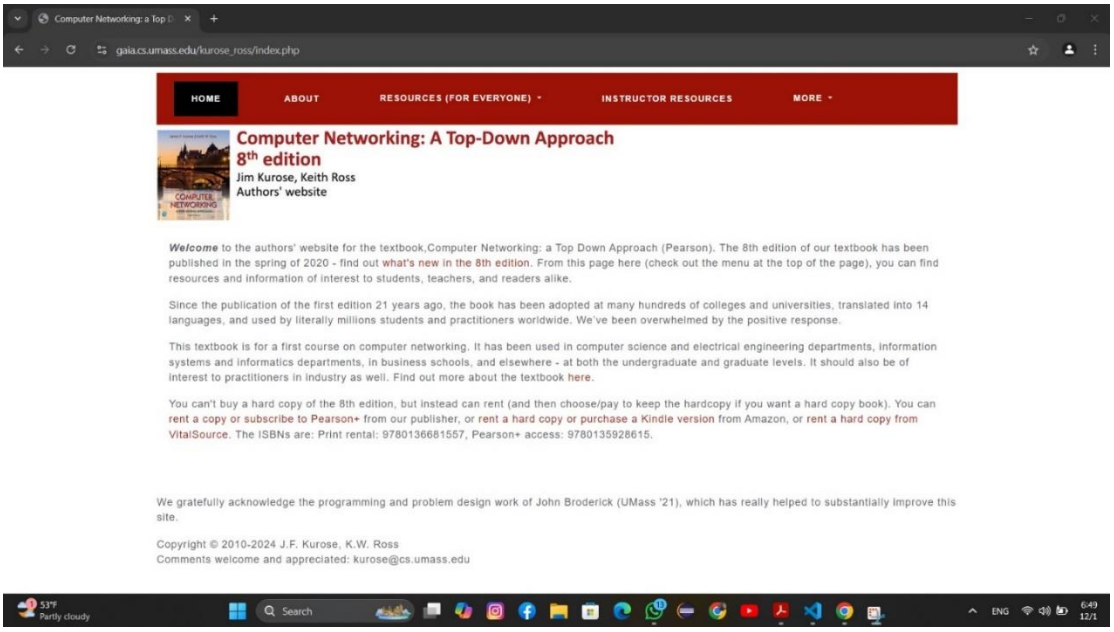


Figure 28 redirect testing

This code in figure 26 checks if the file is equal to "itc". If it is, the server sends an HTTP response with a 307 Temporary Redirect status, indicating a temporary redirection. It sets the content type to HTML and redirects the client to "https://itc.birzeit.edu/" by specifying the location header.

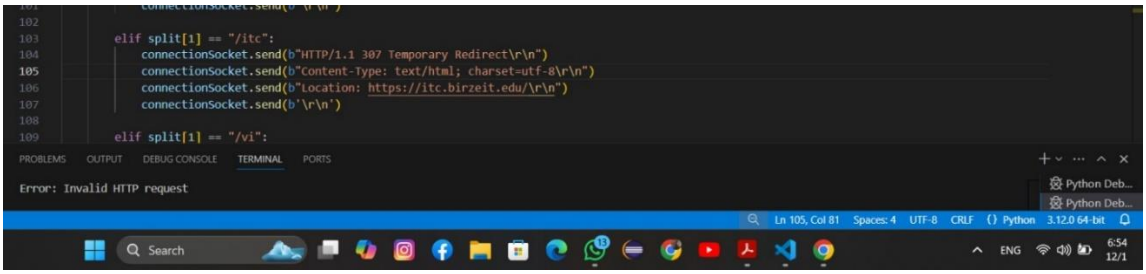


Figure 29 redirect code 1

- Directly transmitted to itc website after sending

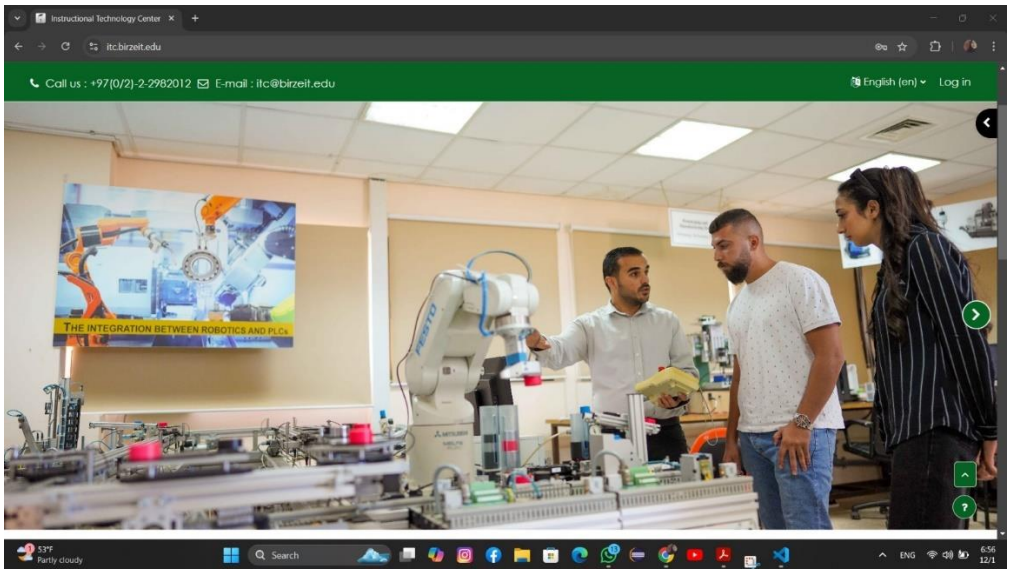


Figure 30 testing redirect to itc

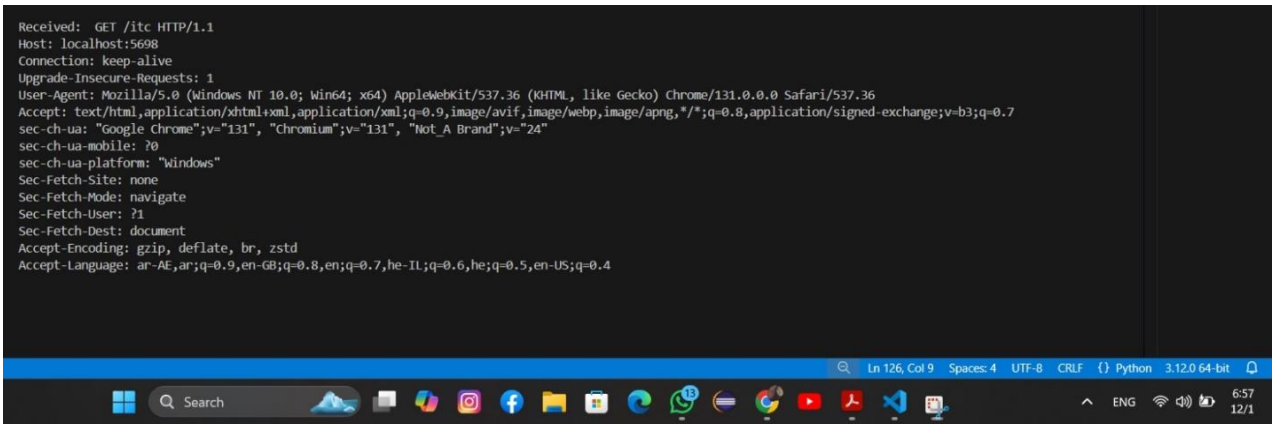


Figure 31 Response of itc

10. Wrong request:

This code handles cases where the requested file does not match any of the previous conditions. In such cases, the server sends an HTTP response with a 404 Not Found status, indicating that the requested resource is unavailable. The content type is set to HTML, and the server sends the contents of an "error.html" file along with a custom message that includes the client's IP address and port number.

