Faculty of Engineering & Technology

Electrical & Computer Engineering Department

ENCS3320: Computer Networks

Second Semester (2024/2025)

Project 1: Socket Programming

Prepared by:

Taleed Hamadneh	1220006	sec.4
Qasim Batrawi	1220204	sec.2

Instructors:

Dr. Ibrahim Nemer

Dr. Alhareth Zyoud

Date: May 10, 2025

Abstract

This report explains a project created to help us learn more about computer networks and how to use socket programming. The project is divided into three parts. First, we explored and practiced common network commands and used Wireshark to understand how data moves across the internet. Next, we built a basic web server using sockets that can show web pages in both English and Arabic, manage redirections, and handle missing files or wrong requests. Finally, we created a multiplayer guessing game where players connect to a server and try to guess a secret number. The game uses TCP to set up the connection and UDP for fast communication during the game. Through this project, we gained practical experience in networking, programming, and working as a team.

Table of Contents

Table of Figures

Theory & Procedure

Socket Programming

A typical network application consists of a pair of programs—a client program and a server program—residing in two different end systems. When these two programs are executed, a client process and a server process are created, and these processes communicate with each other by reading from, and writing to, sockets. When creating a network application, the developer's main task is therefore to write the code for both the client and server programs [1].

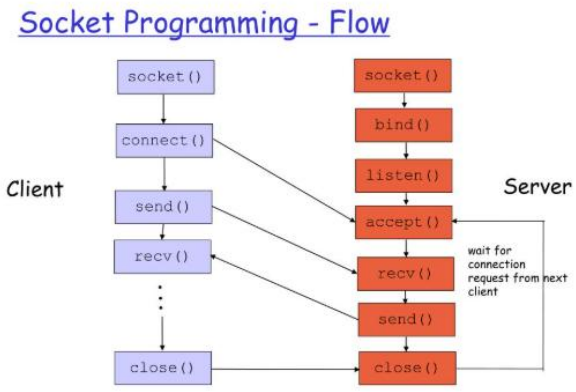


Figure 1 Socket Programming [1]

Socket Programming with UDP

The User Datagram Protocol (UDP) runs on top of the Internet Protocol (IP) and was developed for applications that do not require reliability, acknowledgment, or flow control features at the transport layer. This simple protocol provides transport layer addressing in the form of UDP ports and an optional checksum capability. UDP is a very simple protocol. Messages, so called datagrams, are sent to other hosts on an IP network without the need to set up special transmission channels or data paths beforehand. The UDP socket only needs to be opened for

communication. It listens for incoming messages and sends outgoing messages on request [2].

Socket Programming with TCP

Unlike UDP, TCP is a connection-oriented protocol. This means that before the client and server can start to send data to each other, they first need to handshake and establish a TCP connection. One end of the TCP connection is attached to the client socket and the other end is attached to a server socket. When creating the TCP connection, we associate with it the client socket address (IP address and port number) and the server socket address (IP address and port number). With the TCP connection established, when one side wants to send data to the other side, it just drops the data into the TCP connection via its socket. This is different from UDP, for which the server must attach a destination address to the packet before dropping it into the socket [1].

Figure 2 UDP VS TCP [1]

Network Commands

Network commands are essential tools that help users interact with and control network settings. They are used to troubleshoot problems, check connectivity, gather details about network devices, and monitor how data moves across a network. With these commands, users can check IP settings, trace the path of data packets, translate domain names to IP addresses, and manage different network protocols. These tools are particularly useful for network administrators and others working to improve or repair network performance. Below are the specific commands we will use in this project:

I. **ipconfig**

   Shows your computer's IP address and network details. It's useful for checking your connection or troubleshooting network issues.

II. **ping**

   Sends a small message to another computer or website to see if it's reachable and how long it takes to respond. It's great for checking if your internet or a specific site is working.

III. **tracert/traceroute**

   Shows the path your data takes to get to another computer or website, step by step. It helps to find where delays or problems happen along the way.

IV. **telnet**

   Let’s you connect to another computer over the internet and control it using text commands. It's mostly used for testing or remote access to servers, but isn't very secured.

V. **nslookup**

   Finds the IP address of a website name (or the name from an IP address). It helps check if domain name systems (DNS) are working properly.

Wireshark

Wireshark is a free and powerful tool used to check and understand what's happening on a network. It lets users see network traffic in real time or look at saved data to find problems. Many IT and cybersecurity professionals use it to help fix issues and keep networks secure. With its easy-to-use design, Wireshark supports many types of network systems and offers helpful features to filter, highlight, and study the data—making it easier to spot things like slow connections, lost data, or unusual activity.

Its applications include identifying performance bottlenecks, analyzing VoIP call quality, studying protocol behavior, and detecting security threats such as malware communications. However, using Wireshark requires a good understanding of network protocols, as it can only display the data it captures without decrypting encrypted traffic unless the proper keys are provided. Legal and ethical considerations are also essential, as capturing data without proper authorization may violate privacy laws and regulations [3].

Web Server

Web server is a program which processes the network requests of the users and serves them with files that create web pages. This exchange takes place using Hypertext Transfer Protocol (HTTP). Basically, web servers are computers used to store HTTP files which makes a website and when a client requests a certain website, it delivers the requested website to the client. For example, you want to open Facebook on your laptop and enter the URL in the search bar of google. Now, the laptop will send an HTTP request to view the facebook webpage to another computer known as the webserver. This computer (webserver) contains all the files (usually in HTTP format) which make up the website like text, images, gif files, etc. After processing the request, the webserver will send the requested website-

related files to your computer and then you can reach the website. Different websites can be stored on the same or different web servers but that doesn't affect the actual website that you are seeing in your computer. The web server can be any software or hardware but is usually a software running on a computer. One web server can handle multiple users at any given time which is a necessity otherwise there had to be a web server for each user and considering the current world population, is nearly close to impossible. A web server is never disconnected from the internet because if it was, then it won't be able to receive any requests, and therefore cannot process them [4].

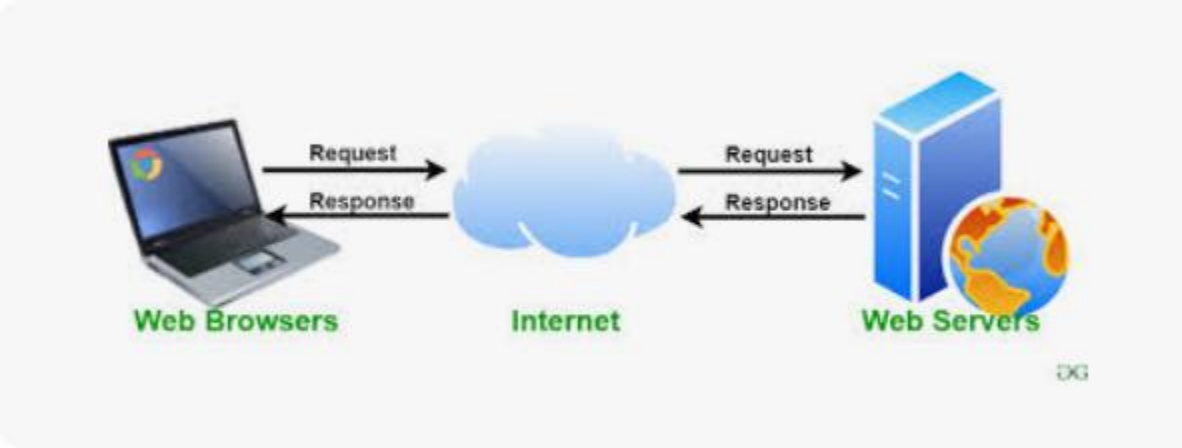


Figure 3 Web Server [4]

Methods and Components Used in Task 2

For Task 2, we developed a simple HTTP server using the socket library that listens on port 9906 to handle requests for static resources such as HTML, CSS, images, and videos. It begins by creating a TCP socket using **socket()**, binds it to the specified IP address and port using **bind()**, and prepares to accept incoming client connections with **listen()**. The server listens for requests, and once a connection is made, it accepts the client's connection with **accept()**.

The server processes incoming HTTP requests by first identifying the requested file. It checks for the existence of HTML, CSS, image, and video files. If the file is found, the server sends a 200 OK response along with the requested content using the **send_response()** method. If the file is not found, the server responds with a 404 Not Found error page designed using the **Error()** method. The error page includes the client's IP address and port number for transparency.

Additionally, for image and video requests, if the file is missing, the server issues a 307 Temporary Redirect response. In such cases, the client is redirected to Google for images or YouTube for videos for alternative search options. This approach provides a flexible way for the server to respond to various file requests, ensuring a smooth user experience even in cases of missing media files, by managing both errors and redirections effectively.

Methods and Components Used in Task 3

For Task 3, we developed a hybrid TCP/UDP server for a multiplayer guessing game, using TCP for reliable connection setup and UDP for fast-paced, real-time gameplay. The server listens on two different ports, one for TCP (port 6000) and one for UDP (port 6001). Key methods used in the code include: **bind()** to assign the server IP and ports to the sockets, **listen()** to listen for incoming TCP connections, and **accept()** to handle client connections via TCP. Once connected, the server checks for unique usernames and adds players to the game. The game starts once the minimum number of players is reached, and players send their guesses via UDP. **recvfrom()** is used to receive guesses from clients, while **sendto()** sends real-time feedback to the clients, such as "Higher," "Lower," or "Correct." The server also manages game flow by continuously checking the number of players and monitoring the guessing phase, and uses **Thread()** to handle multiple clients simultaneously. Additionally, methods like **settimeout()** ensure the

server manages time limits, and **close()** properly closes the connections after the game ends. This approach efficiently combines TCP for control messages and UDP for fast gameplay updates, creating an interactive and responsive game environment.

Results and Discussion

Task 1: Network Commands and Wireshark

Executing Network Commands

I. ipconfig /all command

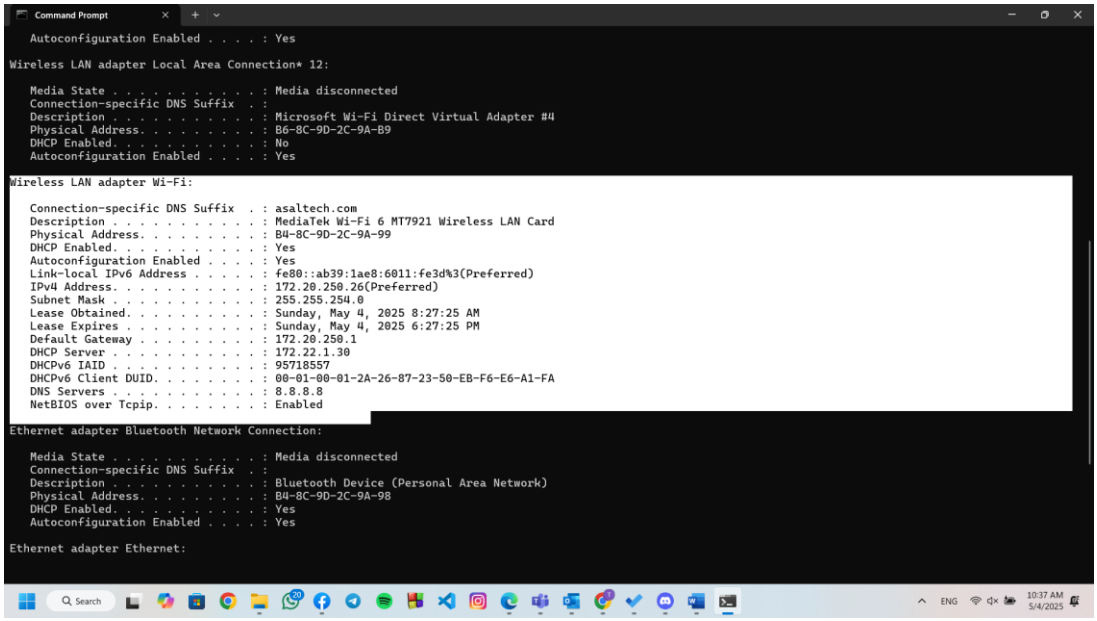


Figure 4 ipconfig/all command

- IPv4 Address: 172.20.250.26
- Subnet Mask: 255.255.254.0
- Default Gateway: 172.20.250.1
- DNS Server: 8.8.8.8

II. Ping command

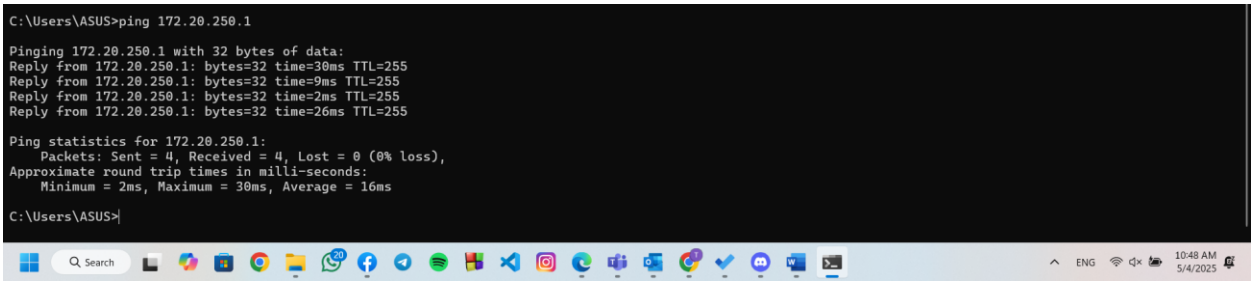


Figure 5 ping device

The ping command was used to check the connection to 172.20.250.1. All 4 packets were sent and received with no loss, showing a stable connection. The response times ranged from 2ms to 30ms, averaging 16ms. The TTL value was 255. This confirms the device is reachable and the network is working well.

III. Ping gaia.cs.umass.edu command

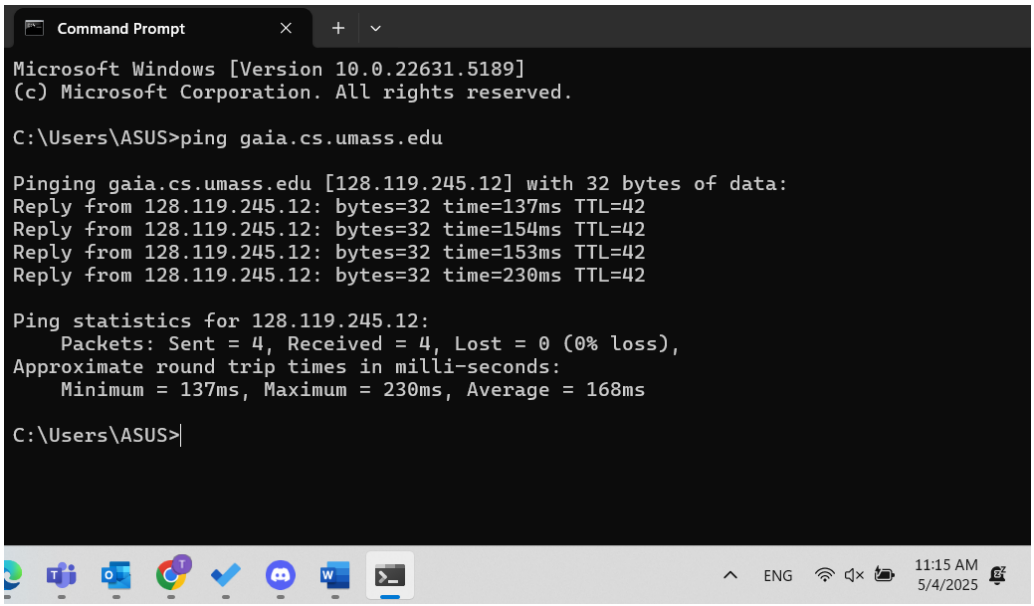


Figure 6 Ping gaia.cs.umass.edu

The server **gaia.cs.umass.edu** resolves to the IP address **128.119.245.12**. A ping test shows that four packets were sent and received successfully with **no packet loss**. The round-trip time for the packets ranged between **137 and 230ms**,

indicating stable network performance. These results confirm that the server is reachable and properly connected to the network.

Based on the results shown in the figure below, the IP address **128.119.245.12** is located in **Amherst, Massachusetts, USA**. Although the domain is associated with an educational institution, the **University of Massachusetts - Amherst**, the server is hosted in the United States. This indicates that the service is operated locally by the university.