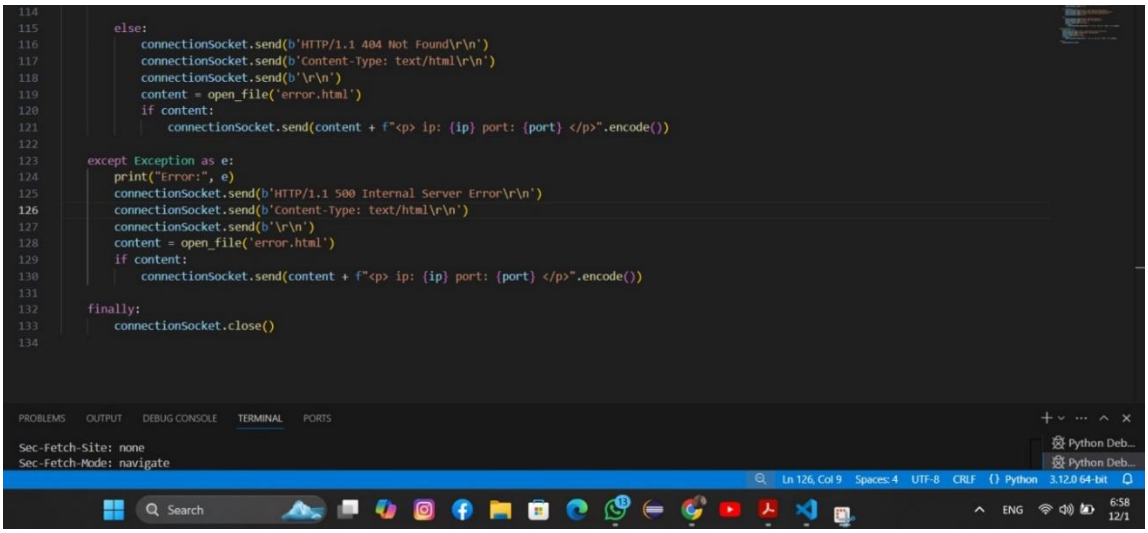


Figure 32 wrong request handle code

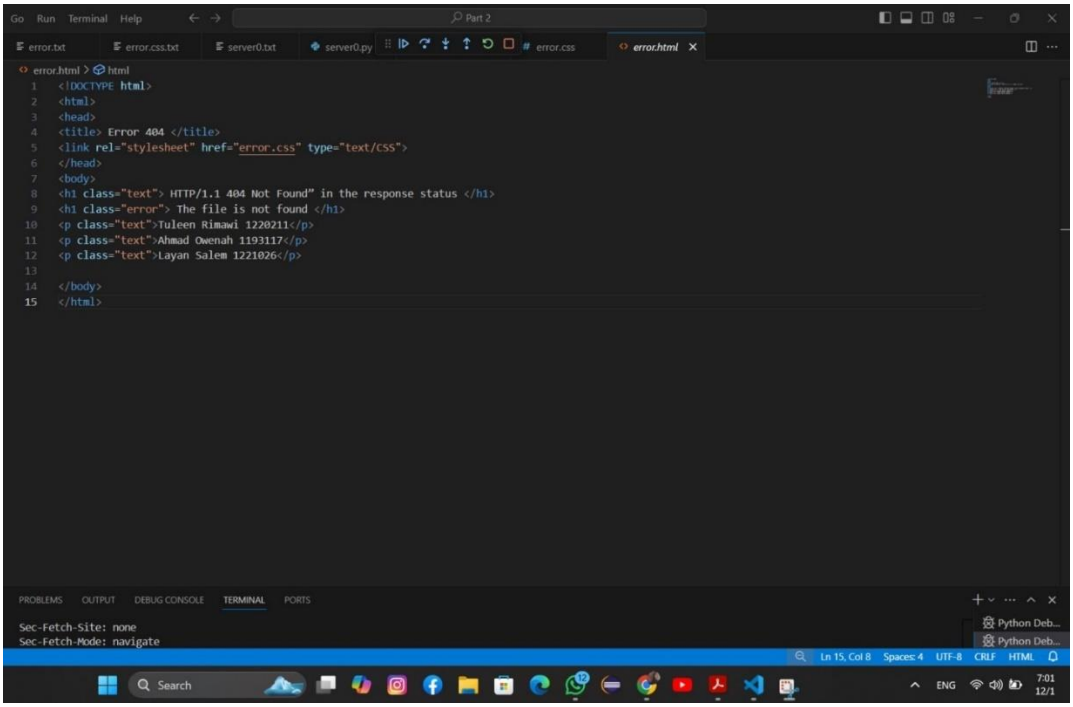


Figure 33 html code for wrong request

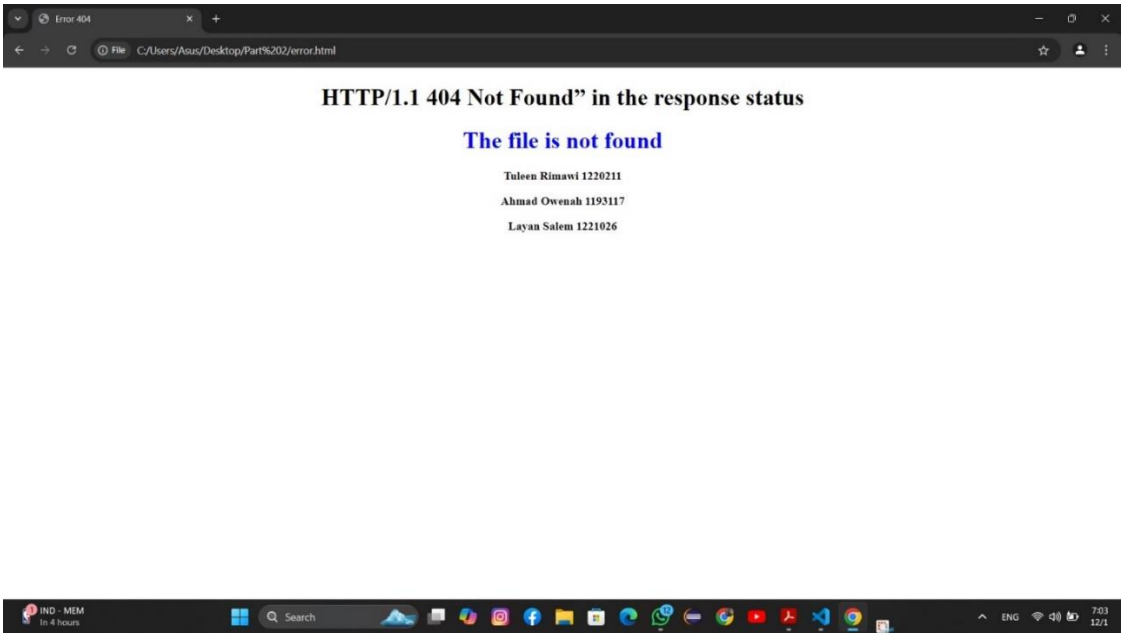


Figure 34 wrong request page

Task 3:

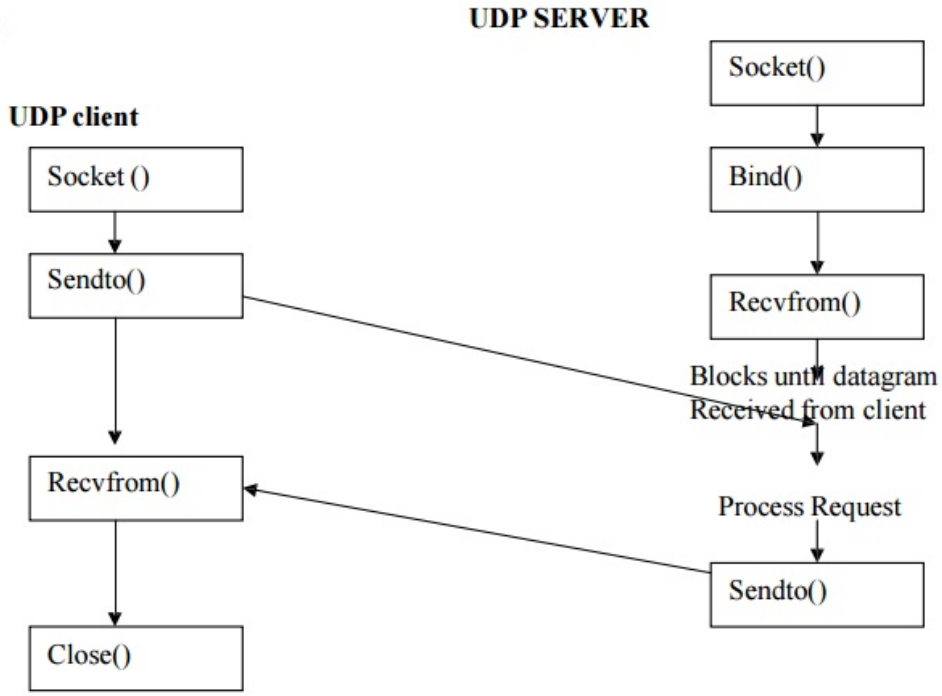


Figure 35 UDP datagram

This task is about creating an engaging multiplayer trivia game that uses UDP communication between a server and multiple clients. The goal is to implement a real-time game where players compete to answer trivia questions, with the server managing the game and scoring.

UDP (User Datagram Protocol) is fast and efficient, which makes it perfect for a trivia game where quick responses are more important than guaranteed delivery of every packet.

Unlike TCP, UDP doesn't waste time establishing connections or resending lost messages. This keeps the game flowing smoothly even if a few responses are missed.

How the game Works:

- The server is like the host of a quiz show. It sends out questions, collects answers, keeps track of scores, and announces winners.

- The clients are the players. They connect to the server, receive questions, and send their answers back, hoping to outsmart and outrun their opponents.

Procedure:

The server is the brain of the game:

1. Setting Up the Server

server.py × client.py dataset.txt

```
1   from socket import *
2   import time
3   import random
4
5   serverPort = 5689
6   serverSocket = socket(AF_INET, SOCK_DGRAM)
7   serverSocket.bind(('', serverPort))
8   serverSocket.settimeout(1)
9
10  print("The server is ready to receive on port", serverPort)
11
```

Figure 36 Setting Up the Server code

This is in the server code sets up a UDP server on port 5689 to listen for incoming datagrams. It sets a 1-second timeout for receiving data.

Table 1 methods used in setting server

Method	purpose
socket(AF_INET, SOCK_DGRAM)	Creates a UDP socket for communication between the server and clients. UDP is used for faster data transfer with minimal overhead.
bind(('', serverPort))	Binds the socket to port 5689 to allow the server to listen for incoming messages from clients.
settimeout(1)	Ensures the server doesn't wait indefinitely for client messages, allowing it to handle other tasks if no data is received within 1 second.

2. Loading the Question Dataset

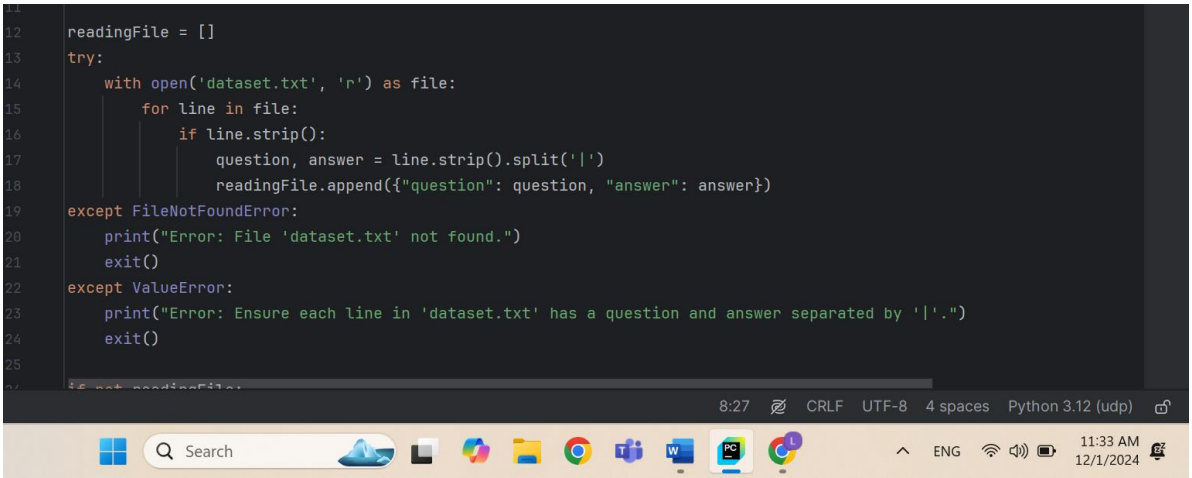


Figure 37 loading the Question Dataset code

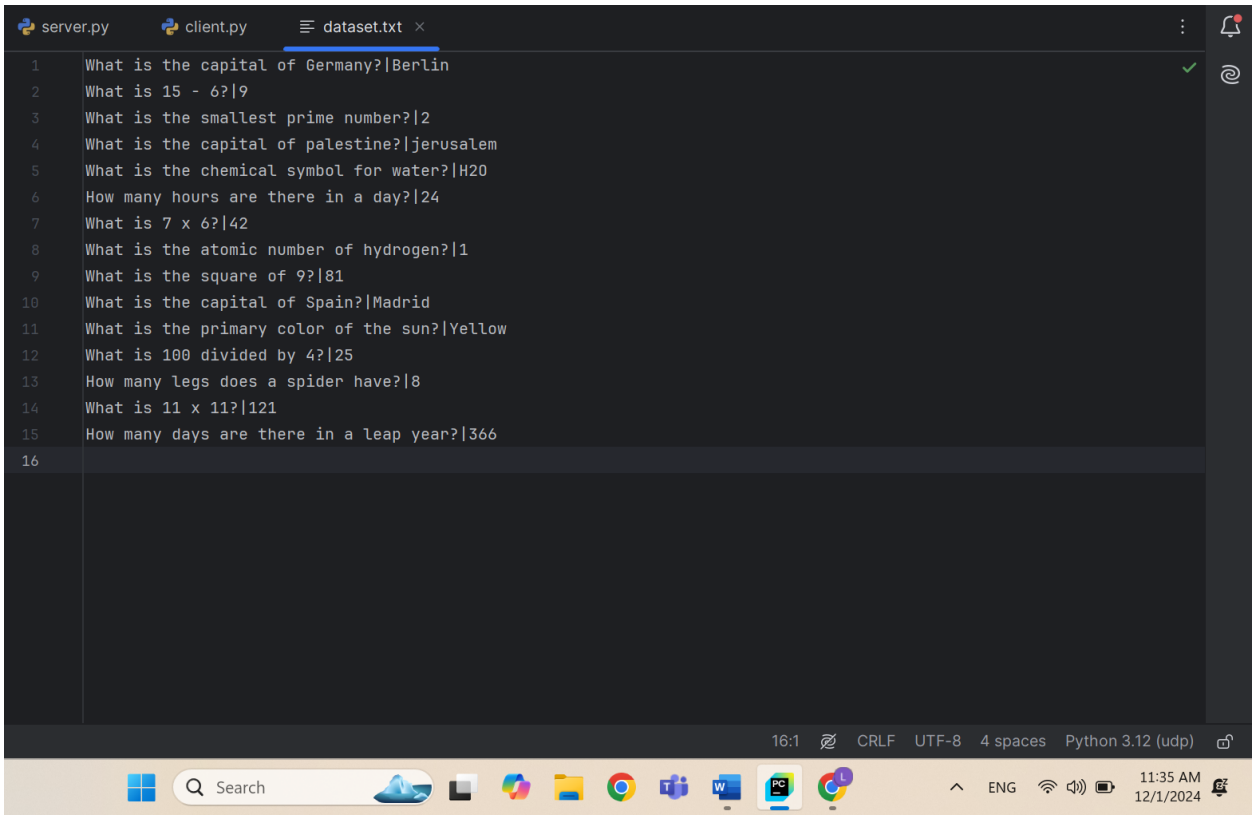


Figure 38 dataset file

Table 2 methods used in loading dataset

Method	Purpose
open('dataset.txt', 'r')	Opens the dataset file containing questions and answers for the game.
strip()	Removes unnecessary spaces or newlines to ensure clean data processing.
append({"question": question, "answer": answer})	Stores each question and answer in a list for random selection during the game.
FileNotFoundError	Handles the case where the dataset file is missing, ensuring the server exits gracefully.
ValueError	Ensures that the dataset is properly formatted (each line should have exactly one question and one answer separated by pipe " | "

3. Handling Client Connections

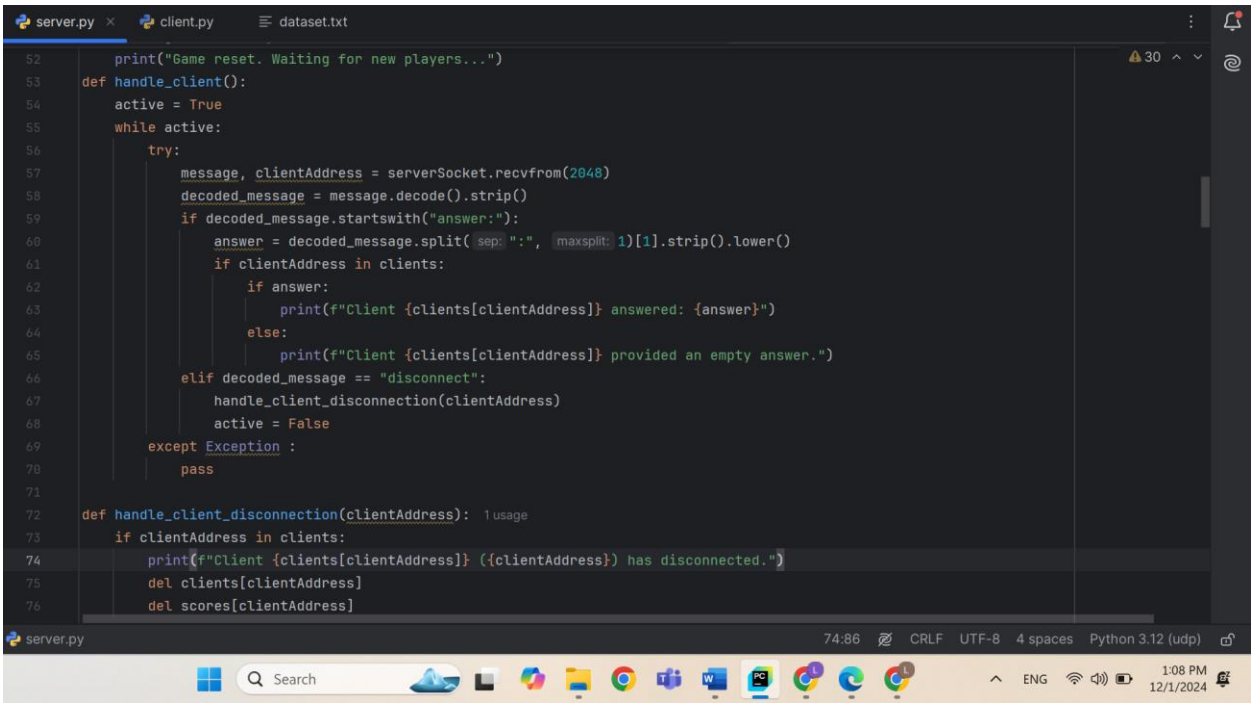


Figure 39 handling Client Connections code

Table 3 methods used in handling Client Connections code

Method	Purpose
recvfrom(2048)	Waits for client messages, such as username submissions or answers, and extracts the client's address.
startswith("username:")	Checks if a message contains a username, indicating that a new client is trying to join the game.
clients[clientAddress] = username	Adds the new player to the list of connected clients.
sendto(data.encode(), client)	Sends confirmation messages to clients, such as "You have joined the game!" or instructions for the game.

len(clients)	Counts the number of connected players to ensure that at least 2 players are present before starting the game.
del clients[clientAddress]	Removes disconnected players from the clients and scores dictionaries.
print()	Logs client connection events and disconnections for server monitoring.

4. Starting the Game

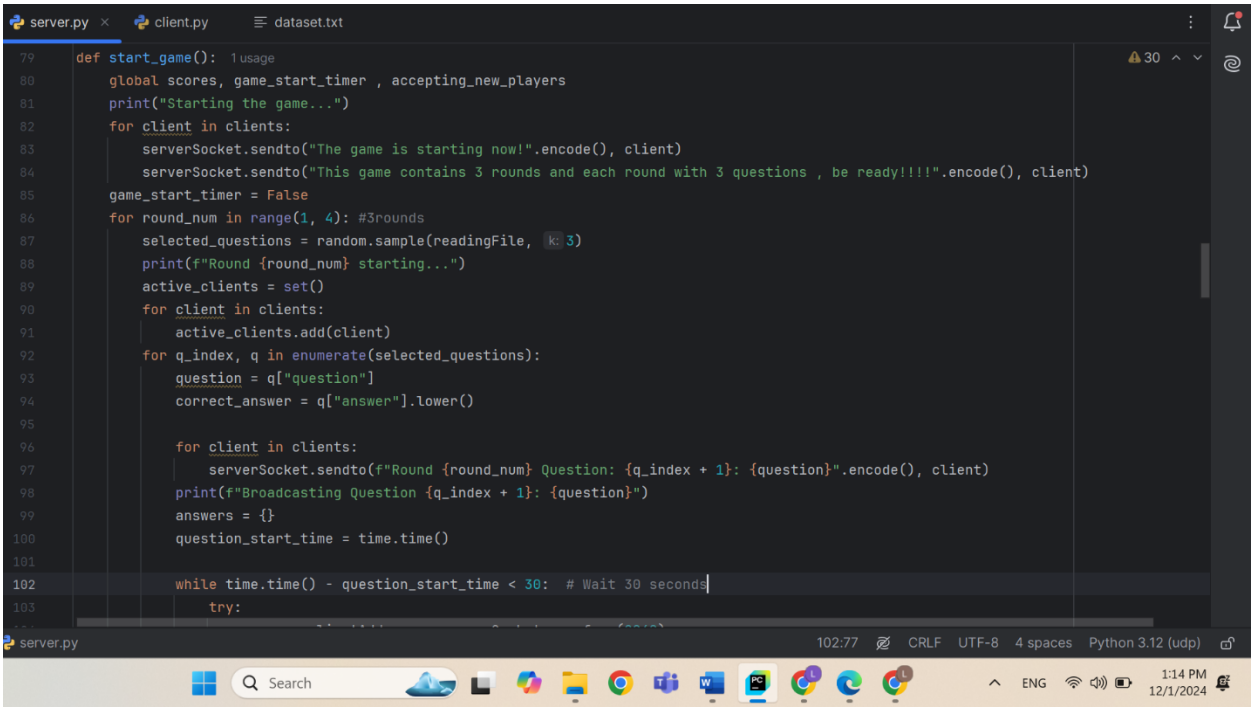


Figure 40 starting game code 1

Table 4 methods used in starting game1

Method	Purpose
time.time()	Tracks the countdown to start the game (30 second timer) and question-answer periods.

time.sleep(30)	Pauses for a set period (30 seconds) to give players time to prepare for the next round.
random.sample(readingFile, 3)	Randomly selects 3 questions for each round to keep the game engaging and unpredictable.

5. Processing Player Answers

```
def start_game():  1 usage
    while time.time() - question_start_time < 30:  # Wait 30 seconds
        try:
            answer, clientAddress = serverSocket.recvfrom(2048)
            if clientAddress in clients and clientAddress not in answers:
                decoded_answer = answer.decode().strip()
                if decoded_answer.startswith("answer:"):
                    answers[clientAddress] = decoded_answer[7:].strip().lower()
                    active_clients.add(clientAddress)
                    print(f"Received answer from {clientAddress}: {answers[clientAddress]}")
        except Exception:
            continue
    correct_clients = []
    for client, answer in answers.items():
        if answer == correct_answer:
            correct_clients.append(client)

    for idx, client in enumerate(correct_clients):
        points = 1 - (idx / len(correct_clients))
        scores[client] += points
        serverSocket.sendto(f"Correct! You earned {points:.2f} points.".encode(), client)
    for client in answers:
        if client not in correct_clients:
            serverSocket.sendto("Wrong!".encode(), client)
    correct_answer_message = f"The correct answer was: {correct_answer.capitalize()}"
```

Figure 41 starting game code 2

Table 5 methods used in starting game2

Method	Purpose
recvfrom(2048)	Collects answers from players during the question-answer period.
startswith("answer:")	Ensures the client message contains an answer for evaluation.
lower()	Converts both the player's answer and the correct answer to lowercase for case-insensitive comparison.
if answer == correct_answer	Checks if a player's answer matches the correct one.

scores[client] += points	Updates the player's score based on how quickly they answered correctly.

6. Sending Feedback and Round Results and Managing Game State

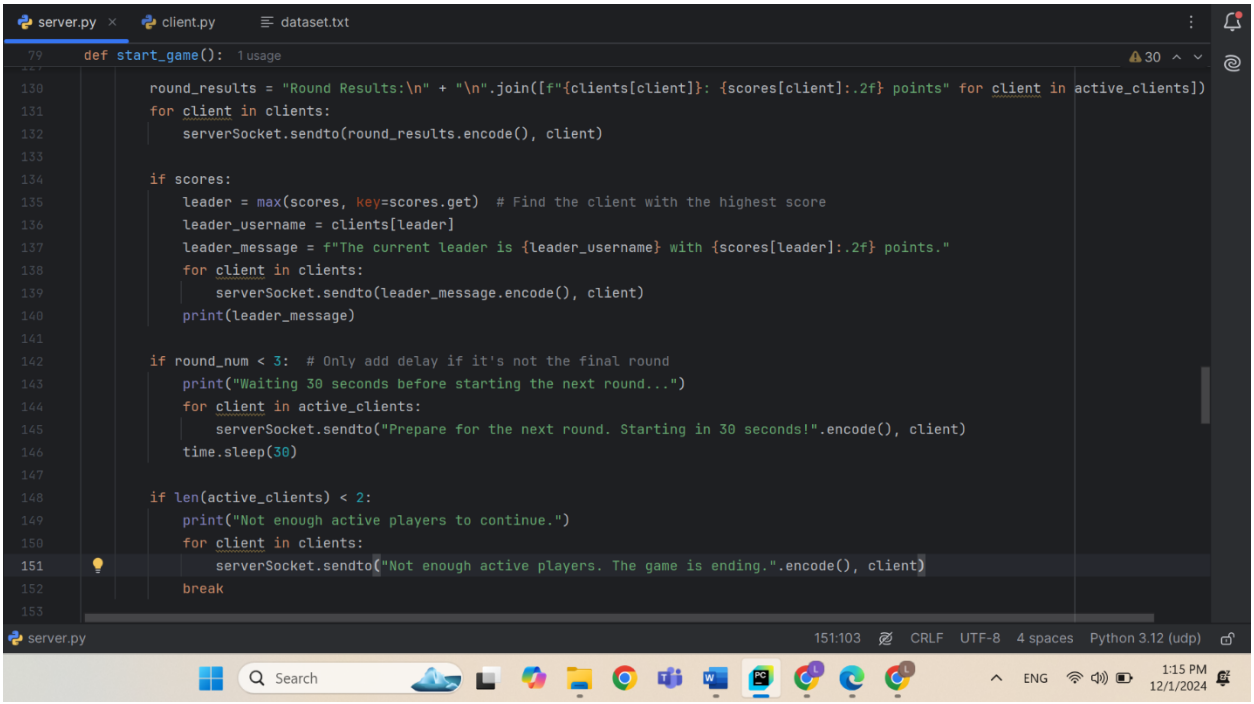


Figure 42 starting game code 3

Table 6 methods used in starting game 3

Method	Purpose
sendto(data.encode(), client)	Sends feedback to players, such as "Correct!", "Wrong!", and the correct answer for each question.
join()	Formats the list of player scores into a readable message for broadcasting results.
max(scores, key=scores.get)	Determines the player with the highest score to announce as the leader or winner.
reset_game()	Resets all game variables (e.g., scores, clients) for a new game.

global	Allows functions like reset_game to modify global variables (scores, clients, etc.).
if len(active_clients) < 2	Checks if there are enough players to continue the game. Ends the game early if fewer than 2 players are active.