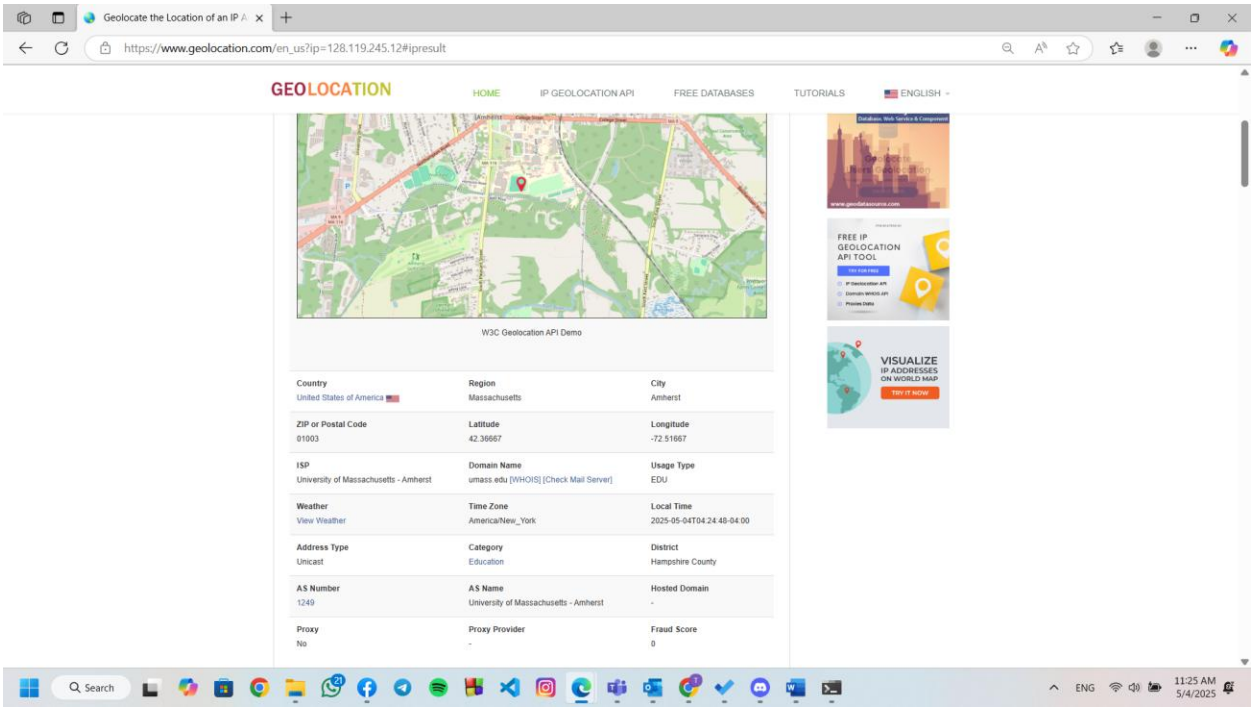


Figure 7 Location

IV. Tracert gaia.cs.umass.edu command

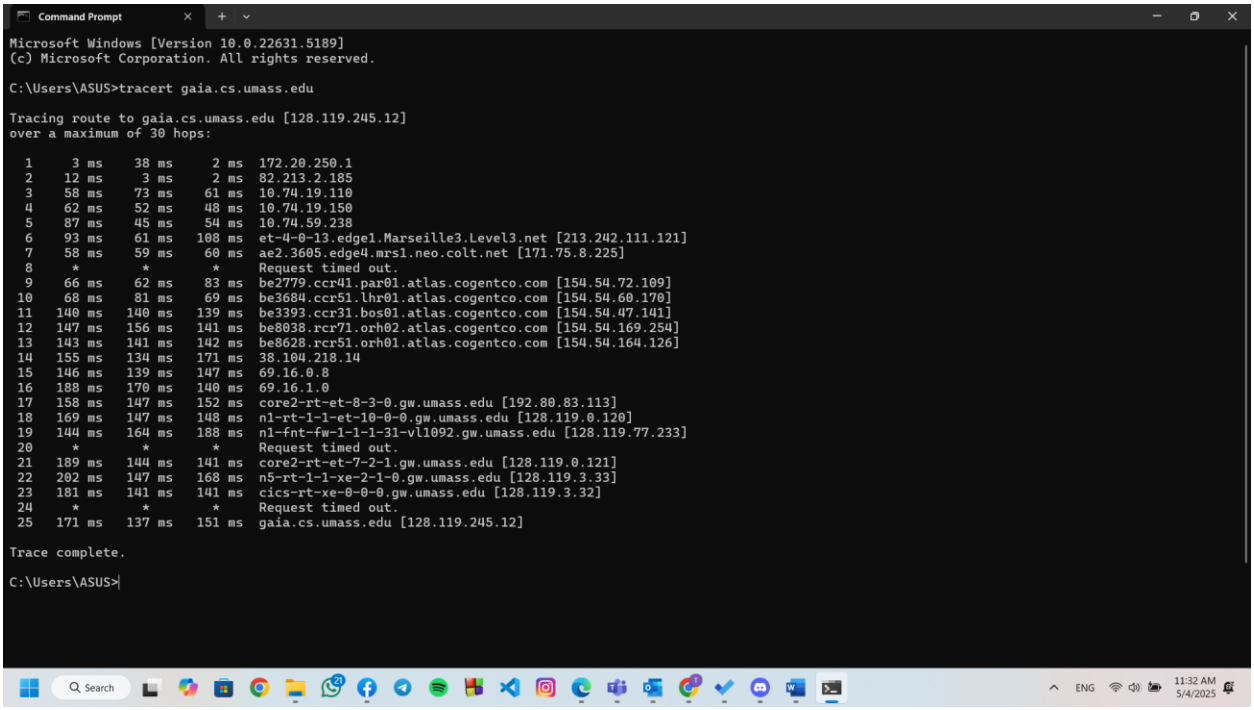


Figure 8 Tracert gaia.cs.umass.edu

The figure above displays the result of the "tracert" command, which tracks the route a packet follows from the user's device to its destination. Here, the target IP address is **128.119.245.12**. The command reveals each step the packet takes, showing the IP addresses of the routers it passes through. The final IP address shown is that of the destination, confirming the packet has successfully arrived.

V. nslookup gaia.cs.umass.edu command

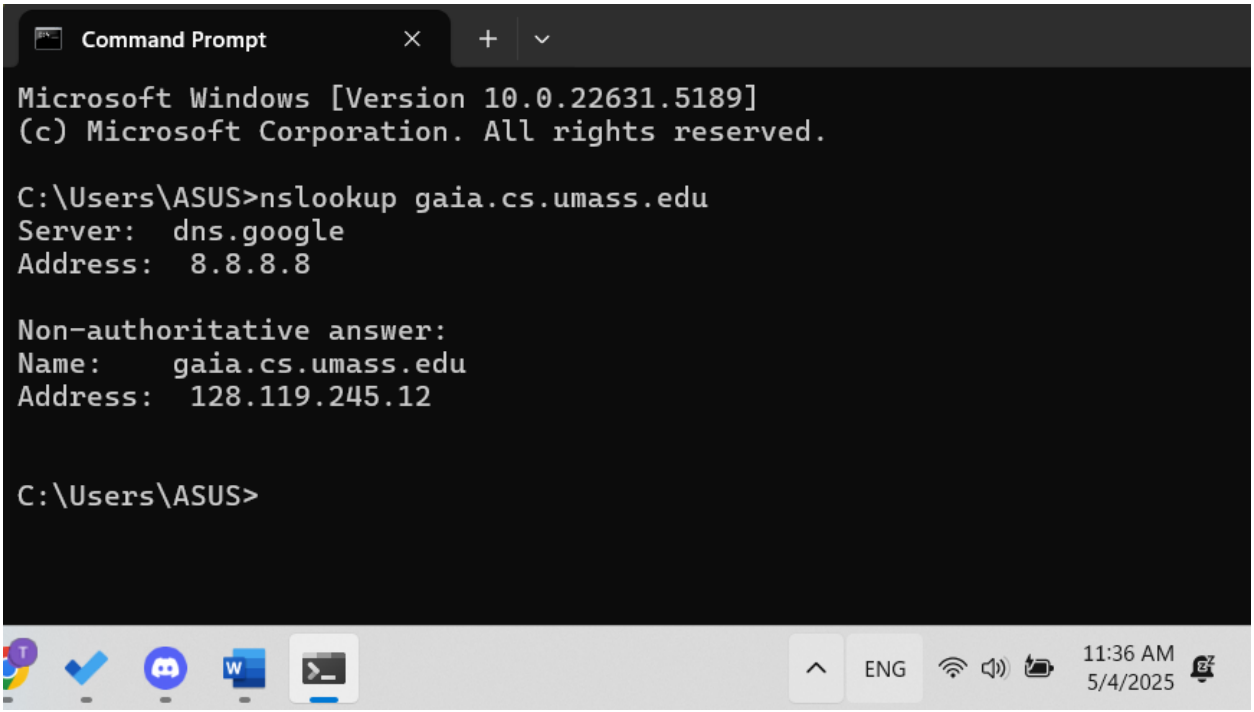


Figure 9 nslookup gaia.cs.umass.edu

The figure above shows the result of the "nslookup" command, which translates a domain name into its IP address. It also displays the DNS server used, helping verify domain resolution and DNS settings.

VI. telnet gaia.cs.umass.edu command

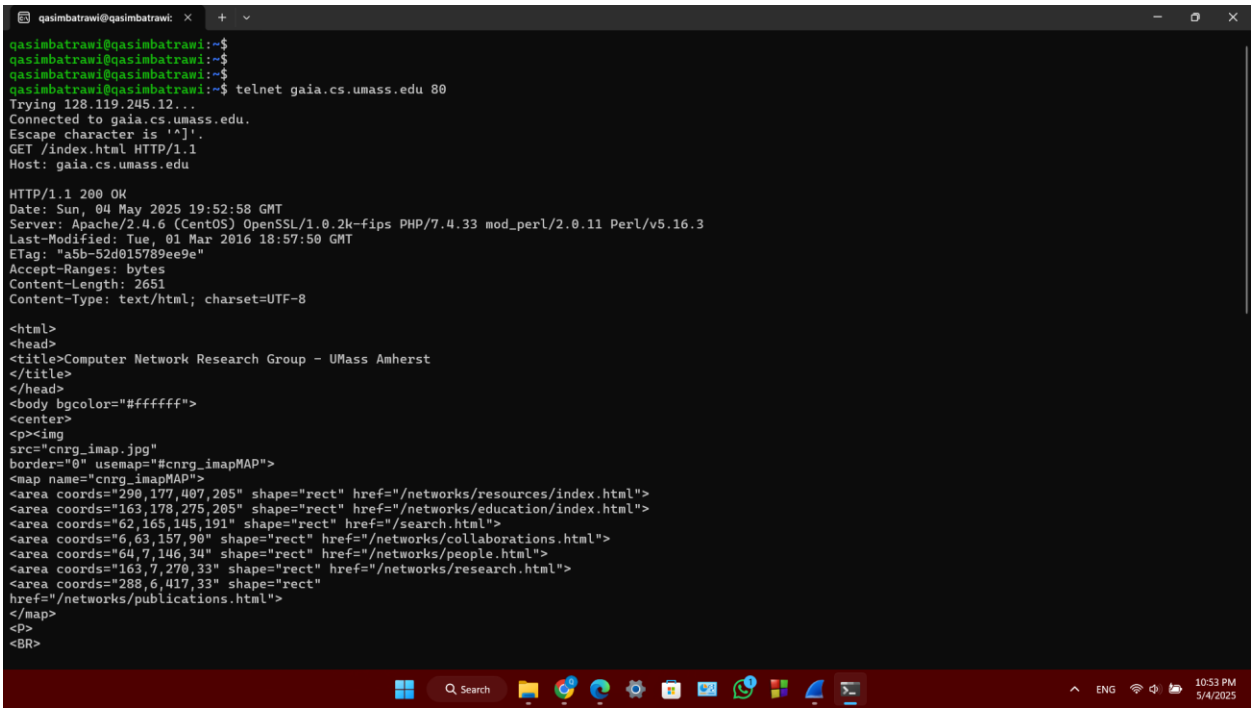


Figure 10 telnet gaia.cs.umass.edu

The result shows a successful connection to gaia.cs.umass.edu on port 80, which is the default port for HTTP communication. Using telnet command, we have sent an HTTP GET request for the /index.html page and the server responded with 200 OK status, followed by the HTML content page.

VII. BGP and AS Information Lookup for gaia.cs.umass.edu

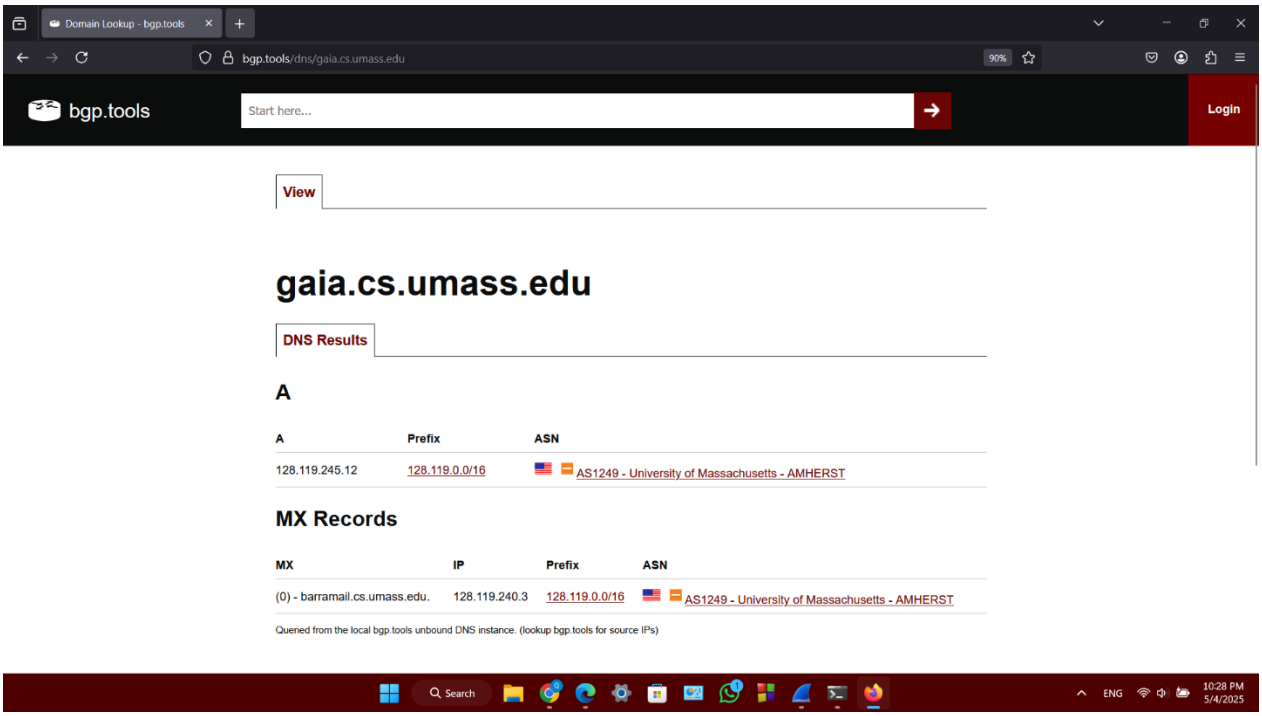


Figure 11 AS Information for gaia.cs.umass.edu

Using http://bgp.tools/, the figure above shows that gaia.cs.umass.edu has an autonomous system number of **AS1429**, which is registered to the University of Massachusetts Amherst. Using www.bgpview.io, the following information was obtained:

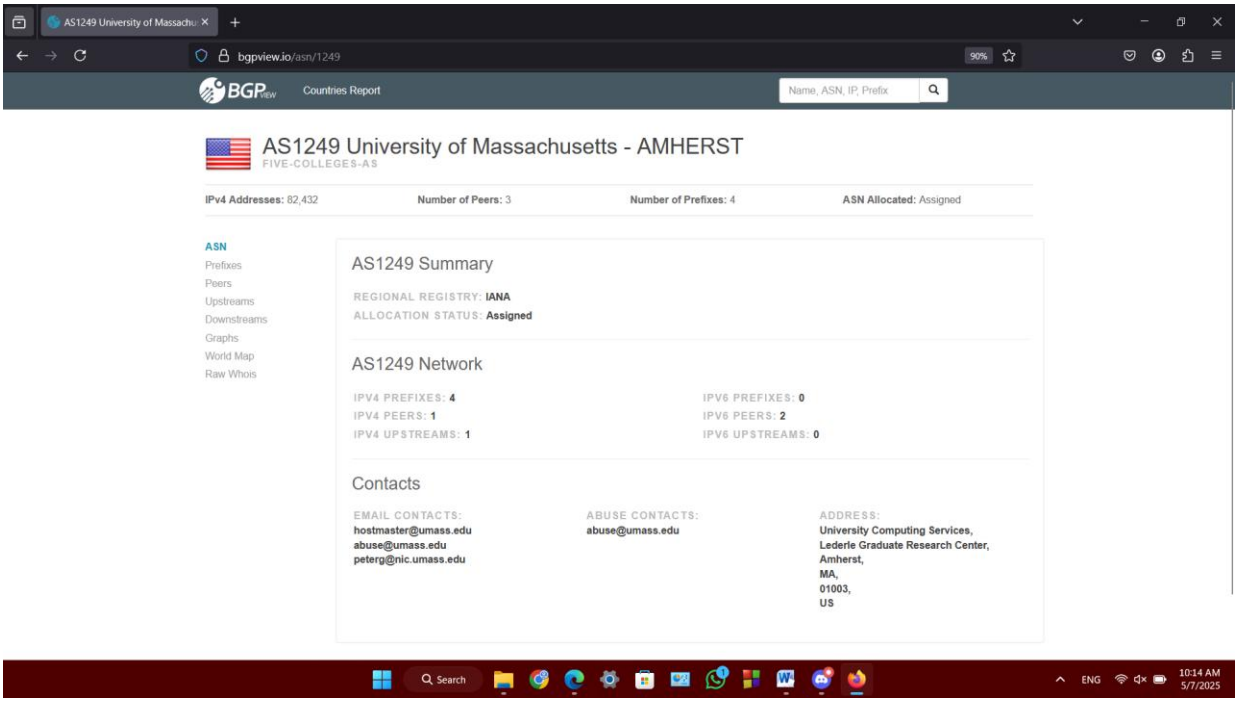


Figure 12 Information about gaia.cs.umass.edu

- Number of IP addresses: 82,432.
- Number of prefixes: 4 prefixes for IPv4, and 0 for IPv6.
- Number of peers: 1 for IPv4, and 2 for IPv6.