7. Error Handling

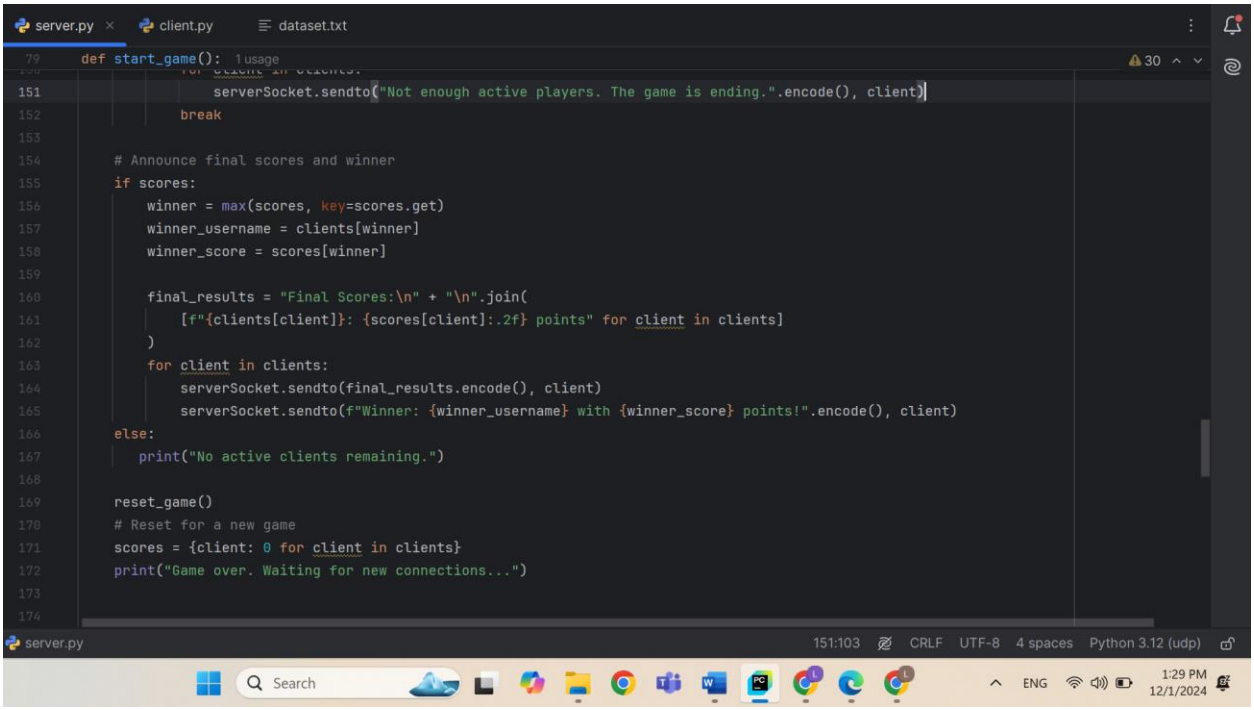


Figure 43 starting game code 4

Table 7 methods used in starting game 3

Method	Purpose
try and except	Prevents server crashes when errors occur (e.g., file not found, invalid client input, timeout while waiting for data).
exit()	Stops the server gracefully if critical errors occur (e.g., missing or invalid dataset).

8. Server Waiting for Players

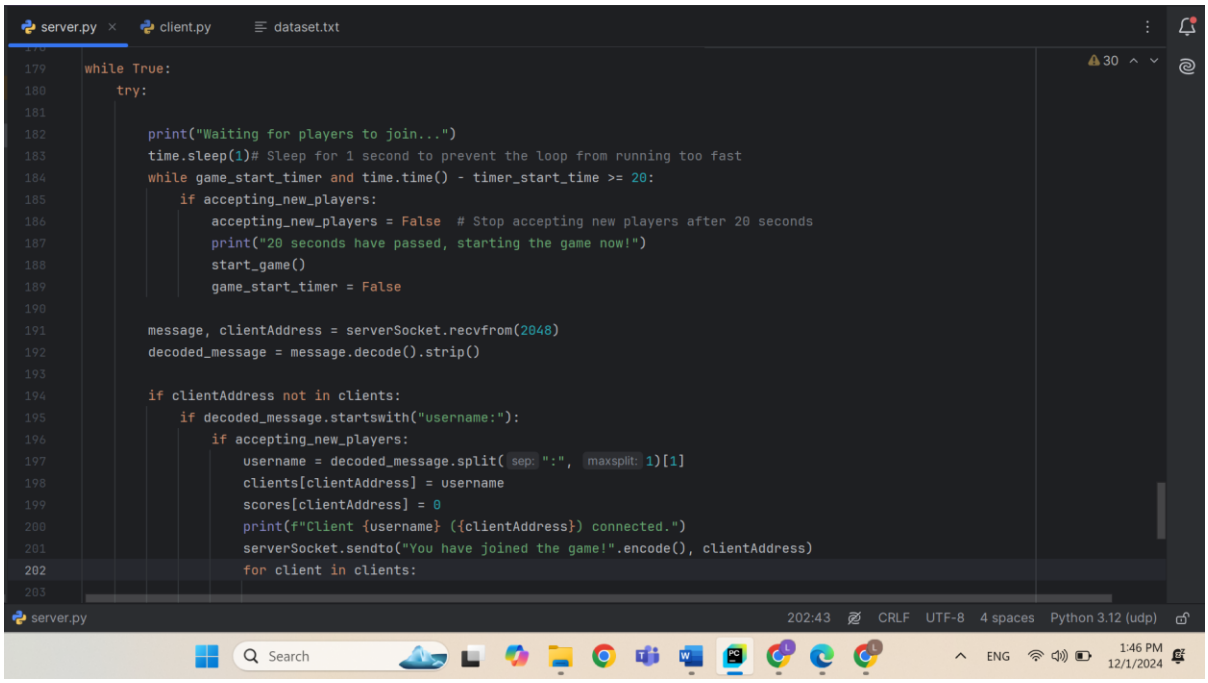


Figure 44 main server code

Table 8 methods used in main server code

Component/Method	Purpose
Main Loop (while True)	Keeps the server running, continuously waiting for players to join.
print("Waiting for players...")	Logs server activity for monitoring.
recvfrom(2048)	Waits to receive client messages and their addresses.
startswith("username:")	Checks if a client message contains a username (indicating a new player joining).
clients[clientAddress] = username	Adds the player's username and address to the active players list.
sendto("...".encode(), client)	Sends confirmation messages to players, such as "You have joined!" or "Game in progress, please wait."

9. Starting Game Countdown and Handling Errors and Late Joiners

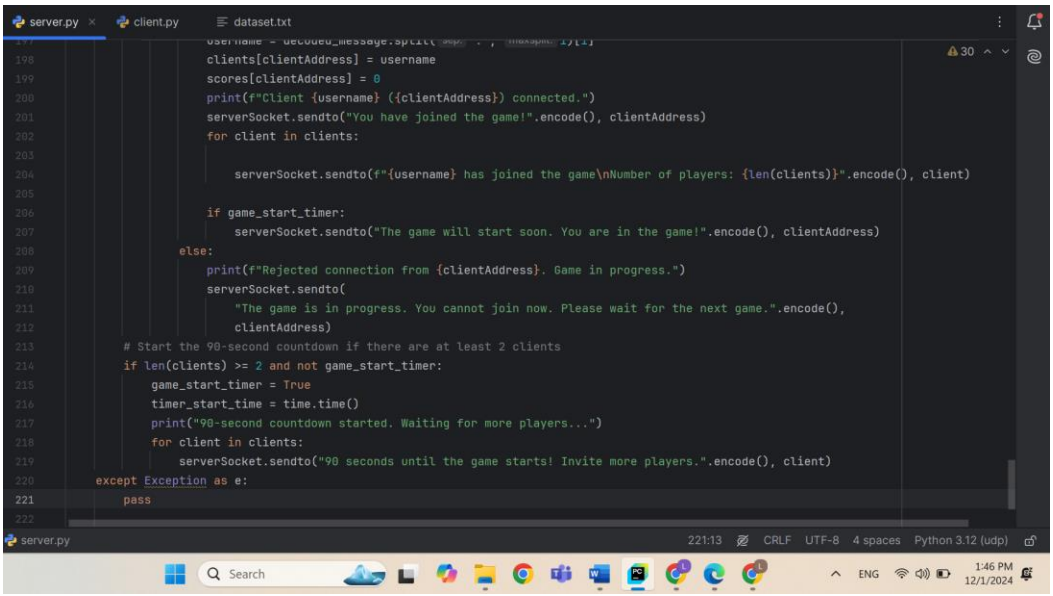


Figure 45 main server code 2

Table 9 methods used in main server code

Component/Method	Purpose
if len(clients) >= 2:	Begins the game countdown only when there are at least 2 players.
game_start_timer = True	Activates the countdown timer for game initiation.
timer_start_time = time.time()	Tracks when the countdown began to limit player entry after 20 seconds.
sendto("90 seconds until the game starts...".encode(), client)	Notifies all players about the countdown and encourages them to invite more players.
accepting_new_players = False	Stops accepting new players when the countdown ends, locking the player list.
start_game()	Initiates the actual gameplay process after the countdown ends.
if clientAddress not in clients:	Ensures new players can only join if the game hasn't started.

sendto("Game in progress...".encode(), clientAddress)	Notifies late joiners that they cannot participate and must wait for the next game.
except Exception as e:	Catches and suppresses errors (e.g., invalid messages, timeout) to ensure the server remains stable.
pass	Allows the loop to continue uninterrupted even if an error occurs.

Player is the clients who joining the game and connect to the server.

The client code allows a user to connect to a game server via UDP, register their username, and participate in a timed question-answer game. It listens for server messages, submits answers, and handles disconnection when the game ends or if there are not enough active players. The client operates in a loop, ensuring smooth communication with the server throughout the game.