Results shows that the autonomous system AS1249 has a total of 82,432 unique IPv4 addresses, 4 blocks of IPv4 addresses and 0 for IPv6, 1 direct peer for IPv4 and 2 for IPv6. The following figure shows the routing relationships between AS1249 and other networks in the internet:

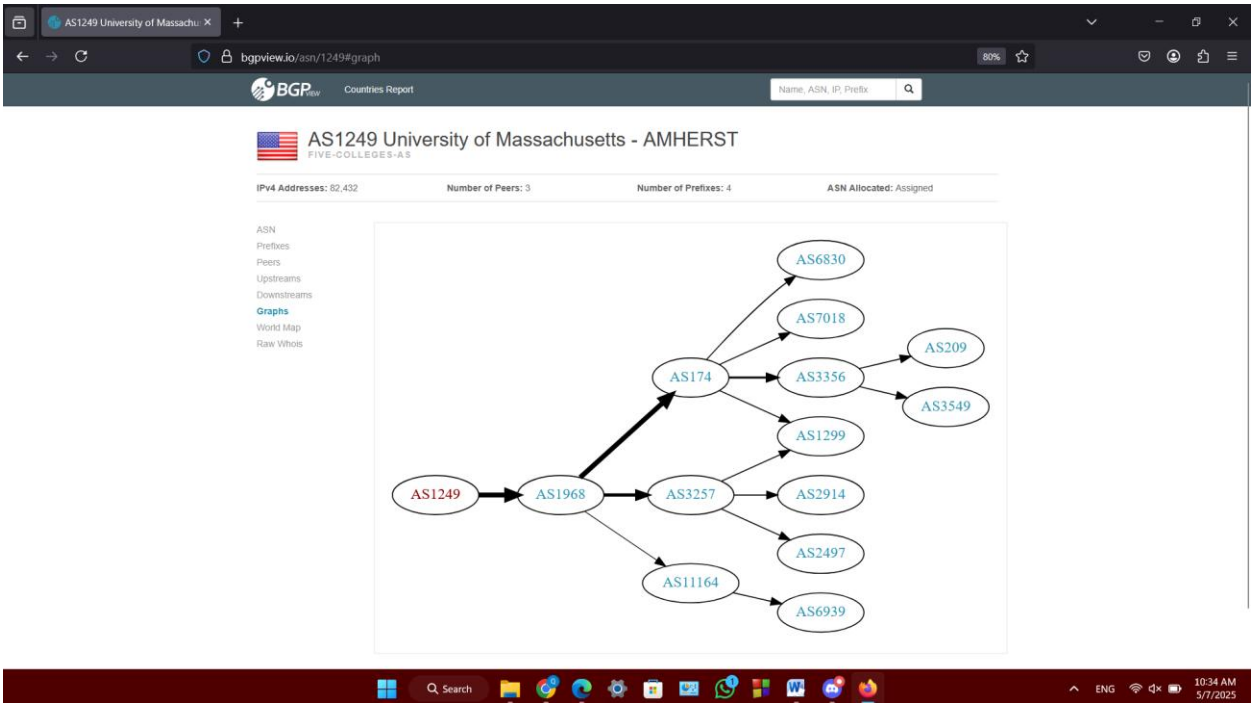


Figure 13 Routing relationships between AS1249 and other networks in the internet

Searching the list of Tier 1 ISPs on Wikipedia and looking at the autonomous systems in the figure above, we have reached that the following are considered as Tier 1 ISPs:

- AS7018 – AT&T
- AS3356 – Lumen Technologies
- AS1299 – Arelion
- AS6830 – Liberty Global
- AS3257 – GTT Communications
- AS2914 – NTT Communications

Wireshark DNS Capture for gaia.cs.umass.edu

To capture a DNS query and response for gaia.cs.umass.edu, the cache has been cleared and the following result is obtained using Wireshark:

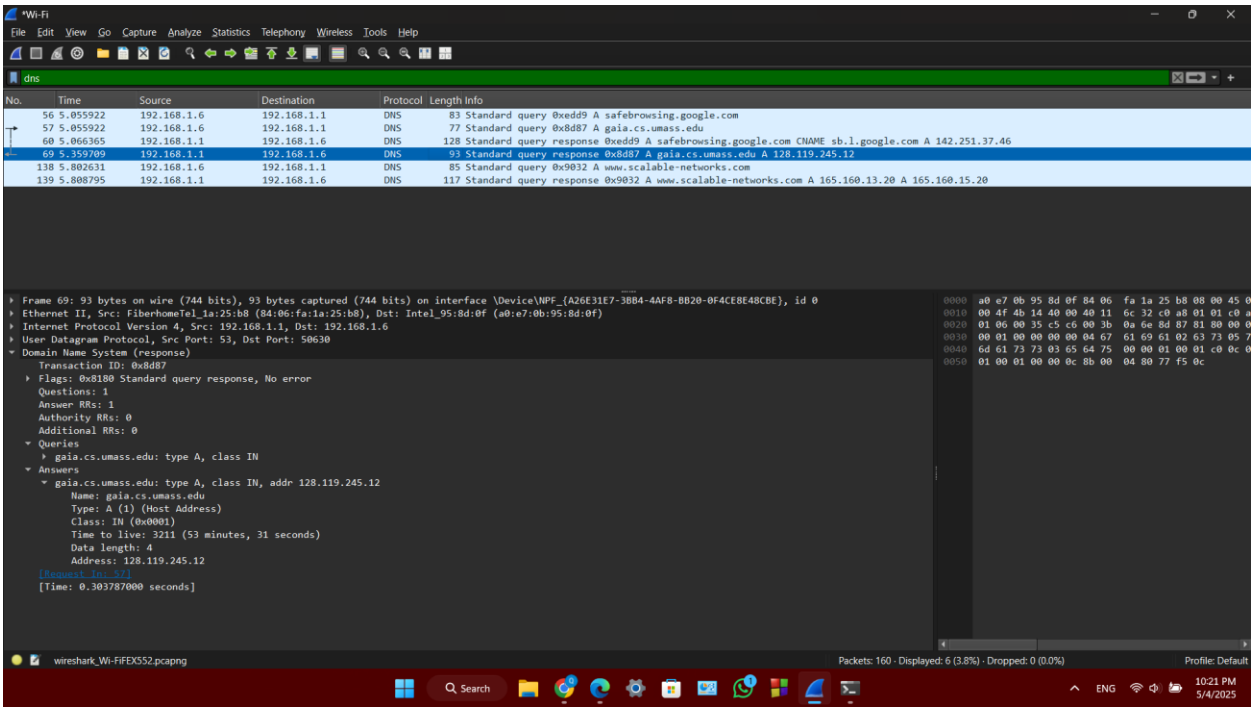


Figure 14 Capturing DNS Query.

As shown in the figure, in packet 57, the client device with IP address 192.168.1.1 sends a DNS query to the DNS server with IP 192.168.1.6. The client is requesting for the IP address of gaia.cs.umass.edu, which uses UDP on port 53. The DNS server responds in packet 69 showing the IP address for gaia.cs.umass.edu, which is 128.119.245.12.

Task 2: Simple Web Server Using Socket Programming

In this task, we were asked to build a web server using socket programming. The server uses the TCP protocol to handle HTTP requests from clients and listens on port **9906**.

First, we created several web pages using HTML and CSS: the main English page, the main Arabic page, the Local Web English page, and the Local Web Arabic page. Our server is designed to support all of these pages.

After that, we developed the web server using Python socket programming. It listens for client connections on port **9906** and uses TCP sockets to receive and respond to HTTP requests. The server replies with a **200 OK** status when the requested file is available, sends a **307 Temporary Redirect** to Google (for images) or YouTube (for videos) if the file is not on the server, and responds with **404 Not Found** if the requested item doesn't exist at all. The functionality and implementation details of the server will be explained in depth throughout this report.

Webpages

Main English Webpage (main_en.html):

This webpage serves as the main page of the English version. It includes the page title, detailed information about the team members, and a link to the Local Webpage. As shown in the figure below, the browser tab displays the title **"ENCS3320-Webserver"** in red, and the main heading on the page is **"Welcome to ENCS3320 - Computer Networks Webserver"**, presented with a background image. The page also features three navigation buttons that link to the Birzeit University website, the textbook website, and the Local Webpage.

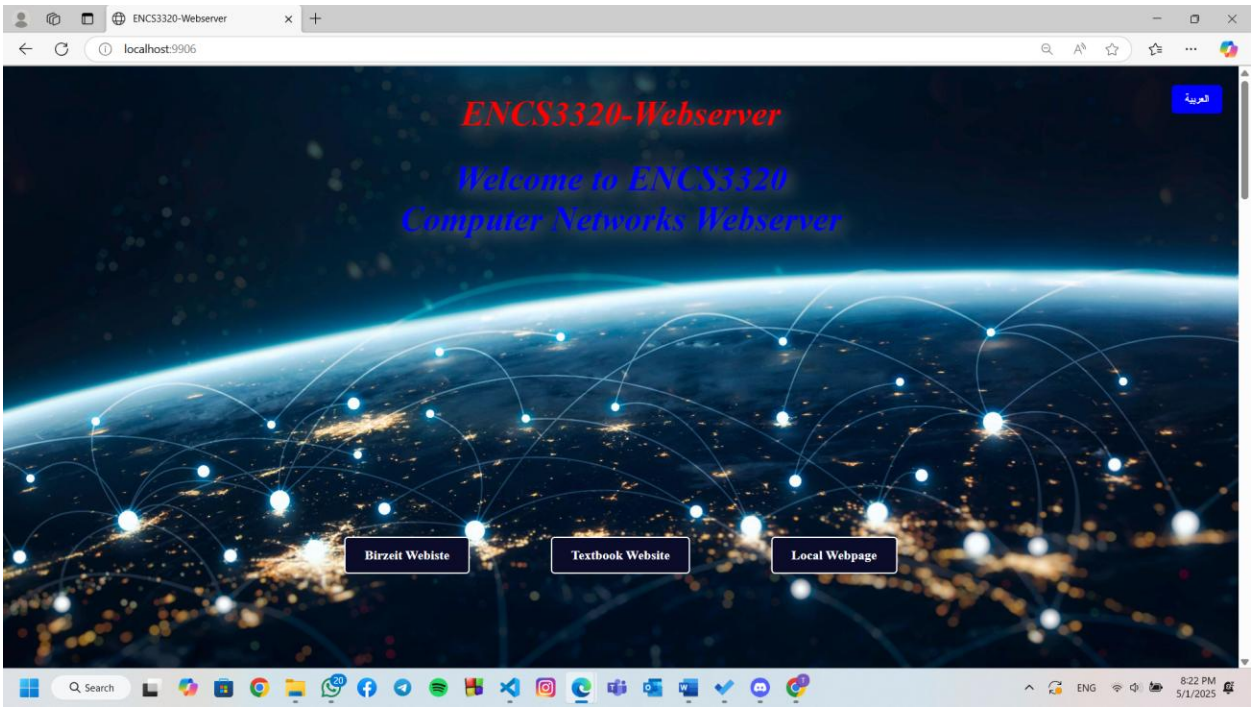


Figure 15 Main English Webpage 1

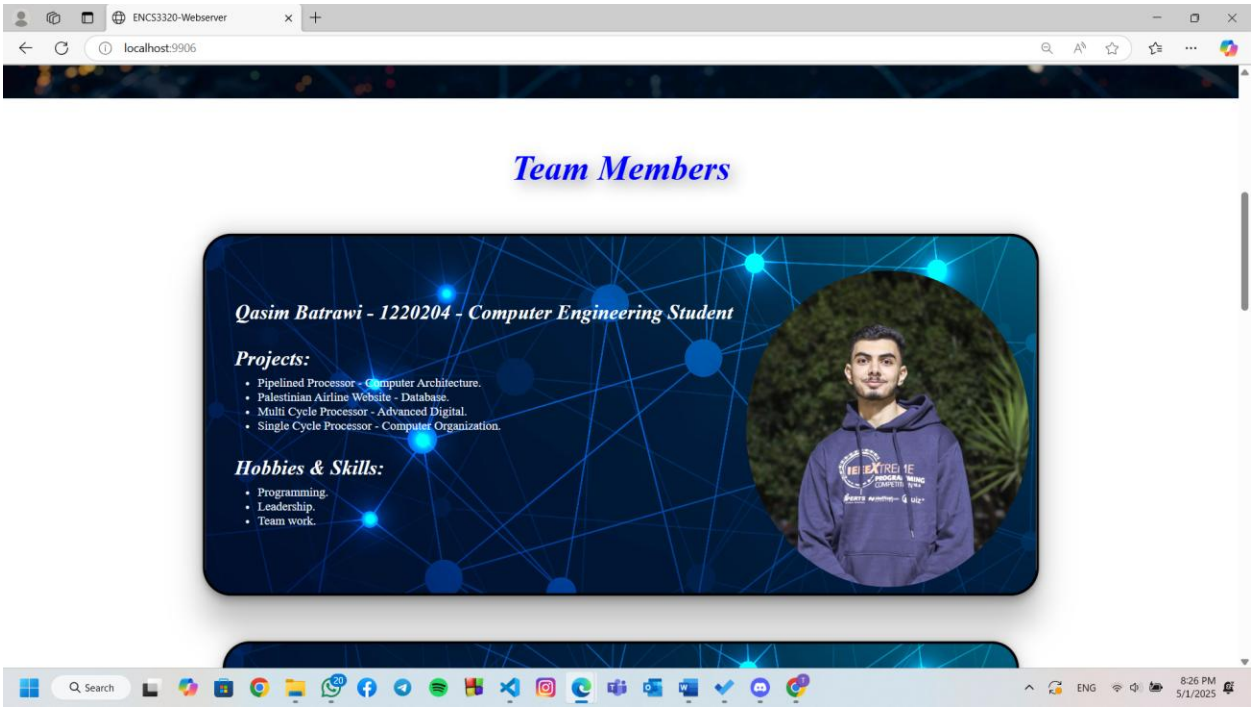


Figure 16 Main English Webpage 2

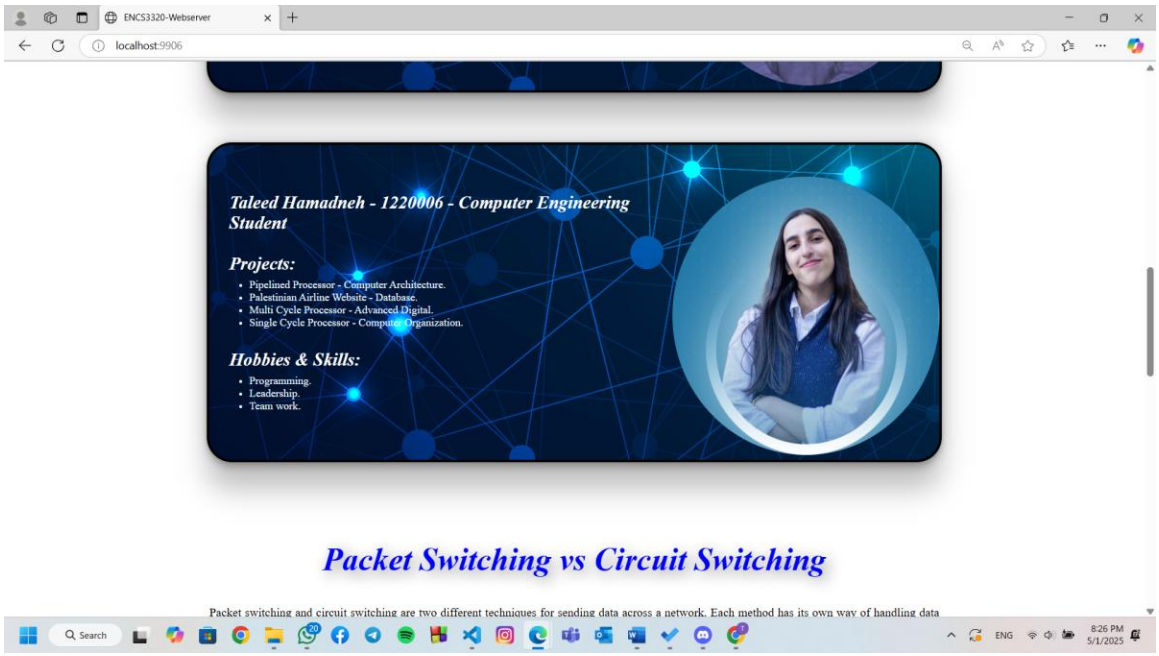


Figure 17 Main English Webpage 3

The team members' section presents individual profiles, including each member's name, student ID, academic major, completed projects, skills, and hobbies. Each profile is accompanied by a personal photo and is organized in a clear, structured layout, as shown in the figures above.

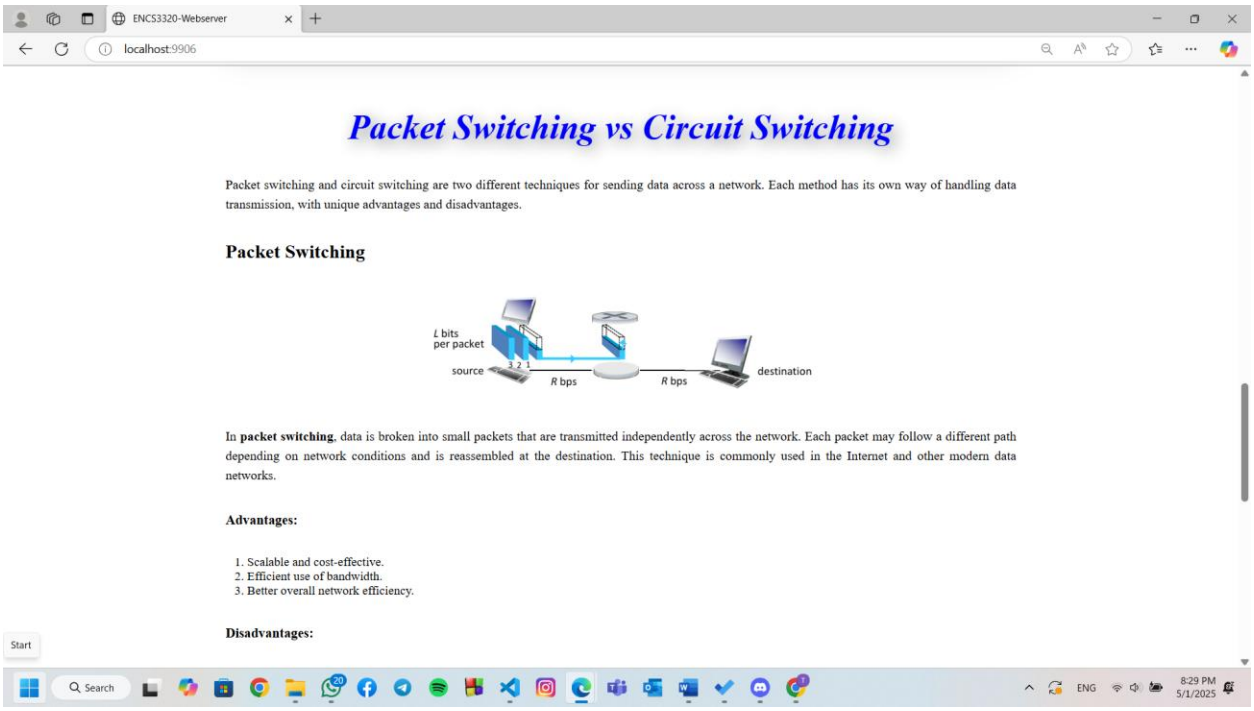


Figure 18 Main English Webpage 4

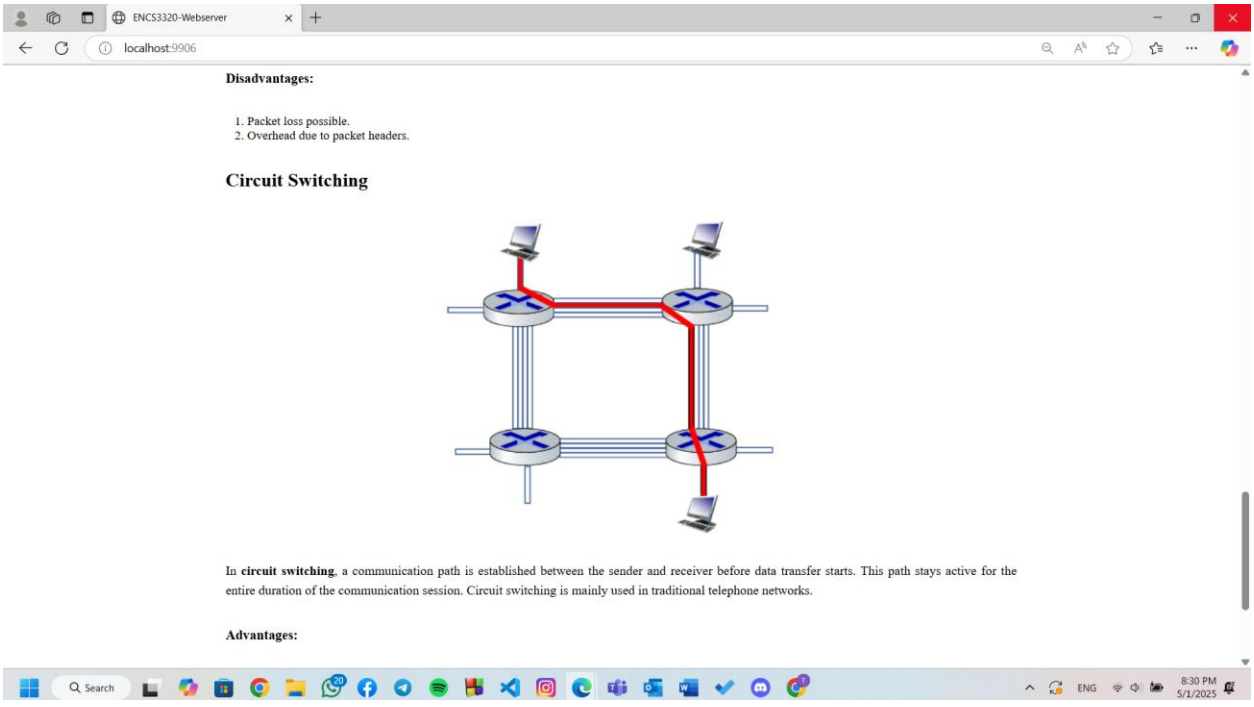


Figure 19 Main English Webpage 5

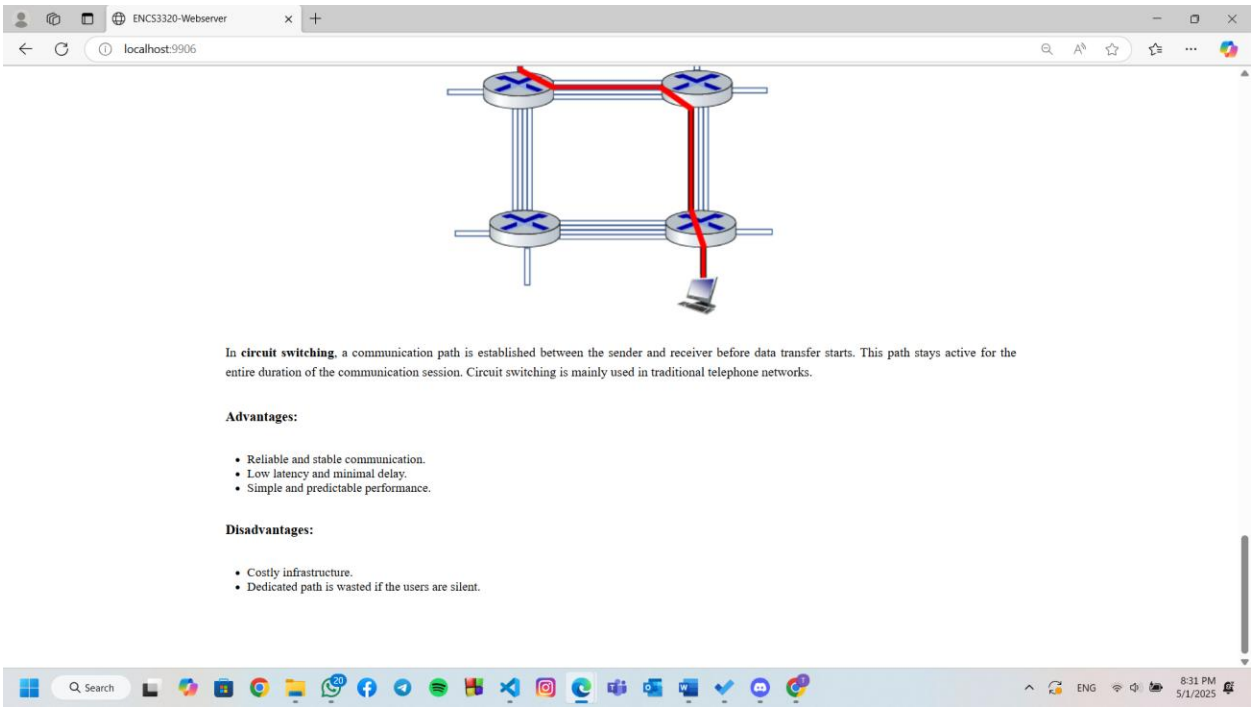


Figure 20 Main English Webpage 6

In the three figures above, a well-structured comparison between packet switching and circuit switching is presented as part of the first chapter topic from the textbook. Each technique is explained through clear headings, descriptive paragraphs, and relevant images in .jpg format. The packet switching section includes an ordered list highlighting its advantages and disadvantages, such as efficient bandwidth usage and potential packet loss. Similarly, the circuit switching section is supported by an unordered list, outlining its stable performance and higher cost. This section effectively uses headings, formatted text, and color styling to enhance readability and support the explanation visually and textually.

Main Arabic Webpage (main_ar.html):

This webpage serves as the main page of the Arabic version and contains the same content as the English main page, but presented in the Arabic language.

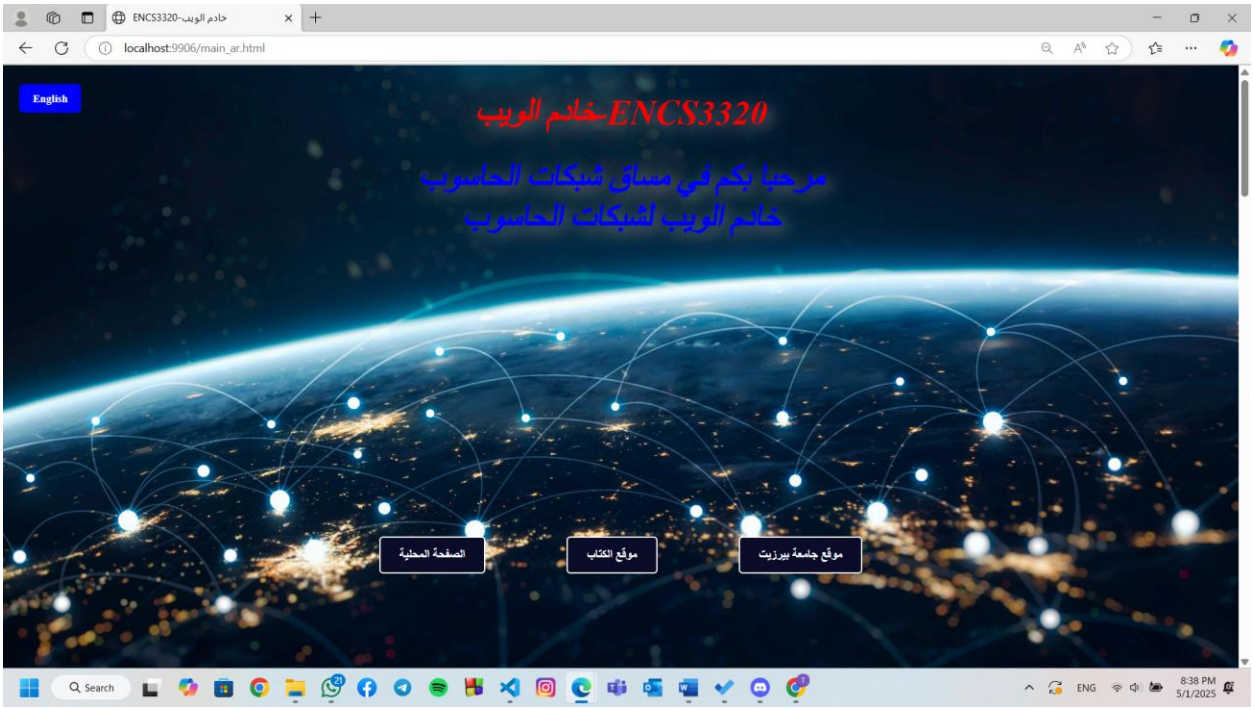


Figure 21 Main Arabic Webpage 1

Figure 22 Main Arabic Webpage 2

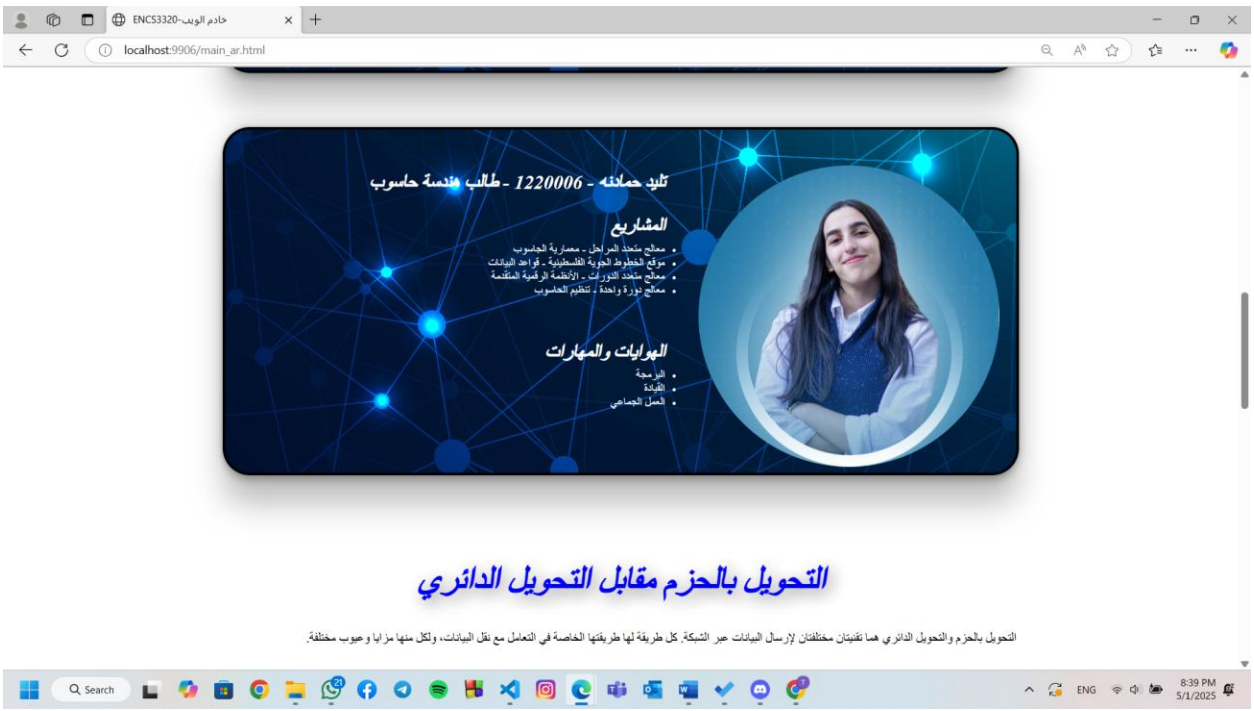


Figure 23 Main Arabic Webpage 3

Figure 24 Main Arabic Webpage 4

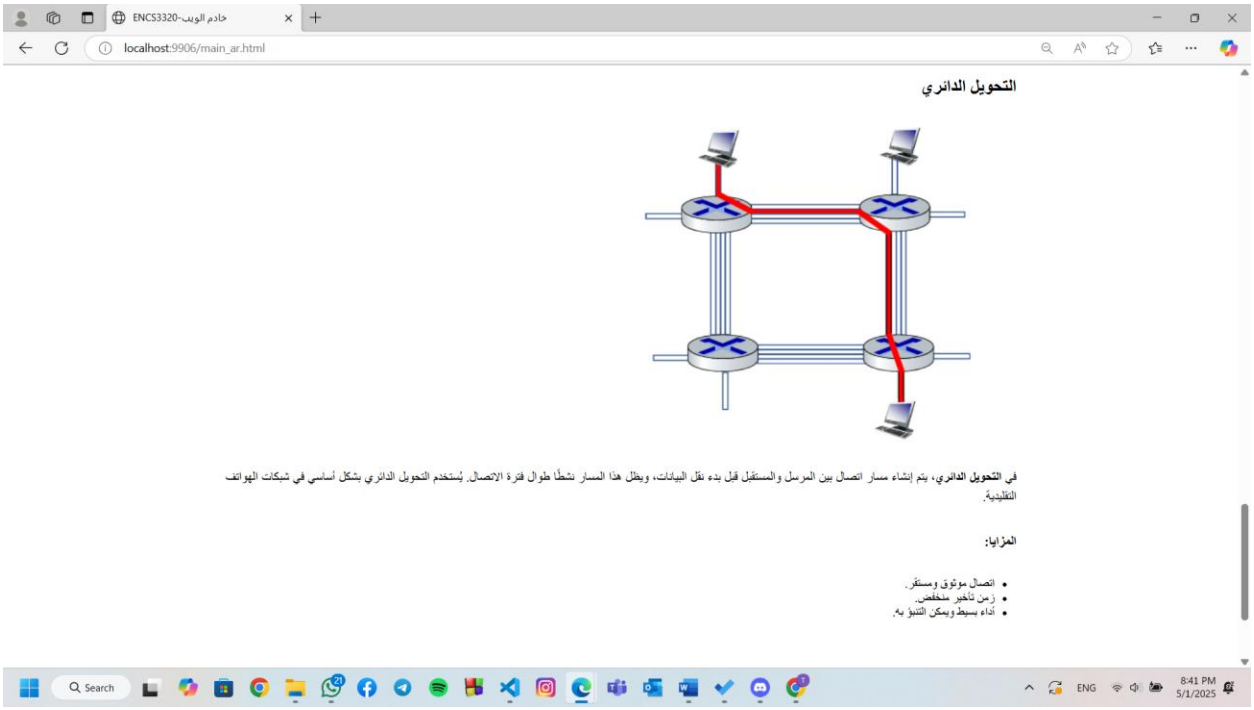


Figure 25 Main Arabic Webpage 5

Figure 26 Main Arabic Webpage 6

English Local Webpage (mySite_1220006_en.html):

The Local Webpage, as shown in the figure below, is designed to provide users with supporting material related to the computer networking topic introduced in the main English page. It features a form where users can enter the name of an image or video file they wish to request. If the requested file is available on the server, it is returned and displayed accordingly. However, if the file is not found, the server responds with a "307 Temporary Redirect" status code. Based on the file type, the user is then redirected either to a Google Images search (for images) or to a video search (for videos), allowing them to find the content externally.