1- Setup and Username Registration

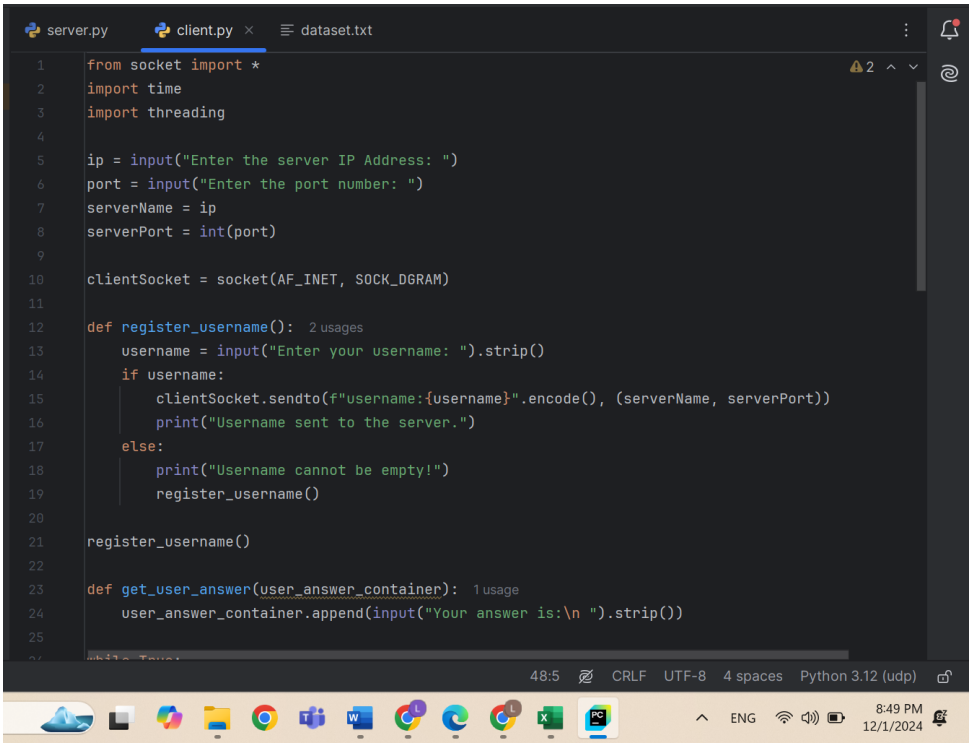


Figure 46 client code 1

Table 10 methods used in client code

Component/Method	Purpose
socket(AF_INET, SOCK_DGRAM)	Creates a UDP client socket to communicate with the server.
input("Enter the server IP Address...")	takes the server's IP and port number as input to connect the client to the game
register_username()	Prompts the user to input a username and sends it to the server in the format username:<name>. Ensures the username is not empty.
sendto(f"username:{username}".encode(), (serverName, serverPort))	Sends the username to the server, enabling the user to join the game.

2- Listening to Server Messages and Answer Submission

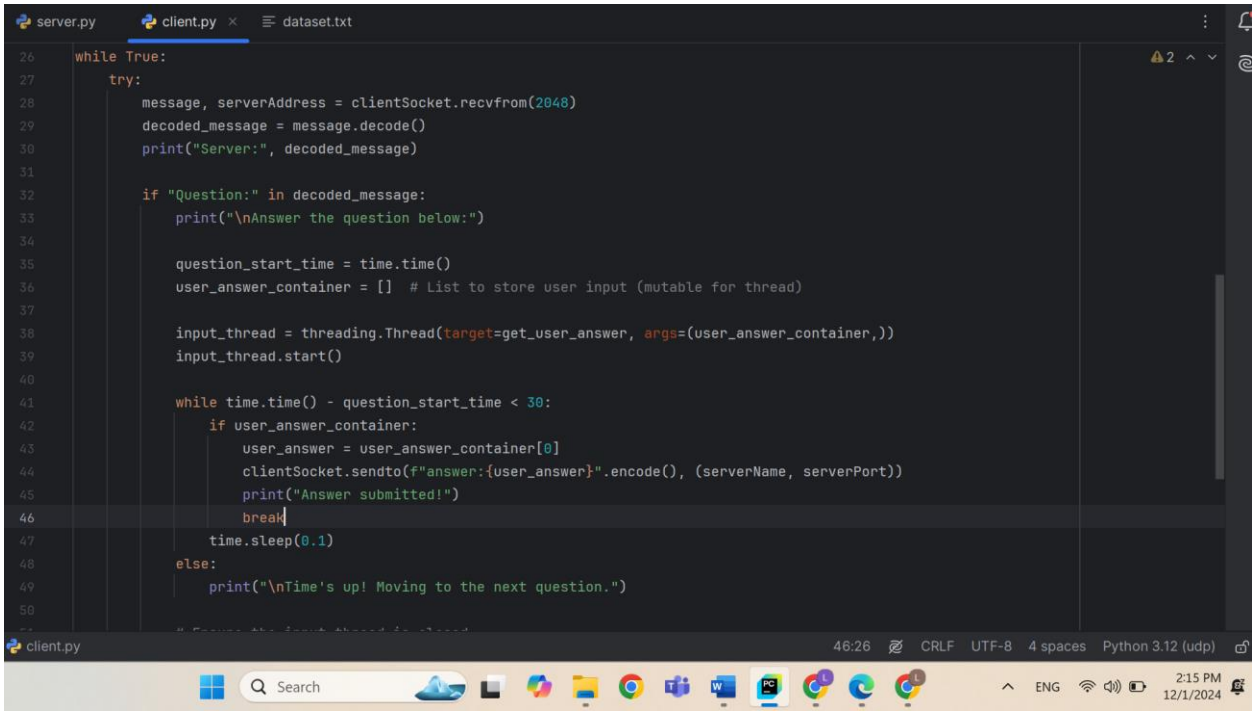


Figure 47 client code 2

Table 11 methods used in client code 2

Method	Purpose
recvfrom(2048)	Listens for messages from the server (e.g., questions, instructions, or results)
decode()	Decodes the received bytes into a readable string for display to the user.
if "Question:" in decoded_message:	Checks if the server has sent a new question, triggering the question-answer phase.
print("Server:", decoded_message)	Displays messages from the server, such as game updates or questions, to the user.
threading.Thread(target=get_user_answer...)	Starts a thread to allow the user to input their answer while a timer runs concurrently.
time.time() - question_start_time < 30	Sets a 30-second timer for answering the question. If the user doesn't respond in time, it moves to the next question.
clientSocket.sendto(f"answer:{user_answer}".encode(), (serverName, serverPort))	Sends the user's answer to the server.
input_thread.join(timeout=0)	Ensures the input thread terminates after the answer is submitted or the timer expires.

3- Error Handling

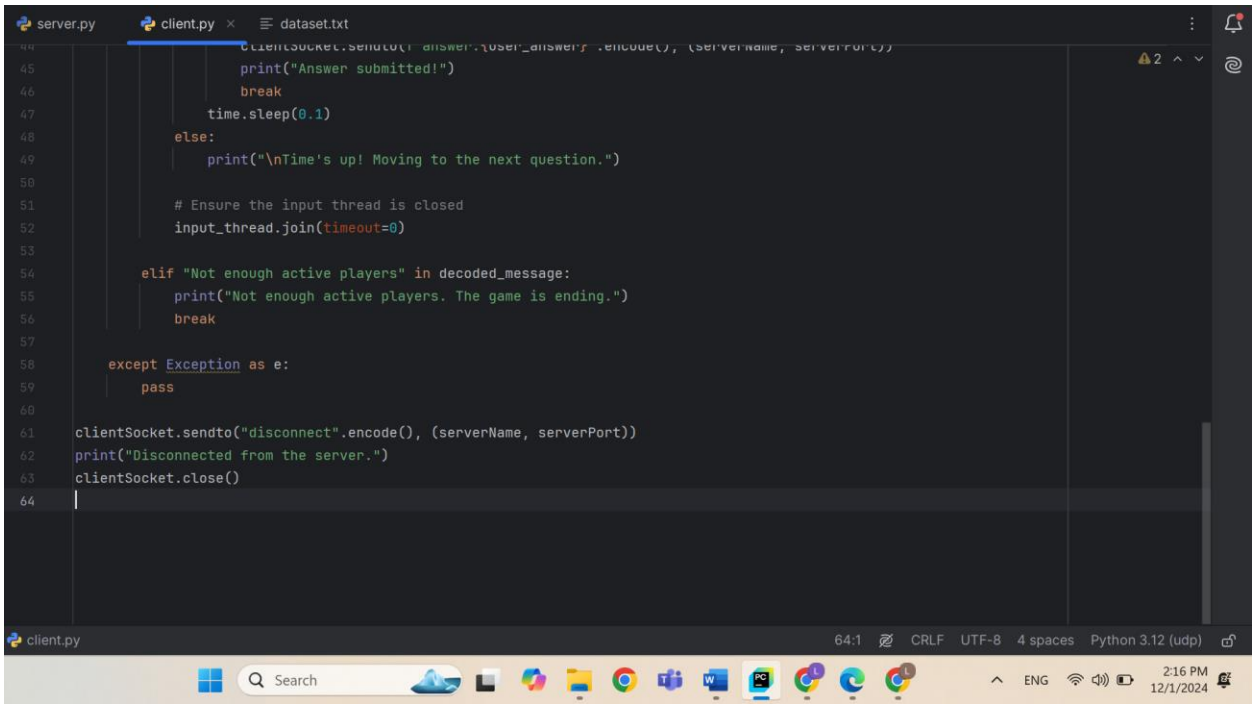


Figure 48 client code 3

Table 12 methods used in client code3

Method	Purpose
if "Not enough active players" in decoded_message:	Terminates the game if the server ends it due to insufficient players.
sendto("disconnect".encode(), (serverName, serverPort))	Notifies the server when the client disconnects.
except Exception as e:	Handles errors, such as network issues or invalid server responses, ensuring the client does not crash unexpectedly.
clientSocket.close()	Closes the socket connection after the client disconnects, freeing up resources.

Results and Discussions

Task one:

Explanation

- ipconfig - this is used to show the network configuration of a computer, including its IP address, subnet mask, and default gateway that are crucial for communication over networks.

- ping: Sends small packets of data to a specific IP address to determine if a device is reachable over the network, while it measures how long it takes for packets to make the round trip.
- tracert: Packet sent to a destination IP, route followed traced showing the IP of every router packets pass through and time taken to reach each hop.

- Nslookup: Queries a DNS server to translate a domain name into its corresponding IP address or to resolve an IP address back to its domain name.

- Telnet: A protocol that enables a user to establish a command-line interface session with a remote server or device, interactively getting connected over a network.

Trying the commands

1. Run the ipconfig /all

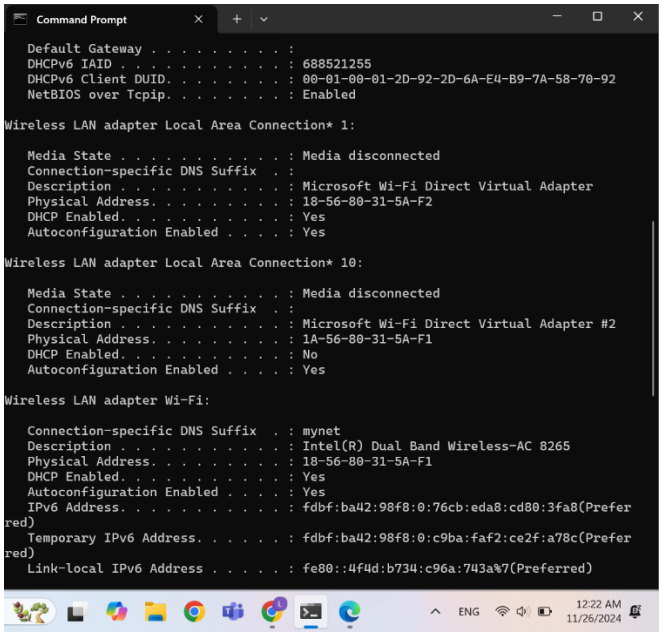


Figure 49 running ipconfig

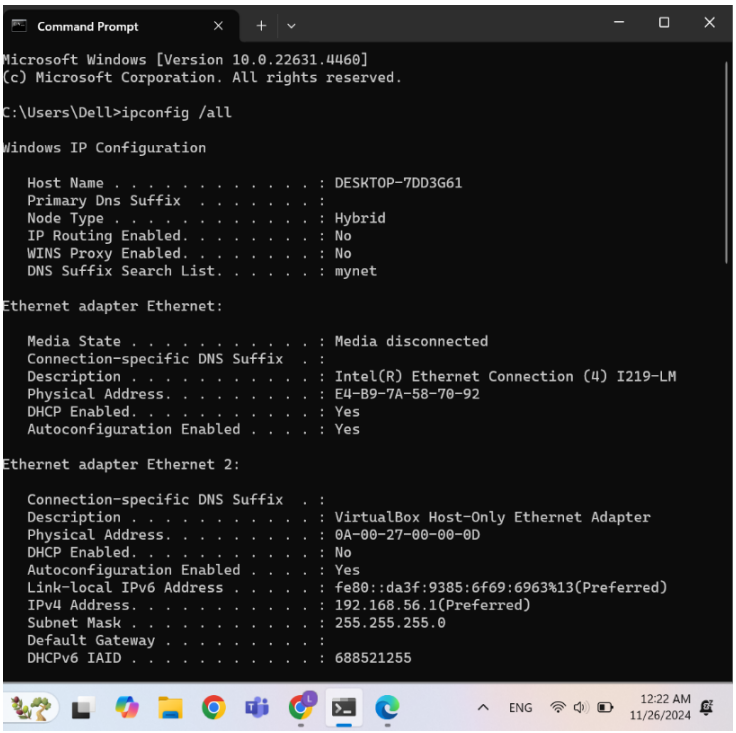


Figure 50 running ipconfig 2

Figure 51 running ipconfig 3

This output shows the network configuration of the computer, indicating details like:

- IP or (IPv4) Address: 192.168.1.12
- subnet mask: 255.255.255.0
- default gateway: 192.168.1.1(The address of the router, which connects the local network to the internet.)
- DNS server addresses: 192.168.1.1 (Indicates that the router is also serving as the DNS server for name resolution).

2. Ping

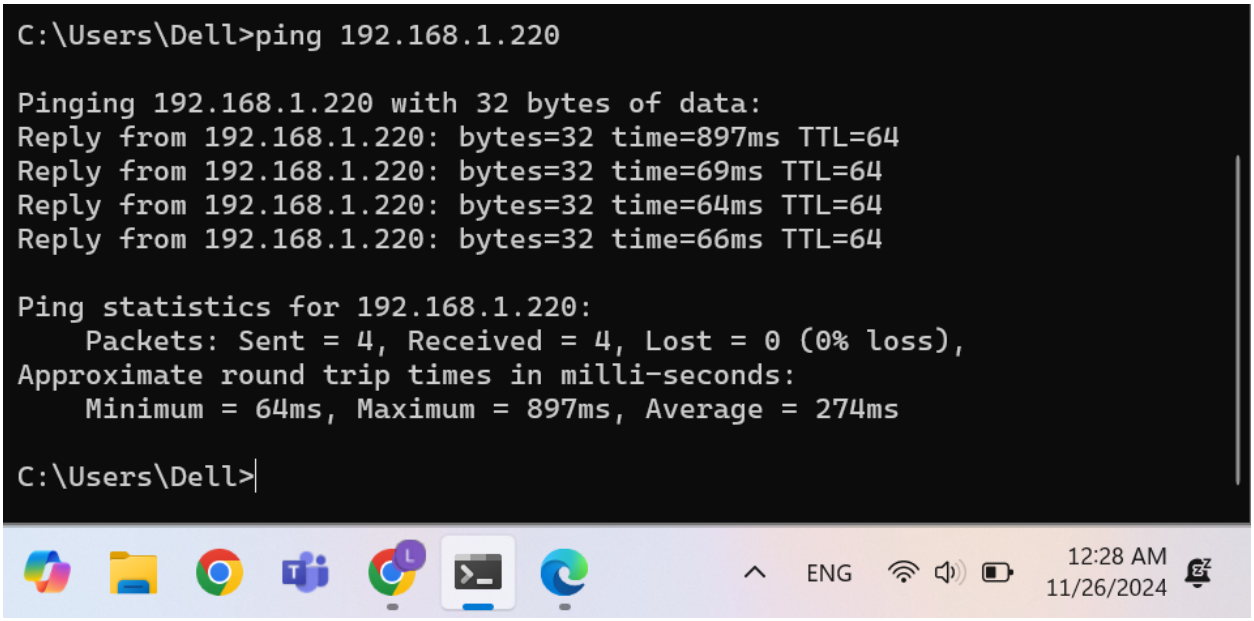


Figure 52 testing ping1

To ping a device in the same network, IP address is needed, the device in Fig -----is my smart phone, the IP address:192.168.1.220 was taken from the settings and run the ping command to it. Ping showed path of the packets sent to that device and the times taken.

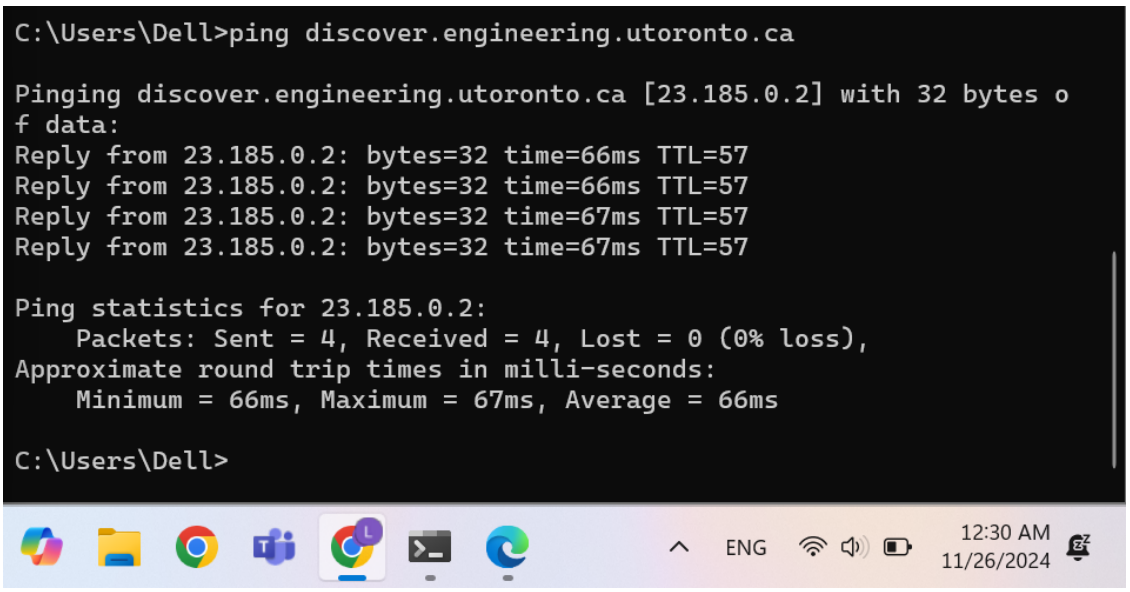
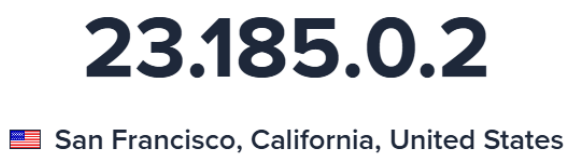


Figure 53 testing ping 2

The ping results show a response time of around 65 ms, which suggests the server is not very far. However, the IP address belongs to Fastly, a Content Delivery Network (CDN).



This means the response might be coming from a server in USA used to cache content, rather than the original server in Canada.

3. Tracert

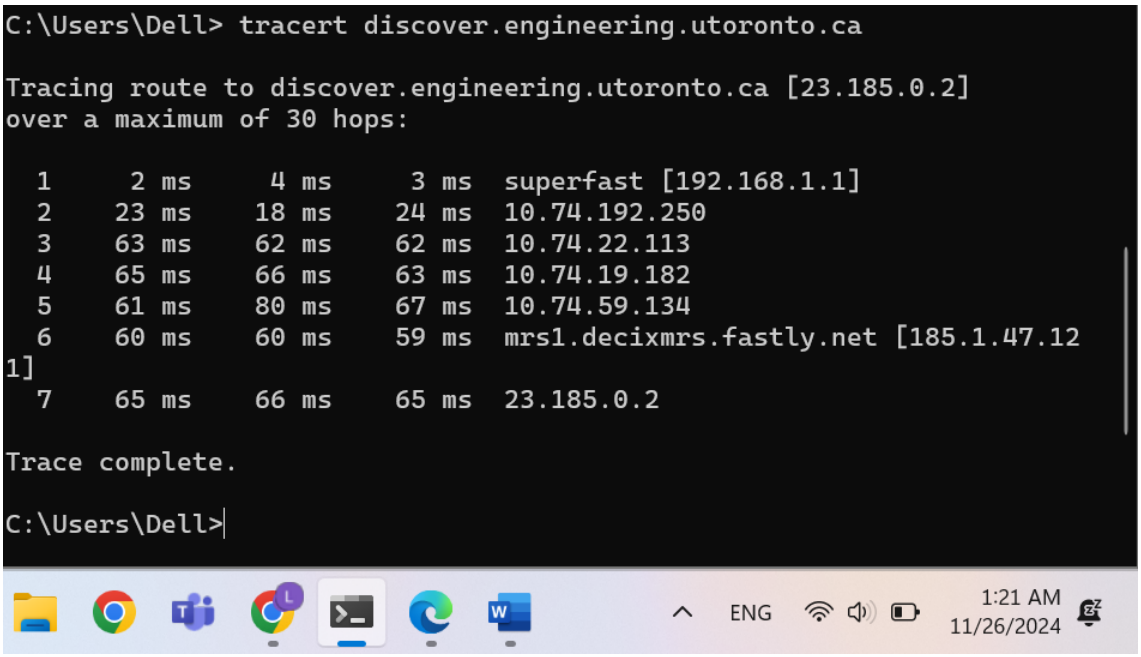


Figure 54 testing tracert

The results of the tracert the path taken by packets to reach the server discover.engineering.utoronto.ca, listing all intermediate network devices (hops) and their IP addresses. It can be noticed that the final IP is 10.74.59.134 , which is the IP for the website 23.185.0.2, as it was shown in the previous Figure (when run ping).

4. nslookup

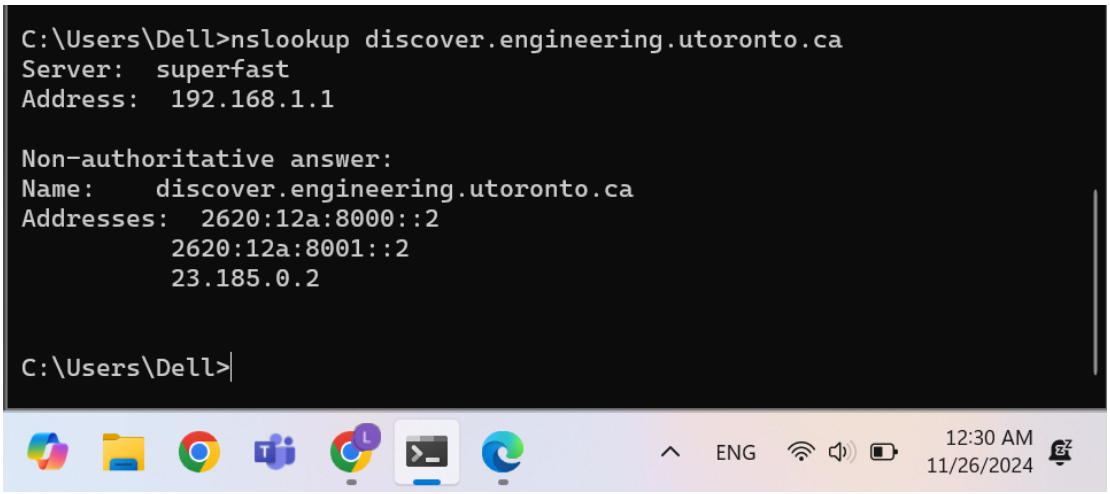


Figure 55 testing nslookup

DNS lookup for discover.engineering.utoronto.ca, showing the resolved IP address and the DNS server used to perform the query.