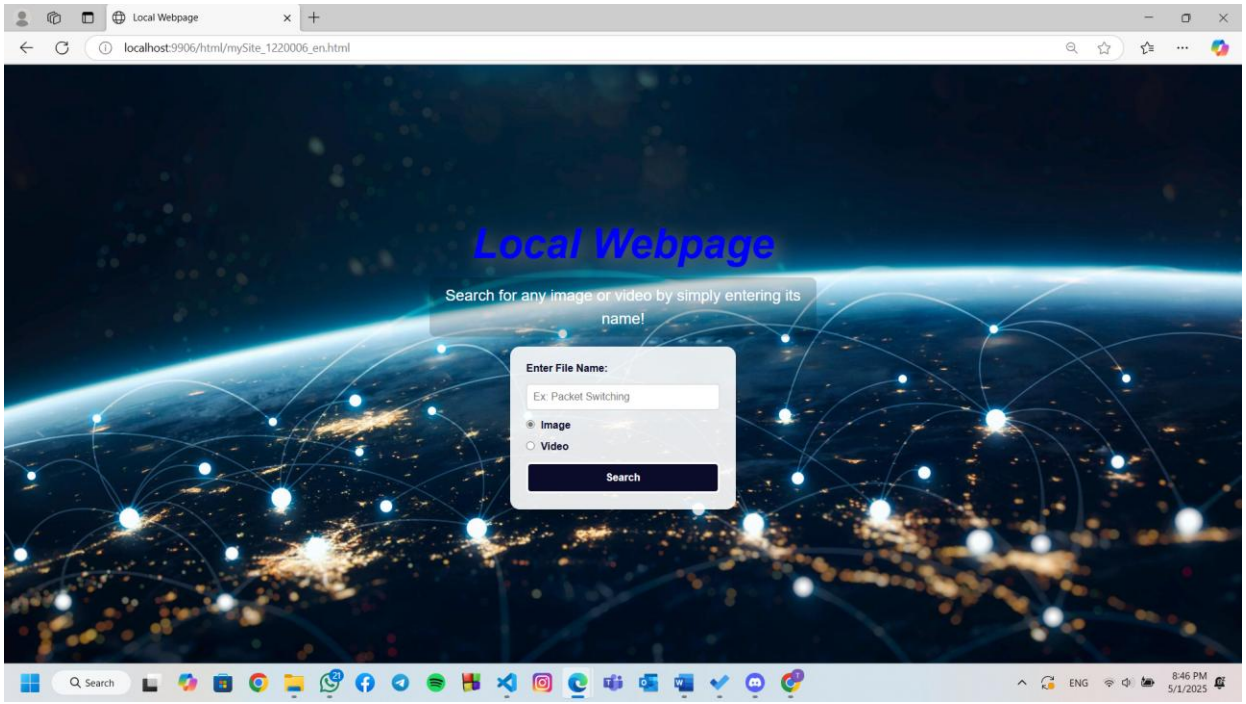


Figure 27 English Local Webpage

Arabic Local Webpage (mySite_1220006_ar.html):

The Arabic version of this page contains the same content and functionality, but is fully translated into Arabic.

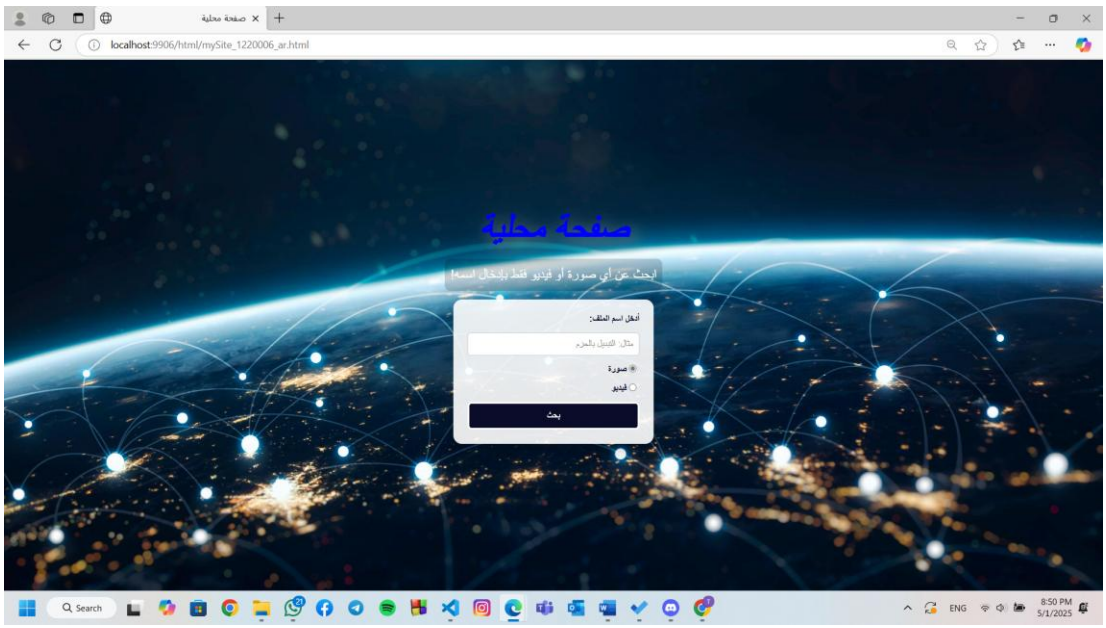
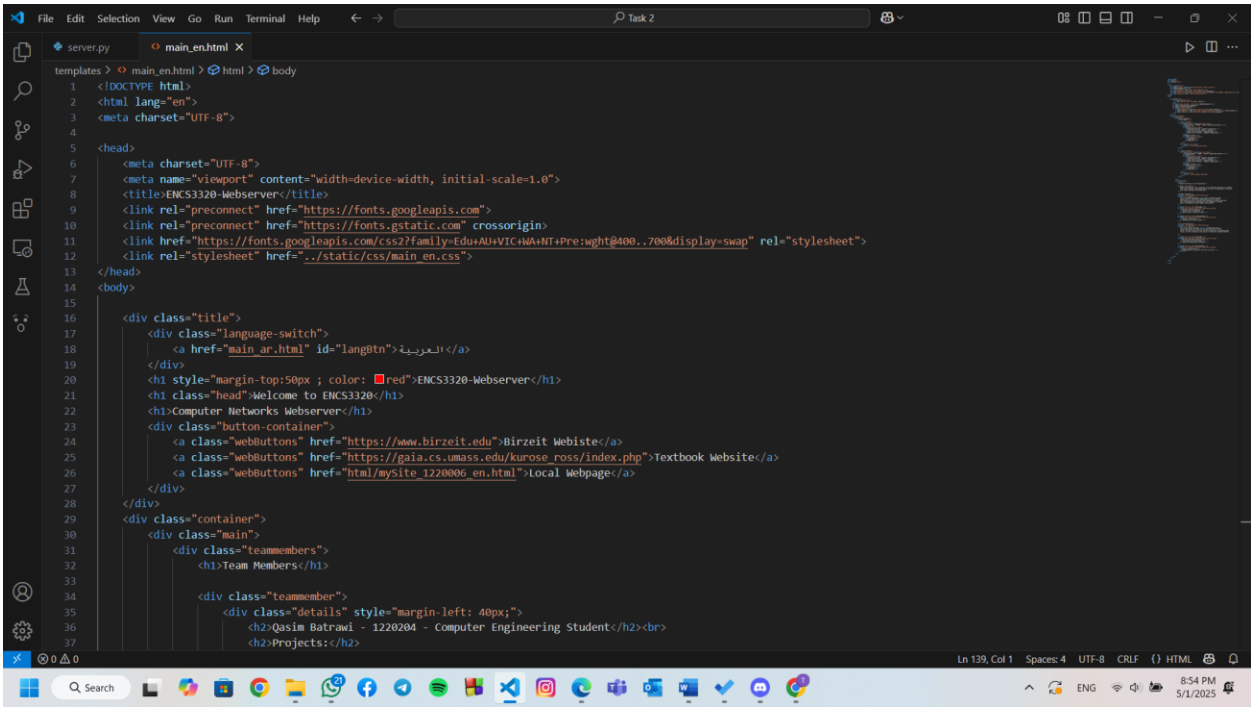


Figure 28 Arabic Local Webpage

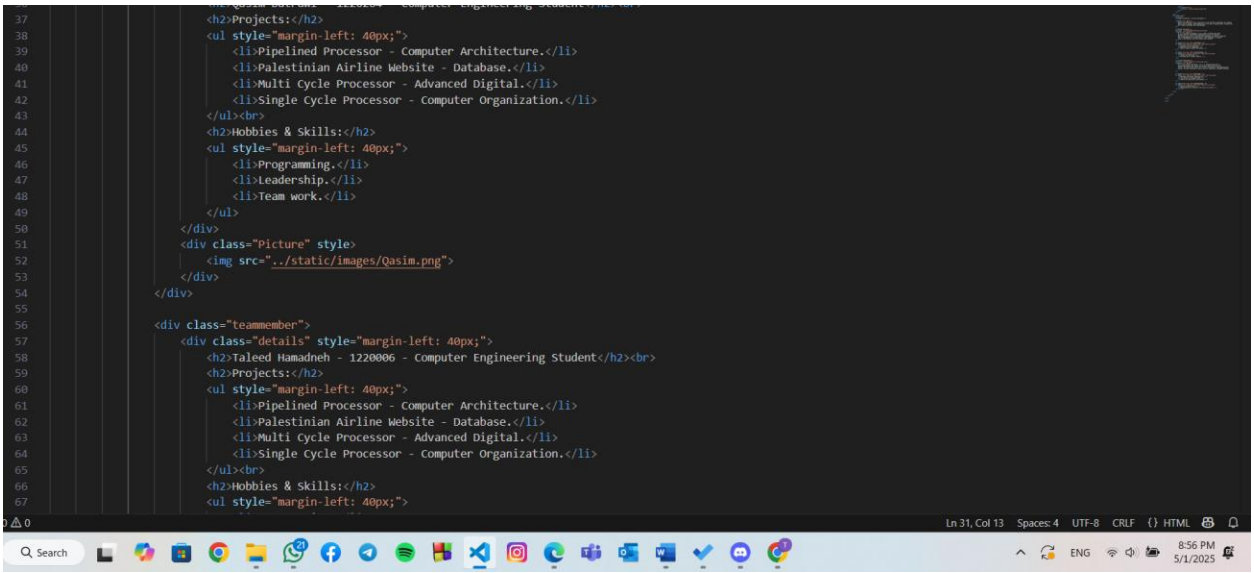
HTML & CSS

Main English Webpage HTML



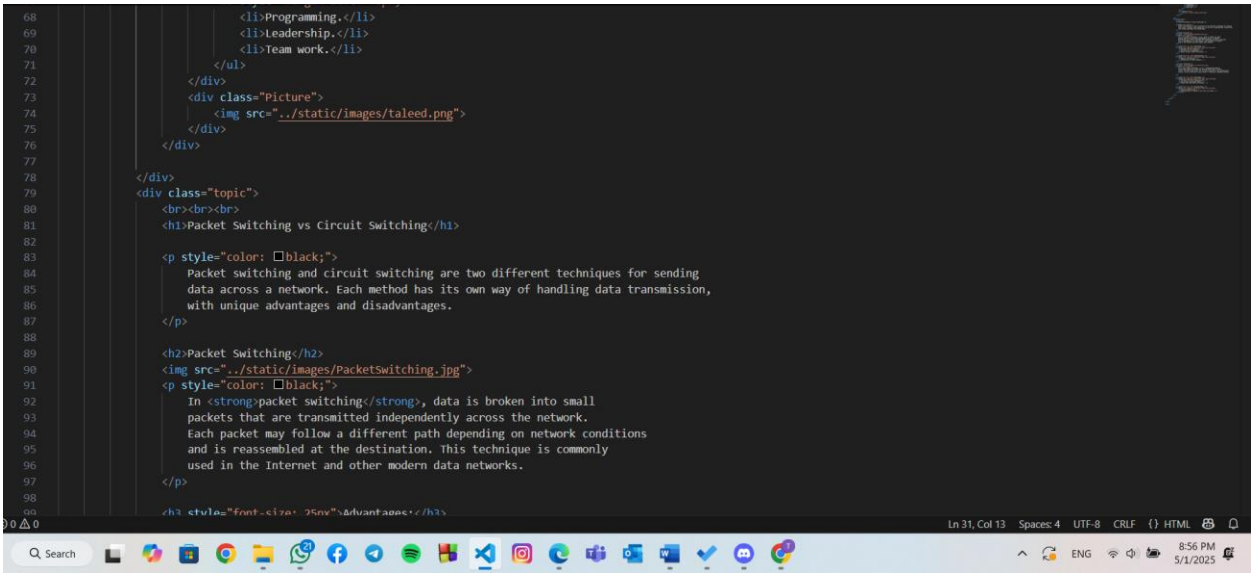
```
<!DOCTYPE html>
<html lang="en">
<meta charset="UTF-8">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ENCS3320-Webserver</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Edu+AU+VIC+WA+NT+Pre:wght@400..700&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="../static/css/main_en.css">
</head>
<body>

    <div class="title">
        <div class="language-switch">
            <a href="main_ar.html" id="langBtn">العربية</a>
        </div>
        <h1 style="margin-top:50px ; color: red">ENCS3320-Webserver</h1>
        <h1 class="head">Welcome to ENCS3320</h1>
        <h1>Computer Networks Webserver</h1>
        <div class="button-container">
            <a class="webButtons" href="https://www.birzeit.edu">Birzeit Webiste</a>
            <a class="webButtons" href="https://gaia.cs.umass.edu/kurose_ross/index.php">Textbook Website</a>
            <a class="webButtons" href="html/mySite_1220006_en.html">Local Webpage</a>
        </div>
    </div>
    <div class="container">
        <div class="main">
            <div class="teammembers">
                <h1>Team Members</h1>

                <div class="teammember">
                    <div class="details" style="margin-left: 40px;">
                        <h2>Qasim Batrawi - 1220204 - Computer Engineering Student</h2><br>
                        <h2>Projects:</h2>
```