5. Telent

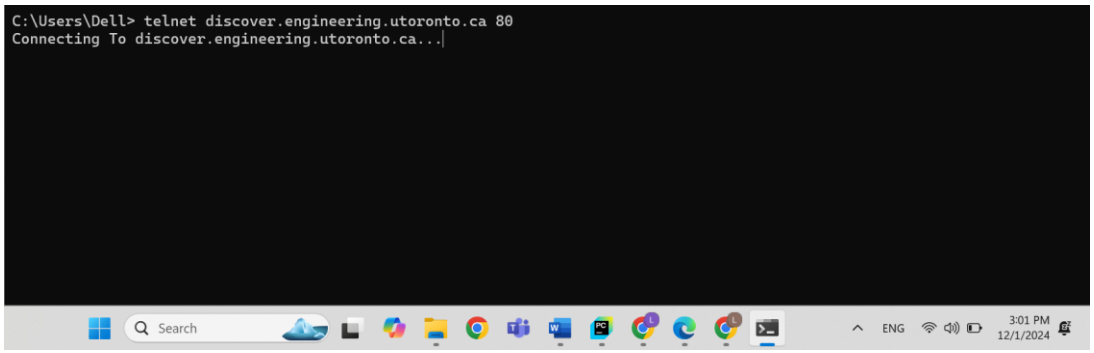


Figure 56 testing telnet

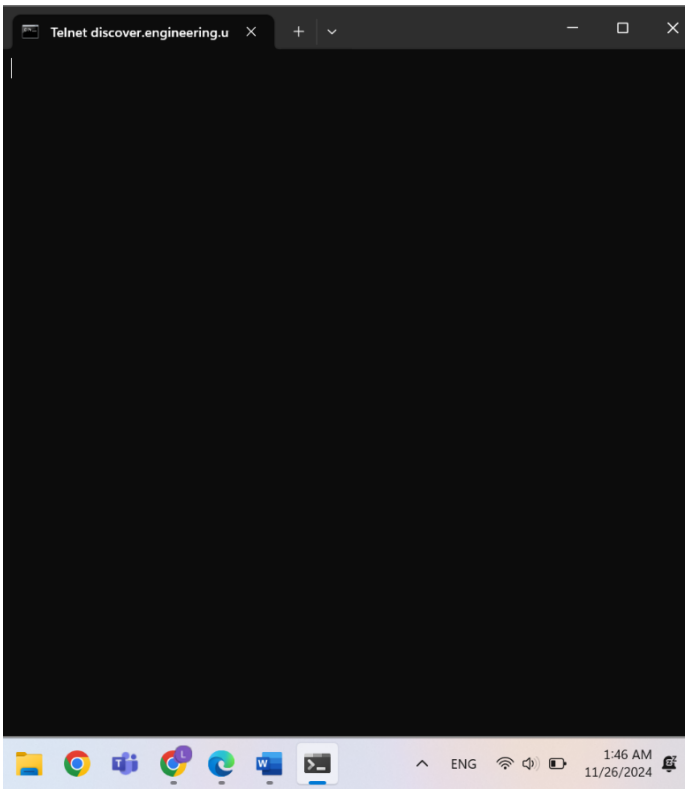


Figure 57 testing telnet 2

The connection for telnet is running, the server is waiting for a message.

Wireshark

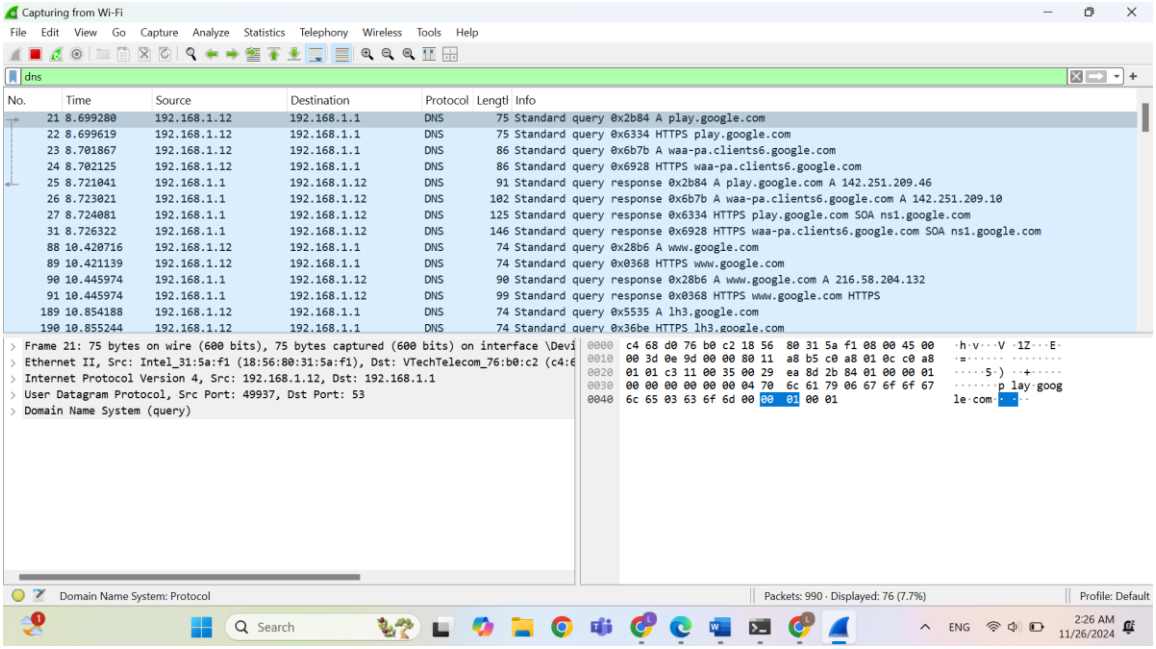


Figure 58 Wireshark test

Here is the output of the Wireshark software when capturing some DNS messages.

Task two:

The testing process for Task 2 is detailed in the procedure section. It explains the steps taken to verify the functionality, ensuring that the task met the required objectives and operated as expected.

Task three:

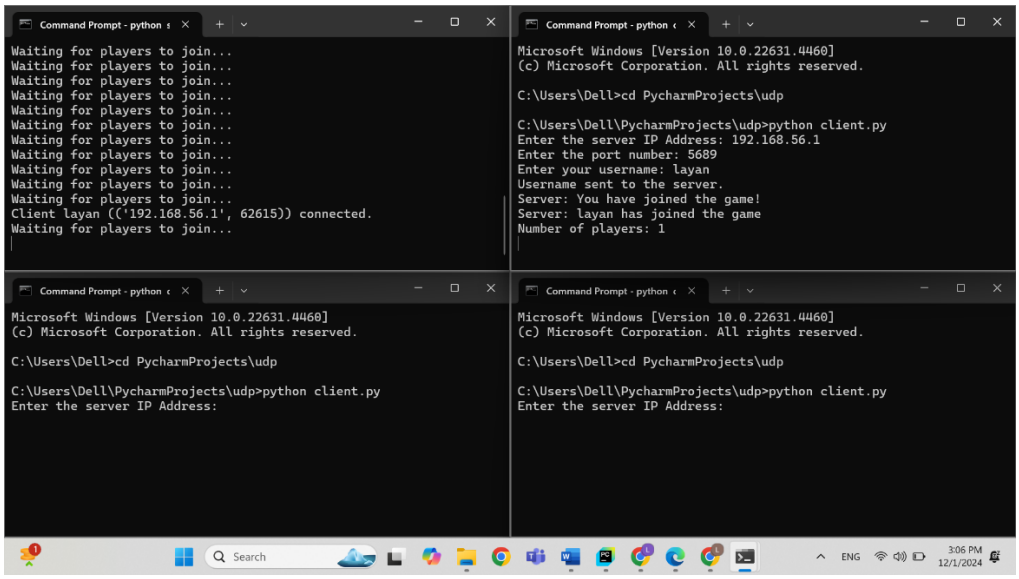


Figure 59 testing task 3 part 1

Figure 59:

Server keep printing until game started, it starts when at least 2 clients joined.

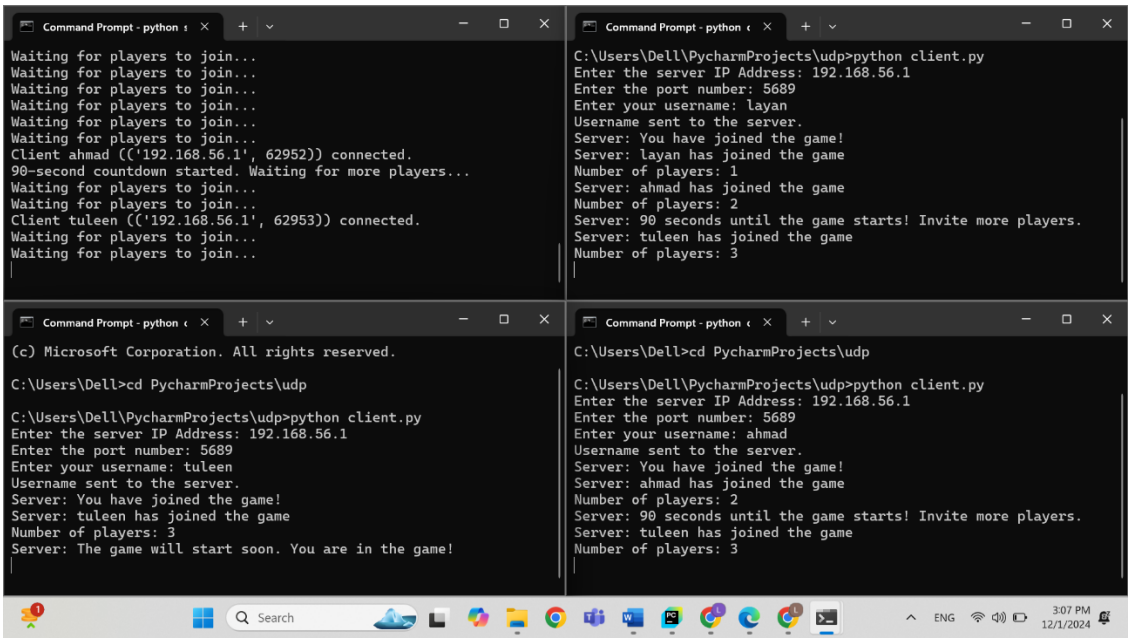


Figure 60 testing task 3 part 2

Figure 60:

Client Side: the client enters the IP address and port number that the server is running on. After that, the username is sent to the server.

Server Side: After two players connect, the server waits for 90 seconds before starting the game, allowing time for additional players to join. The server sends updates to the players regarding the number of active players.