Figure 29 Main English Webpage HTML 1



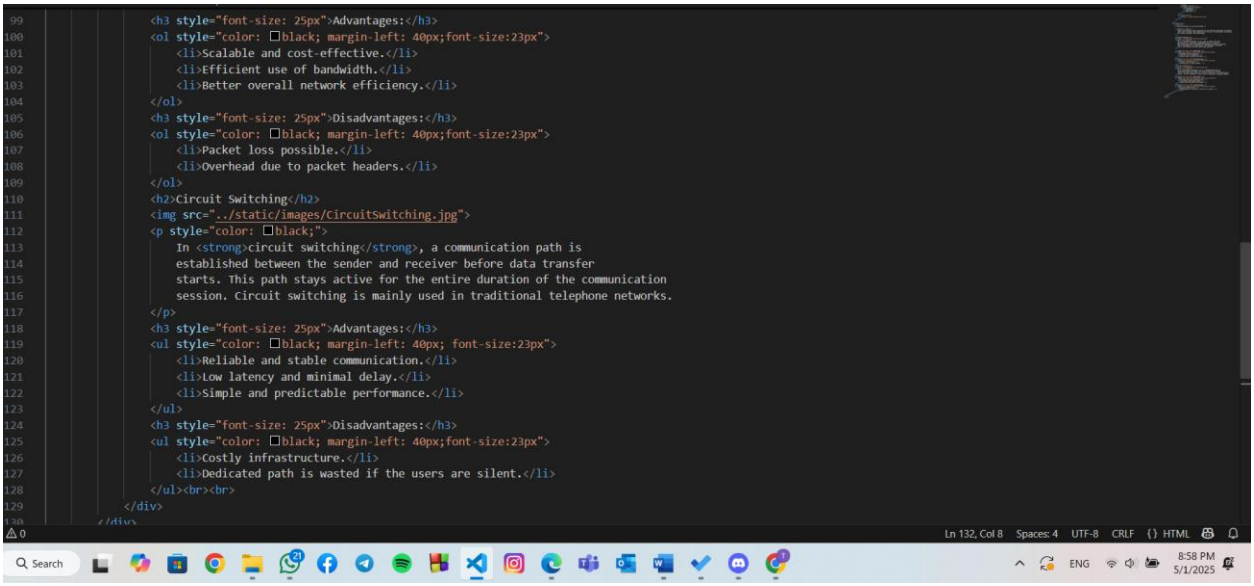
```
                        <h2>Projects:</h2>
                        <ul style="margin-left: 40px;">
                            <li>Pipelined Processor - Computer Architecture.</li>
                            <li>Palestinian Airline Website - Database.</li>
                            <li>Multi Cycle Processor - Advanced Digital.</li>
                            <li>Single Cycle Processor - Computer Organization.</li>
                        </ul><br>
                        <h2>Hobbies & Skills:</h2>
                        <ul style="margin-left: 40px;">
                            <li>Programming.</li>
                            <li>Leadership.</li>
                            <li>Team work.</li>
                        </ul>
                    </div>
                    <div class="Picture" style>
                        <img src="../static/images/Qasim.png">
                    </div>
                </div>

                <div class="teammember">
                    <div class="details" style="margin-left: 40px;">
                        <h2>Taleed Hamadneh - 1220006 - Computer Engineering Student</h2><br>
                        <h2>Projects:</h2>
                        <ul style="margin-left: 40px;">
                            <li>Pipelined Processor - Computer Architecture.</li>
                            <li>Palestinian Airline Website - Database.</li>
                            <li>Multi Cycle Processor - Advanced Digital.</li>
                            <li>Single Cycle Processor - Computer Organization.</li>
                        </ul><br>
                        <h2>Hobbies & Skills:</h2>
                        <ul style="margin-left: 40px;">
```

Figure 30 Main English Webpage HTML 2



```
                        <li>Programming.</li>
                        <li>Leadership.</li>
                        <li>Team work.</li>
                    </ul>
                </div>
                <div class="Picture">
                    <img src="../static/images/taleed.png">
                </div>
            </div>

        </div>
        <div class="topic">
            <br><br><br>
            <h1>Packet Switching vs Circuit Switching</h1>

            <p style="color: black;">
                Packet switching and circuit switching are two different techniques for sending
                data across a network. Each method has its own way of handling data transmission,
                with unique advantages and disadvantages.
            </p>

            <h2>Packet Switching</h2>
            <img src="../static/images/PacketSwitching.jpg">
            <p style="color: black;">
                In <strong>packet switching</strong>, data is broken into small
                packets that are transmitted independently across the network.
                Each packet may follow a different path depending on network conditions
                and is reassembled at the destination. This technique is commonly
                used in the Internet and other modern data networks.
            </p>

            <h3 style="font-size: 25px">Advantages:</h3>
```

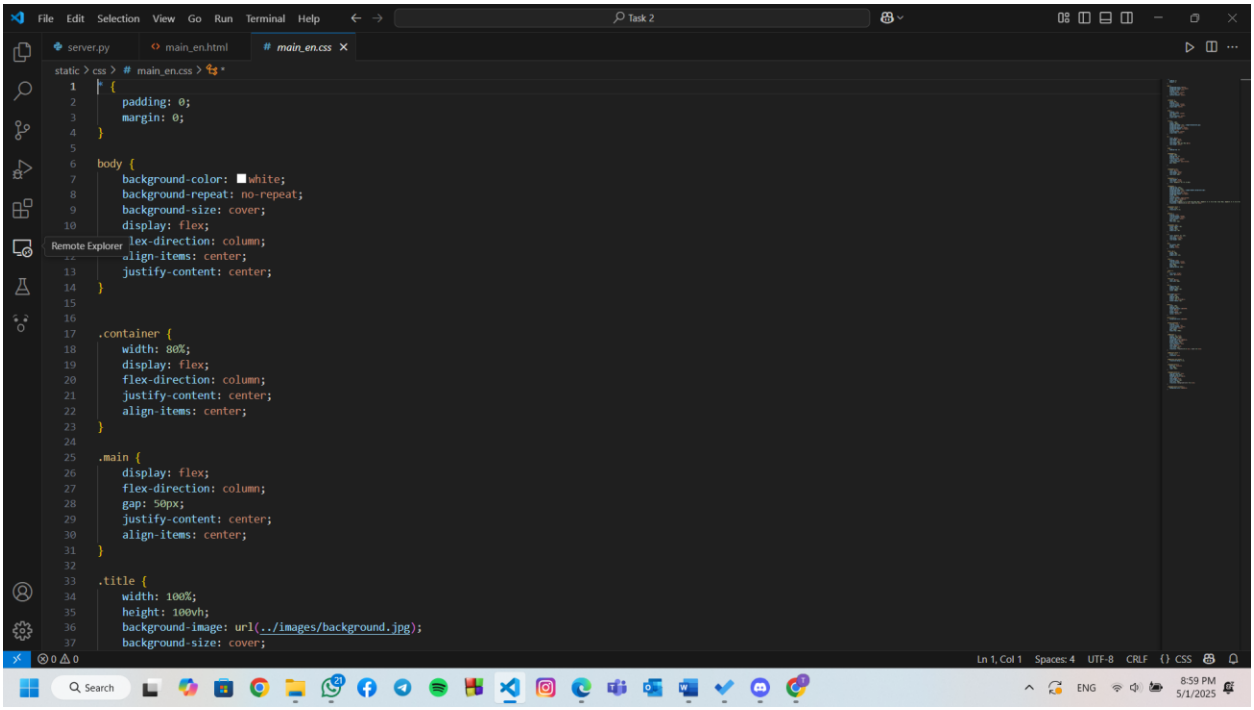
Figure 31 Main English Webpage HTML 3



```
            <h3 style="font-size: 25px">Advantages:</h3>
            <ol style="color: black; margin-left: 40px;font-size:23px">
                <li>Scalable and cost-effective.</li>
                <li>Efficient use of bandwidth.</li>
                <li>Better overall network efficiency.</li>
            </ol>
            <h3 style="font-size: 25px">Disadvantages:</h3>
            <ol style="color: black; margin-left: 40px;font-size:23px">
                <li>Packet loss possible.</li>
                <li>Overhead due to packet headers.</li>
            </ol>
            <h2>Circuit Switching</h2>
            <img src="../static/images/CircuitSwitching.jpg">
            <p style="color: black;">
                In <strong>circuit switching</strong>, a communication path is
                established between the sender and receiver before data transfer
                starts. This path stays active for the entire duration of the communication
                session. Circuit switching is mainly used in traditional telephone networks.
            </p>
            <h3 style="font-size: 25px">Advantages:</h3>
            <ul style="color: black; margin-left: 40px; font-size:23px">
                <li>Reliable and stable communication.</li>
                <li>Low latency and minimal delay.</li>
                <li>Simple and predictable performance.</li>
            </ul>
            <h3 style="font-size: 25px">Disadvantages:</h3>
            <ul style="color: black; margin-left: 40px;font-size:23px">
                <li>Costly infrastructure.</li>
                <li>Dedicated path is wasted if the users are silent.</li>
            </ul><br><br>
        </div>
    </div>
```

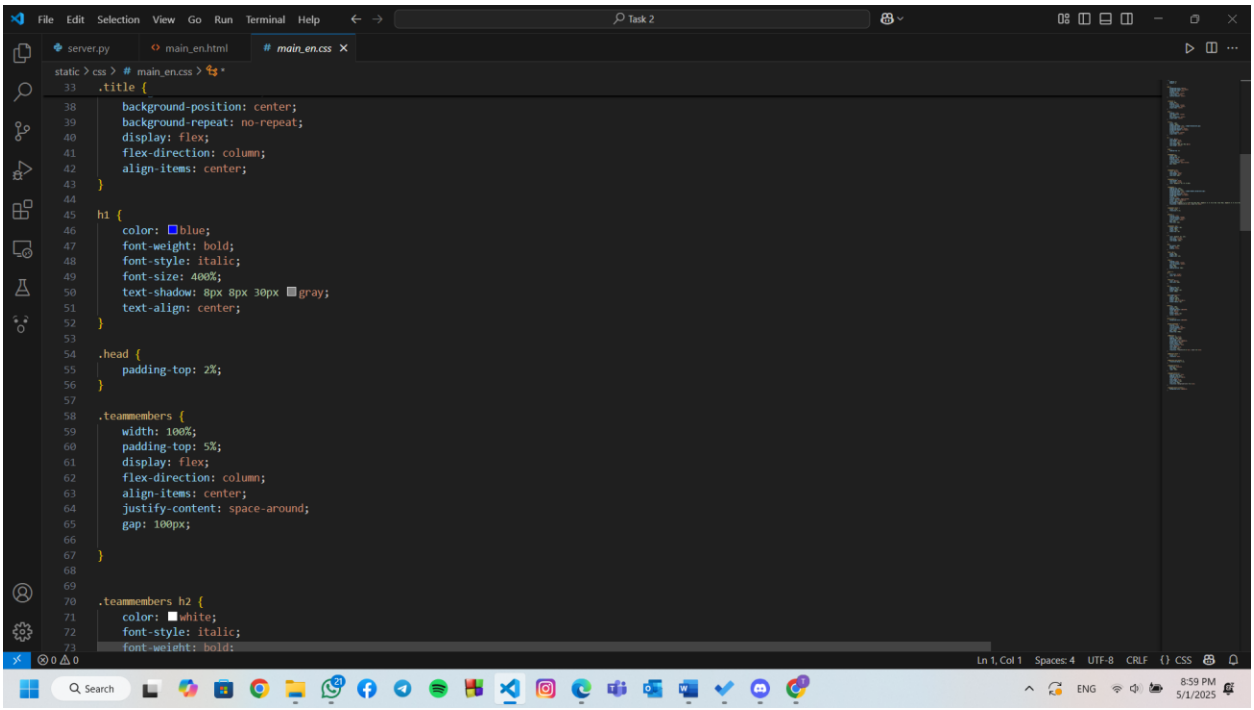
Figure 32 Main English Webpage HTML 4

Main English Webpage CSS




```
* {
    padding: 0;
    margin: 0;
}

body {
    background-color: white;
    background-repeat: no-repeat;
    background-size: cover;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
}


.container {
    width: 80%;
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
}

.main {
    display: flex;
    flex-direction: column;
    gap: 50px;
    justify-content: center;
    align-items: center;
}

.title {
    width: 100%;
    height: 100vh;
    background-image: url(../images/background.jpg);
    background-size: cover;
```

Figure 33 Main English Webpage CSS 1




```
.title {
    background-position: center;
    background-repeat: no-repeat;
    display: flex;
    flex-direction: column;
    align-items: center;
}

h1 {
    color: blue;
    font-weight: bold;
    font-style: italic;
    font-size: 400%;
    text-shadow: 8px 8px 30px gray;
    text-align: center;
}

.head {
    padding-top: 2%;
}

.teammembers {
    width: 100%;
    padding-top: 5%;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: space-around;
    gap: 100px;

}


.teammembers h2 {
    color: white;
    font-style: italic;
    font-weight: bold;
```