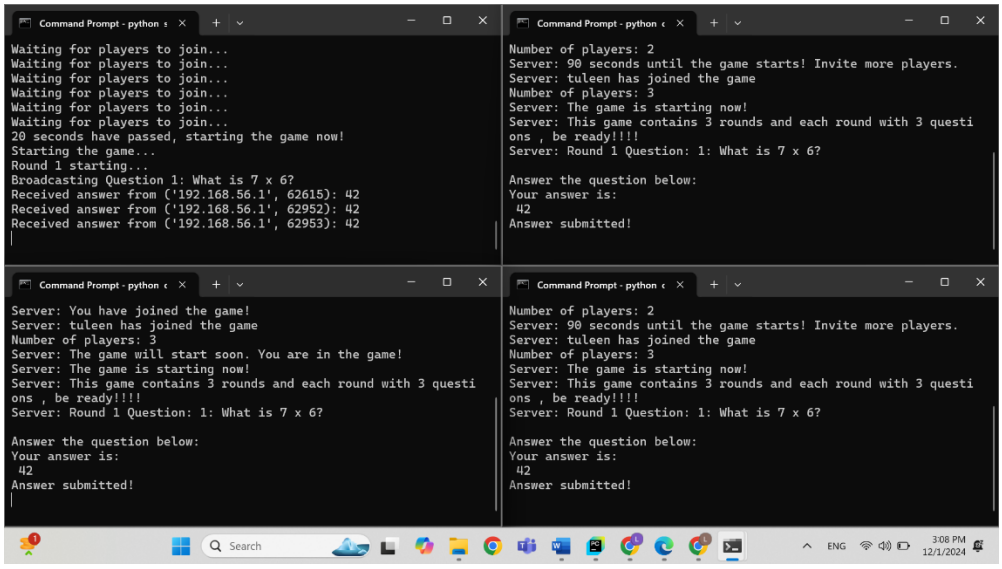


Figure 61 testing task 3 part 3

Figure 61:

After the time has passed, the server starts sending questions, and the players submit their answers.

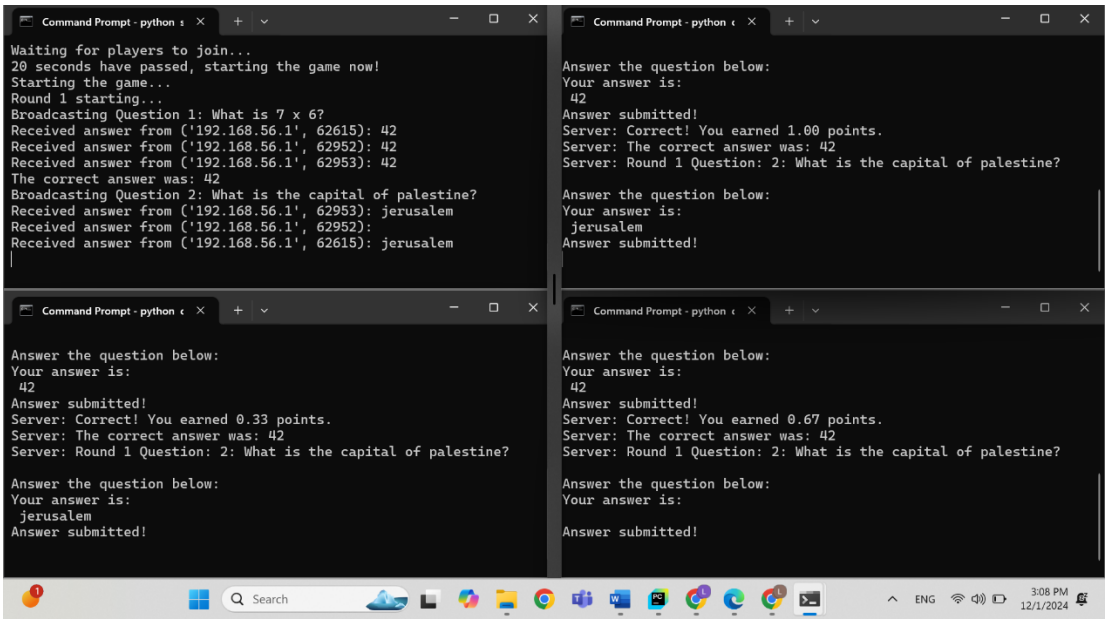


Figure 62 testing task 3 part 4

Figure 62:

Now, the server awards points as follows: 0 points for wrong answers, and for correct answers, points are assigned in order (1 - index/number of correct players). For example, Layan gets 1 point, Ahmad gets 0.66, and Tuleen gets 0.33. After the time for answering passes, the next question in the round is sent.

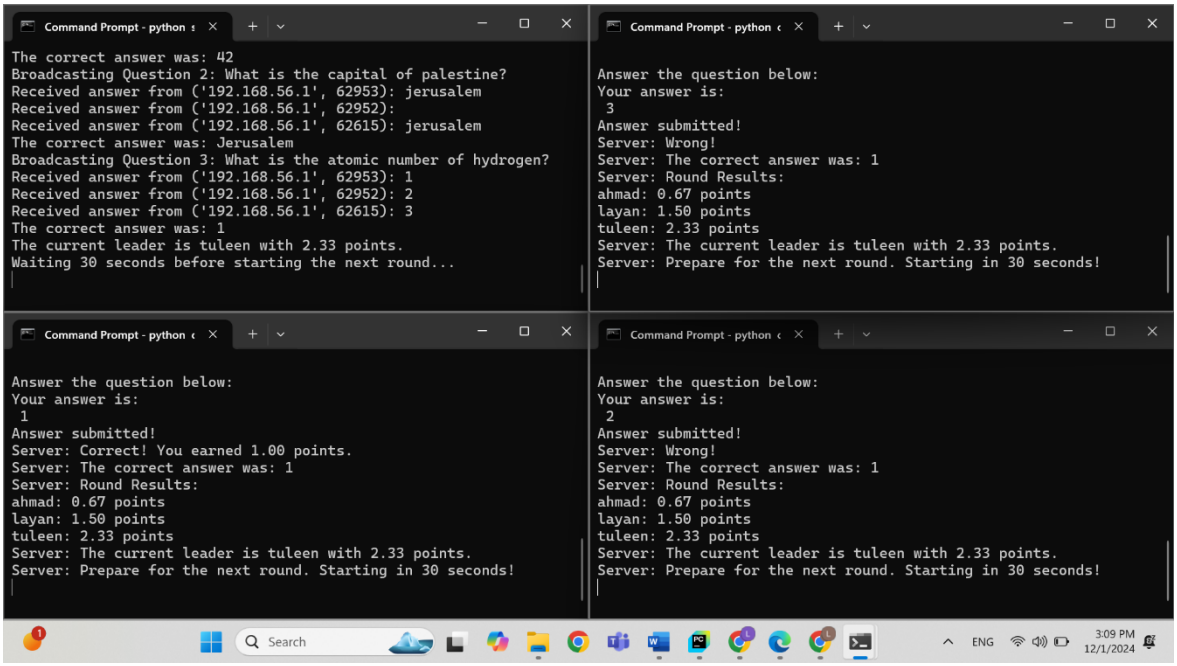


Figure 63 testing task 3 part 4

Figure 63:

After every round, the server sends the results to all players and announces the leader. At the end of the game, the server sends the winner's details to the players.

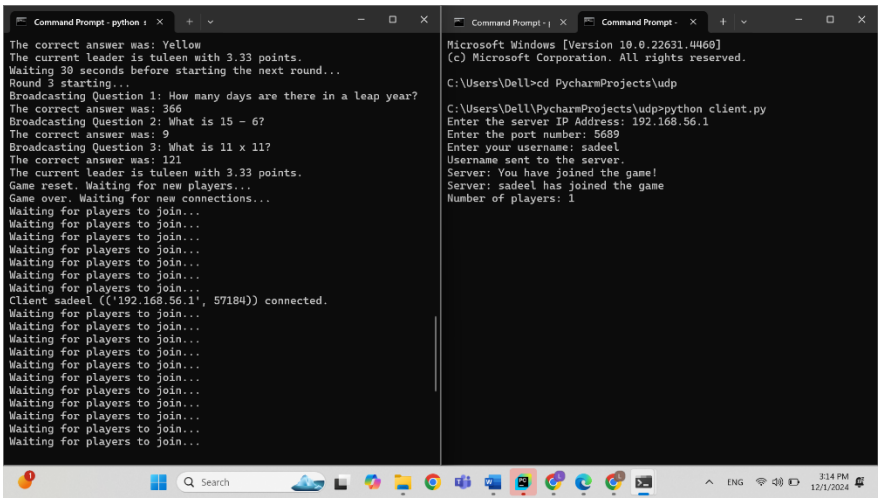


Figure 64 testing task 3 part 5

Figure 64:

After the first game ends, the server continues to run and waits for more players to join, then repeats the game process.

Alternative Solutions, Issues, and Limitations

Task 2:

Web Server:

Working on the project without using any library other than the socket library was quite challenging, especially when it came to handling image searches. Normally, parsing a URL would be easy with a library but since we couldn't use it, we had to implement the functionality manually, which was tricky and time-consuming.

Another challenge we faced was figuring out how to automatically redirect to another webpage. It took a lot of research and effort to come up with a working solution.

Task 3:

Issues Faced and Solutions

1. Blocked Server Operations
    - Issue: The server's message reception was blocked until the message was fully received, preventing the server from continuing its operations. This caused delays and incomplete communication.
    - Solution: We set a timeout for message reception and used a try-except block to handle exceptions. This allowed the server to continue its operations even if a message wasn't received within the specified time, ensuring smoother communication and preventing the server from getting stuck.
2. Client Program Stuck on Answer
    - Issue: If the client didn't answer the question, the program would get stuck, and the next question wouldn't be sent.
    - Solution: We used threading to allow the client to simultaneously handle user input and listen for server messages. This ensured that even if the user didn't provide an answer, the program could continue to receive the next question without delays.

3. Handling Unexpected Exceptions
    - Issue: Using except timeout alone caused other unexpected exceptions to appear, interrupting the program.
    - Solution: We replaced it with a generic except Exception block, which catches all types of exceptions. This ensured the program remained stable and continued running, regardless of the specific error encountered.

Limitations

In the server code, we didn't use threading, and it worked fine during testing because of the small number of players. However, if a large number of players join, the server could face performance issues. Specifically, the message handling could become sequential, causing delays in processing as the server handles one client at a time. This may lead to slower responses, increased latency, or even unresponsiveness under heavy load. Using threading or asynchronous handling would be necessary to handle a large number of players efficiently.

Teamwork Chart analysis:

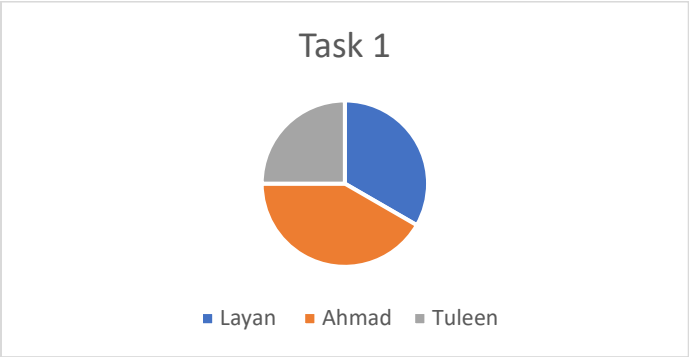


Figure 65 Teamwork Chart analysis

Task One

- In this task, we decided it would be better if each team member tried the commands individually. Ahmad then took responsibility for testing the functionality.

Figure 66 Teamwork Chart analysis

Task Two

- This task required more effort compared to the first one. We divided the work: Tuleen started by designing the pages, and Layan resolved any problems in code encountered during the process.

Figure 67 Teamwork Chart analysis

Task Three

- For this task, Layan wrote the initial code, and Ahmad focused on fixing any issues.

References

[1]: Simplilearn. "Understanding Networking Commands." Retrieved from https://www.simplilearn.com.

[2]: N4L Support. "How to Use Telnet to Check the Status of Ports." Retrieved from https://support.n4l.co.nz.

[3]: Wireshark Documentation. "Introduction to Wireshark." Retrieved from https://www.wireshark.org.

[4]: GeeksforGeeks. "Socket Programming in Python." Retrieved from https://www.geeksforgeeks.org.

[5]: Avast. "TCP vs. UDP: Key Differences." Retrieved from https://www.avast.com.