Figure 34 Main English Webpage CSS 2

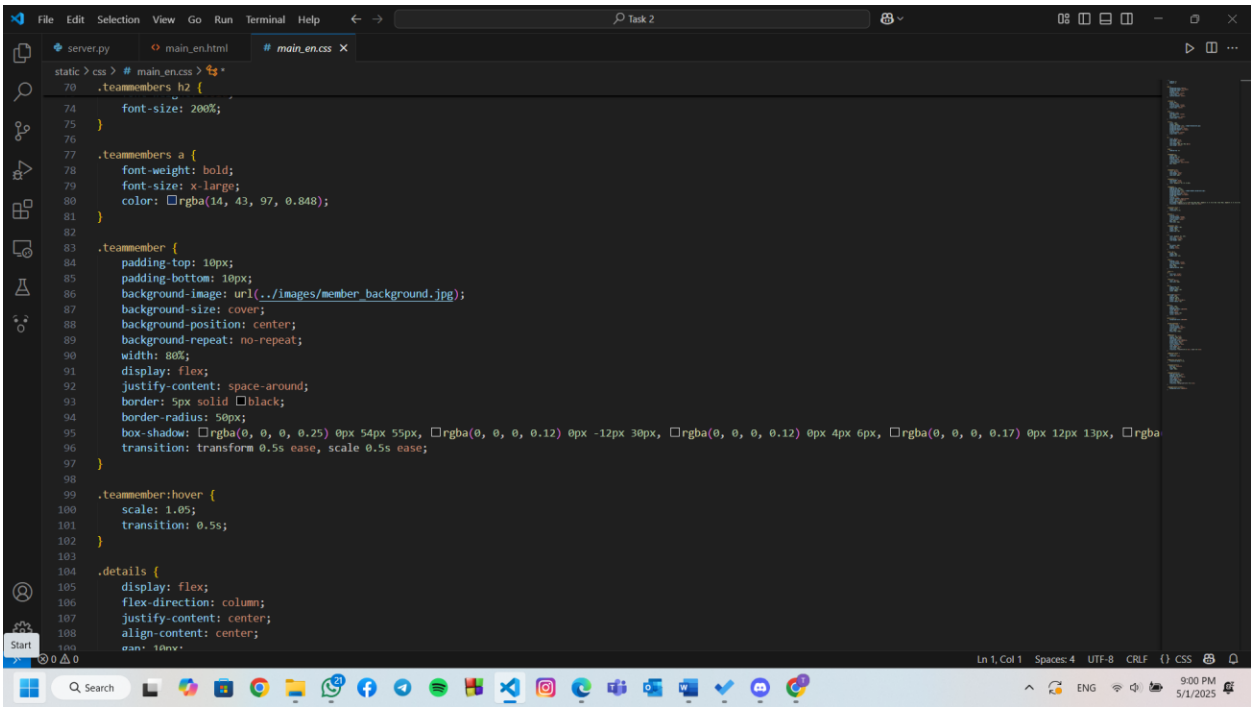


Figure 35 Main English Webpage CSS 3

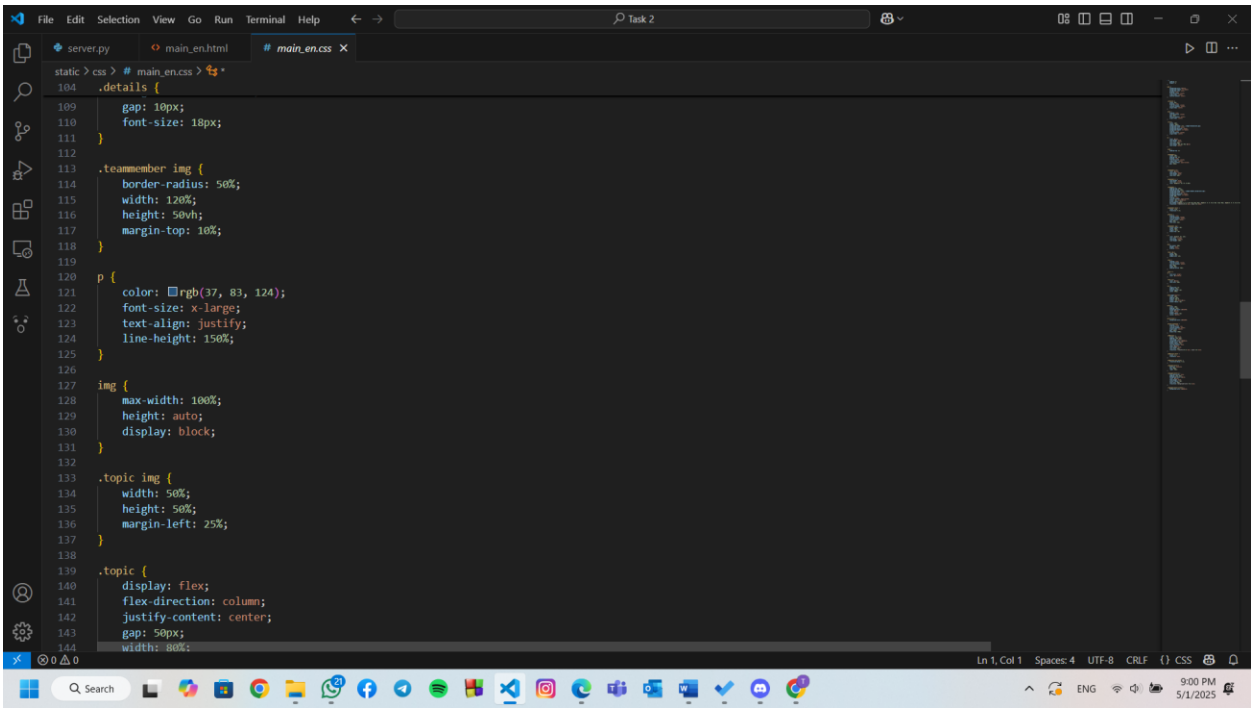
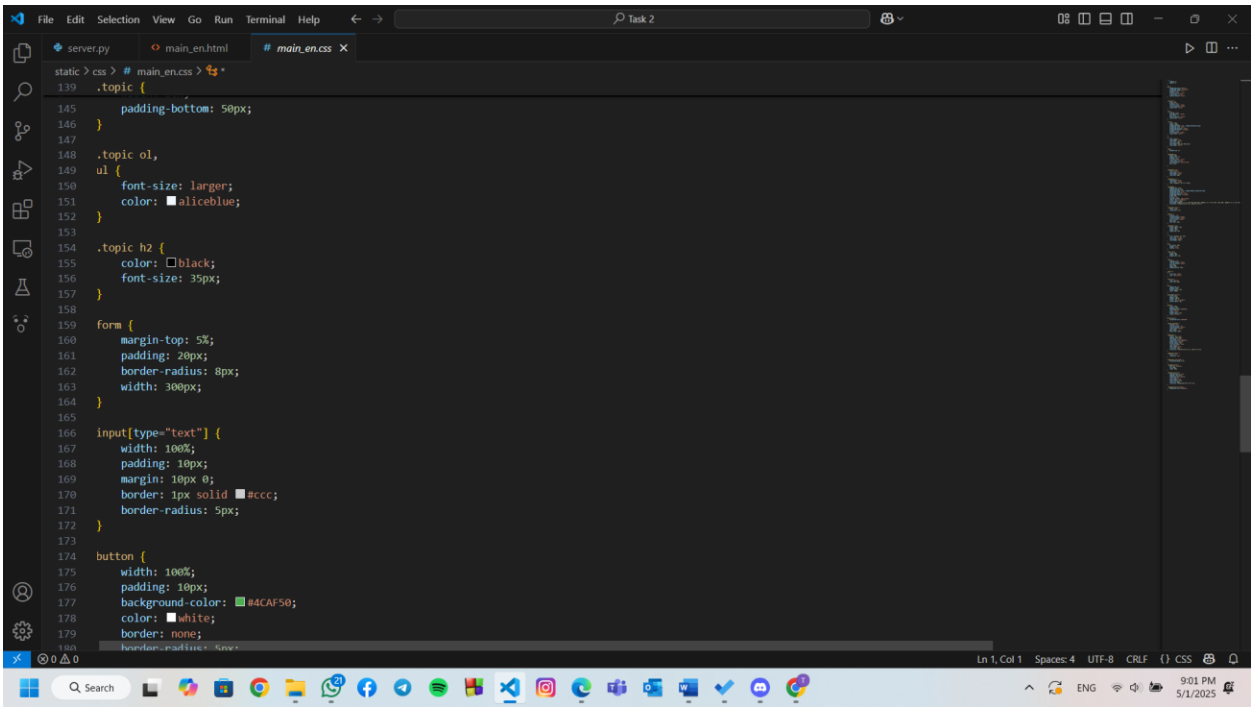
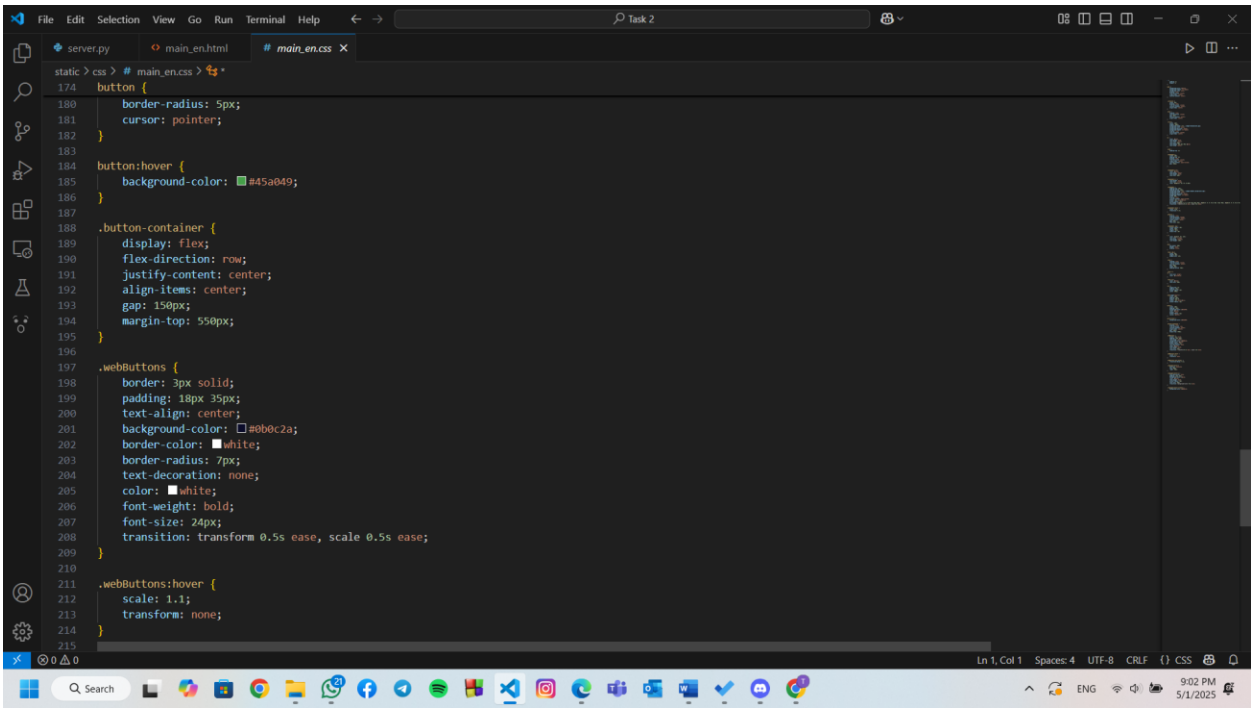


Figure 36 Main English Webpage CSS 4



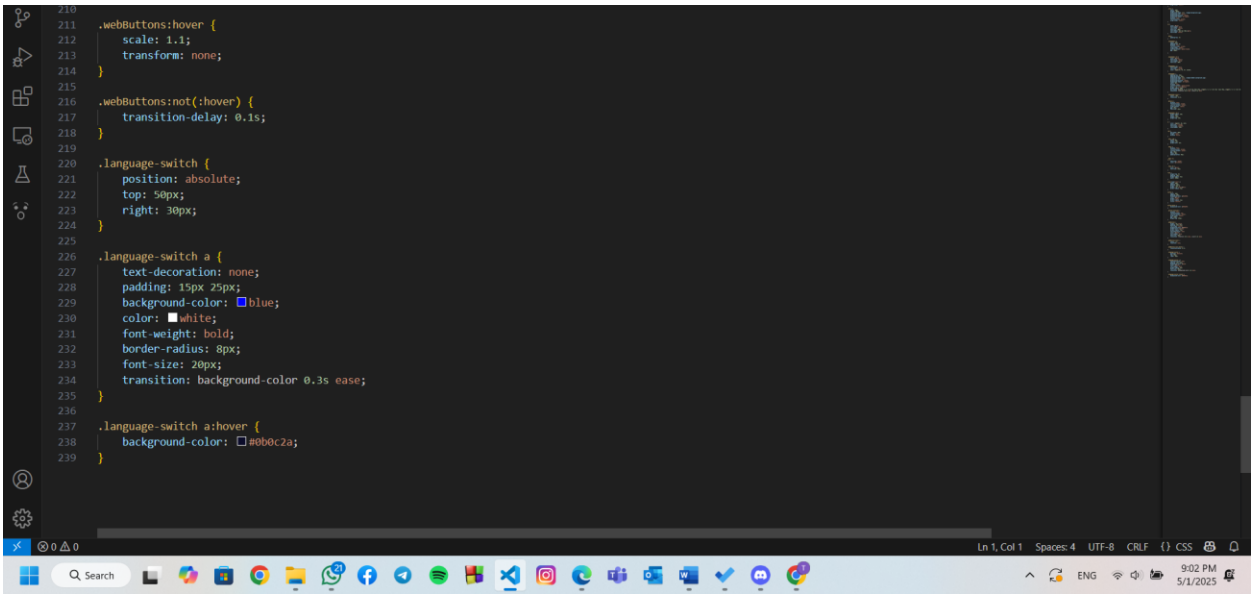
```
.topic {
    padding-bottom: 50px;
}

.topic ol,
ul {
    font-size: larger;
    color: aliceblue;
}

.topic h2 {
    color: black;
    font-size: 35px;
}

form {
    margin-top: 5%;
    padding: 20px;
    border-radius: 8px;
    width: 300px;
}

input[type="text"] {
    width: 100%;
    padding: 10px;
    margin: 10px 0;
    border: 1px solid #ccc;
    border-radius: 5px;
}

button {
    width: 100%;
    padding: 10px;
    background-color: #4CAF50;
    color: white;
    border: none;
```

Figure 37 Main English Webpage CSS 5



```
button {
    border-radius: 5px;
    cursor: pointer;
}

button:hover {
    background-color: #45a049;
}

.button-container {
    display: flex;
    flex-direction: row;
    justify-content: center;
    align-items: center;
    gap: 150px;
    margin-top: 550px;
}

.webButtons {
    border: 3px solid;
    padding: 18px 35px;
    text-align: center;
    background-color: #0b0c2a;
    border-color: white;
    border-radius: 7px;
    text-decoration: none;
    color: white;
    font-weight: bold;
    font-size: 24px;
    transition: transform 0.5s ease, scale 0.5s ease;
}

.webButtons:hover {
    scale: 1.1;
    transform: none;
}
```

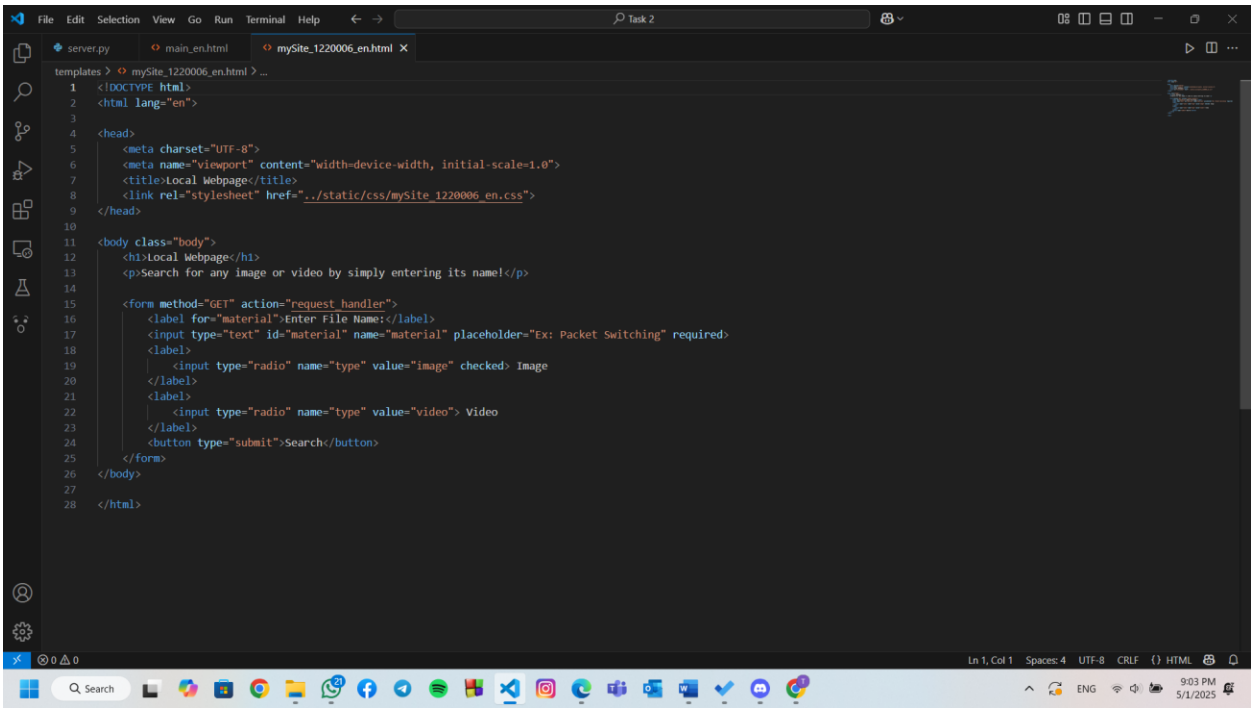
Figure 38 Main English Webpage CSS 6



```
.webButtons:hover {
    scale: 1.1;
    transform: none;
}

.webButtons:not(:hover) {
    transition-delay: 0.1s;
}

.language-switch {
    position: absolute;
    top: 50px;
    right: 30px;
}

.language-switch a {
    text-decoration: none;
    padding: 15px 25px;
    background-color: blue;
    color: white;
    font-weight: bold;
    border-radius: 8px;
    font-size: 20px;
    transition: background-color 0.3s ease;
}

.language-switch a:hover {
    background-color: #0b0c2a;
}
```

Figure 39 Main English Webpage CSS 7

English Local Webpage HTML



```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Local Webpage</title>
    <link rel="stylesheet" href="../static/css/mySite_1220006_en.css">
</head>

<body class="body">
    <h1>Local Webpage</h1>
    <p>Search for any image or video by simply entering its name!</p>

    <form method="GET" action="request_handler">
        <label for="material">Enter File Name:</label>
        <input type="text" id="material" name="material" placeholder="Ex: Packet Switching" required>
        <label>
            <input type="radio" name="type" value="image" checked> Image
        </label>
        <label>
            <input type="radio" name="type" value="video"> Video
        </label>
        <button type="submit">Search</button>
    </form>
</body>

</html>
```

Figure 40 English Local Webpage HTML

English Local Webpage CSS

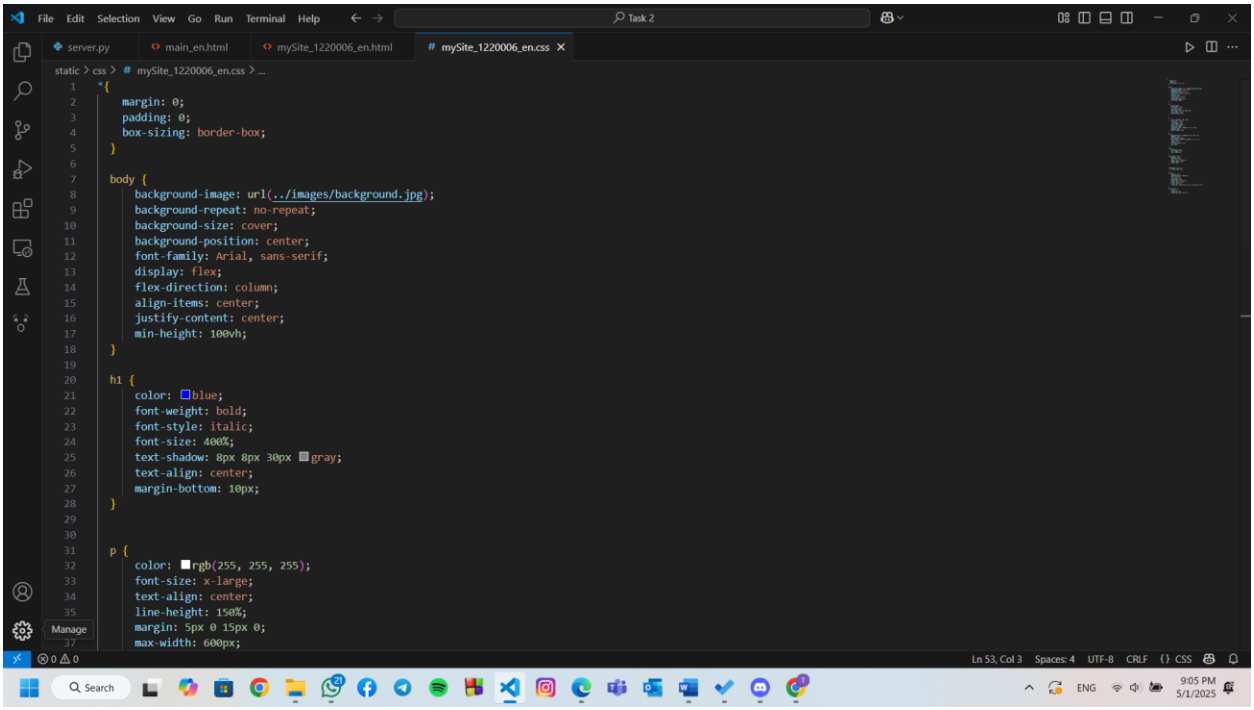


Figure 41 English Local Webpage CSS 1

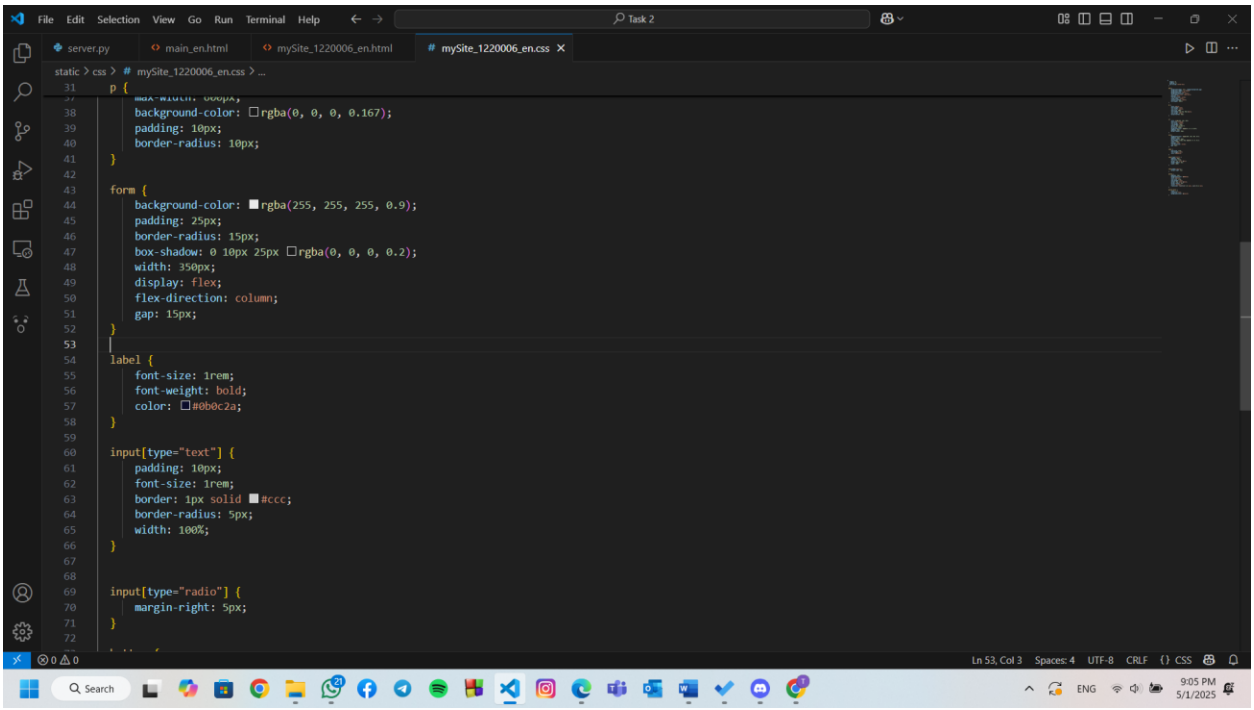


Figure 42 English Local Webpage CSS 2

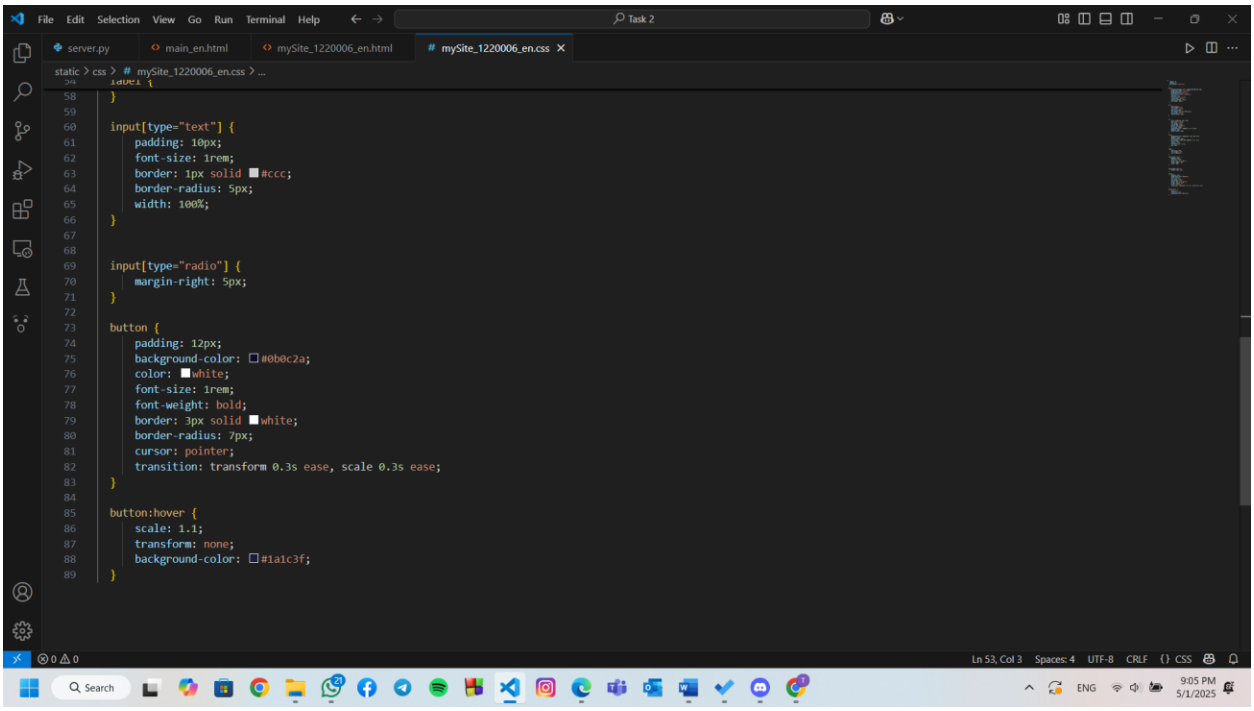


Figure 43 English Local Webpage CSS 3

The HTML and CSS structures illustrated in the figures above represent the design and functionality of all the English pages. Each corresponding Arabic page mirrors the same layout, styling, and features, with the content accurately translated into Arabic to ensure consistency across both language versions.

Python Server Code

server.py > ...

```python
#Taleed Hamadneh 1220006
#Qasim Batrawi 1220204
from socket import *
import os
serverPort = 9906
#TCP socket
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
# listen to the HTTP request on port 9906
serverSocket.listen(30) # up to 30 clients
print('Web Server is listening on port 9906 ...')

def send_response(response_code, response_type):
    status_line = ""
    if int(response_code) == 200:
        status_line = "HTTP/1.1 200 OK\r\n"
    elif int(response_code) == 404:
        status_line = "HTTP/1.1 404 Not Found\r\n"
    elif int(response_code) == 307:
        status_line = "HTTP/1.1 307 Temporary Redirect\r\n"
    else:
        status_line = f"HTTP/1.1 {response_code} Unknown\r\n"

    connectionSocket.send(status_line.encode())
    res = "Content-Type: " + response_type + "; charset=utf-8\r\n"
    connectionSocket.send(res.encode())
    connectionSocket.send("\r\n".encode())

    print(f"Response Sent: {status_line.strip()} | Client IP: {client_ip} | Client Port: {client_port} | Server Port: {serverPort}")
```

Figure 44 Python Server Code 1

As shown in the segment of the code above, a TCP socket is created using Python's socket module and is bound to port 9906. The server listens for incoming HTTP requests, allowing up to 30 clients at once. The function send_response is designed to handle the initial part of the HTTP response. Depending on the status code passed (200, 404, or 307), it sends the appropriate HTTP response line to the client. It then adds the Content-Type header based on the specified resource type (such as text/html or image/jpeg) and sends a blank line to indicate the end of the header section. This ensures proper formatting of the HTTP response so that browsers can correctly interpret the server's replies.

```
#when the requested file is unavailable
def Error(ip,port):
        send_response(404,'text/html')
        error = ('<!DOCTYPE html>'
                 '<html lang="en">'
                 '<style>'
                 '*{text-align: center;padding:10px;}'
                 'h1{font-size:50px;}'
                 'p{font-size:20px;}'
                 '</style>'
                 '<head>'
                 '<meta charset="UTF-8">'
                 '<meta name="viewport" content="width=device-width, initial-scale=1.0">'
                 '<title>Error 404</title>'
                 '</head>'
                 '<body>'
                 '<h1 style="color: red;">The file is not found!</h1>'
                 '<p>IP Address: ' + str(ip) + ', Port Number: ' + str(port) + '</p>'
                 '</body></html>'
        )
        connectionSocket.send(error.encode())
```

Figure 45 Python Server Code 2

As shown in the code segment above, the Error function is called when the requested file is not found on the server. In this case, the server sends a custom error webpage as a response. The page includes the title **"Error 404"** on the browser tab and shows the message **"The file is not found!"** in red on the page itself. It also displays the IP address and port number of the client who made the request. The HTML content of this error page is written directly inside the Python code as a string and sent to the client through the socket connection.

As shown in the code segment below, this part of the server is responsible for continuously listening and handling incoming client requests. It begins by organizing the available server files into categories such as HTML, CSS, images, and videos, based on their respective folders. Within an infinite loop, the server accepts a connection from a client, capturing their IP address and port number. It then receives the HTTP request sent by the client and attempts to extract the requested file name from the first line of the request message. This information is

also printed to the console for monitoring purposes. In case the request is malformed or an unexpected error occurs during this process, the server handles it gracefully by printing "BAD REQUEST" and waiting for the next client, ensuring that the server remains operational without interruption.

```
while True: #ALWAYS ON SERVER
    # Show all the files (that are available in the server) and separate them based on their types
    files = os.listdir('.')
    temp_files = []
    for f in files:
        if os.path.isfile(f):
            temp_files.append(f)
    files = temp_files

    html = os.listdir('./html')
    temp_html = []
    for f in html:
        if os.path.isfile(os.path.join('./html', f)):
            temp_html.append(f)
    html = temp_html

    css = os.listdir('./css')
    temp_css = []
    for f in css:
        if os.path.isfile(os.path.join('./css', f)):
            temp_css.append(f)
    css = temp_css

    images = os.listdir('./imgs')
    normal_images = []
    for f in images:
        if os.path.isfile(os.path.join('./imgs', f)):
            normal_images.append(f)
    images = normal_images

    videos = os.listdir('./videos')
```

Figure 46 Python Server Code 3

```
    #connection is established
    connectionSocket, addr = serverSocket.accept()
    client_ip = addr[0]
    client_port = addr[1]
    print('Got connection from', "IP: " + client_ip + ", Port: " + str(client_port) + ", Server Port: 9906")

    sentence = connectionSocket.recv(4096).decode() #4096 is the maximum number of bytes to receive from the socket at once

    try:
        #get the request line from the request (This line has the requested object)
        sen = sentence.split('\n')[0]
        request = sen.split(' ')[1]
        print("The Request is : " + request) # print http request
        #print(sentence)
    except :
        print("BAD REQUEST")
        continue
```

Figure 47 Python Server Code 4

```
    # Main_English request
    if request == "/" or request == "/index.html" or request == "/main_en.html" or request == "/en":

        if 'main_en.html' in html: # check if the file exists
            with open('./html/main_en.html','r', encoding='utf-8') as Main_English: # encoding is needed to show arabic words
                Main_English = Main_English.read()
            send_response(200,'text/html') # 200 -> OK , the content is html
            connectionSocket.send(Main_English.encode()) # send the file to the client through the socket (encoding from strings to byte)
        else:
            Error(client_ip, client_port) # 404 ERROR

    # Main_Arabic request
    elif request == "/ar" or request == "/main_ar.html":

        if 'main_ar.html' in html: # check if the file exists
            with open('./html/main_ar.html','rb') as Main_Arabic:
                Main_Arabic = Main_Arabic.read()
            send_response(200,'text/html')
            connectionSocket.send(Main_Arabic)
        else:
            Error(client_ip, client_port)
```

Figure 48 Python Server Code 5

This part of the code handles HTTP requests for the main English and Arabic pages. As shown in the segment above, if the request matches common paths like /, /index.html, /main_en.html, or /en, the server interprets it as a request for the English main page. It first checks if the main_en.html file exists. If so, it opens the file using UTF-8 encoding (to correctly display Arabic content if present), reads its content, and sends it back to the client with an HTTP 200 OK response and a content type of HTML. If the file is not found, the server responds with a 404 error using the custom Error function. Similarly, if the request is for the Arabic version (e.g., /ar or /main_ar.html), the server attempts to open and return main_ar.html.

The file is read in binary mode and sent directly to the client, also with a 200 OK response. If this file does not exist either, the same error handling procedure is applied.

```
    # if this statement (/request_handler?material) is in the url, then the user submited an image/video name in the html form
    # request: /html/request_handler?material=example&type=image
    elif request.startswith("/html/request_handler?material"):
        input = request.split('=')[1].split('&')[0] # to get the input from the user
        type = request.split('=')[2] # to get the type image/video

        if type == 'image':
            object1 = input + '.jpg' # to search for the image with jpg extension
            object2 = input + '.png' # to search for the image with png extension

            if object1 in images:
                path = './imgs/' + object1
                with open(path,'rb') as image:
                    image = image.read()
                send_response(200,'image/jpg')
                connectionSocket.send(image)
            elif object2 in images:
                path = './imgs/' + object2
                with open(path,'rb') as image:
                    image = image.read()
                send_response(200,'image/png')
                connectionSocket.send(image)
            else:
                connectionSocket.send("HTTP/1.1 307 Temporary Redirect\r\n".encode())
                connectionSocket.send('Content-Type: text/html; charset=utf-8\r\n'.encode())
                type = type.replace(" ", "+")
                location = "Location:http://www.google.com/search?q="+input+"&udm=2\r\n"
                connectionSocket.send(location.encode())
                connectionSocket.send('\r\n'.encode())
                print(f"Response Sent: HTTP/1.1 307 Temporary Redirect | Client IP: {client_ip} | Client Port: {client_port} | Server Port: {serverPort}\n")
                print("Redirect to Google\r\n")
```

Figure 49 Python Server Code 6

This part of the code handles user requests for images or videos that are submitted through the form on the Local webpage. When the URL contains /request_handler?material, the server understands that the user is asking for a specific file. The URL is typically in the format /request_handler?material=FileName&type=image (or type=video). The server extracts the file name and its type by splitting the request at the (=) signs and (&). If the requested type is an image, the server checks whether the image file (with a .jpg or .png extension) exists in the images folder. If it finds the image, it sends it back with an HTTP 200 OK response and the appropriate content type (like image/png or image/jpg). However, if the image is not found on the server, the server sends a 307 Temporary Redirect response, guiding the browser to automatically search for the image on Google Images. This is done by

sending a redirect location like Location: http://www.google.com/search?q=FileName&udm=2, where FileName is the user's input, and udm=2 is used to display image results. This ensures that even if the file isn't available on the server, the user is still directed to helpful results.

```
159         else:
160             object = input + '.mp4'
161             if object in videos:
162                 path = './videos/' + object
163                 with open(path,'rb') as video:
164                     video = video.read()
165                 send_response(200, 'video/mp4')
166                 connectionSocket.send(video)
167 
168             else:
169                 connectionSocket.send("HTTP/1.1 307 Temporary Redirect\r\n".encode())
170                 connectionSocket.send('Content-Type: text/html; charset=utf-8\r\n'.encode())
171                 location = "Location:https://www.youtube.com/results?search_query=" + input +"\r\n"
172                 connectionSocket.send(location.encode())
173                 connectionSocket.send('\r\n'.encode())
174                 print(f"Response Sent: HTTP/1.1 307 Temporary Redirect | Client IP: {client_ip} | Client Port: {client_port} | Server Port: {serverPort}")
175                 print("Redirect to Youtube\r\n")
176 
```

Figure 50 Python Server Code 7

This part of the code handles video requests. When a user asks for a video, the server checks if the video exists in the videos folder with a .mp4 extension. If it finds the video, it sends the video back to the user with a "200 OK" response.

If the video isn't found, the server sends a "307 Temporary Redirect" response, telling the browser to go to YouTube and search for the video. The server creates a YouTube search link based on what the user entered and sends it to the browser, redirecting the user to YouTube.

```
    # other html file request
    # request: /html/mySite_1220006_ar.html
    elif request.startswith("/html/"):
        html_file_name = request.split('/')[2] # get the html file name (for example: mySite_1220006_ar.html)
        if html_file_name in html:
            path = "./html/" + html_file_name
            with open(path,'rb') as Local_Webpage_Arabic:
                Local_Webpage_Arabic = Local_Webpage_Arabic.read()
            send_response(200,'text/html')
            connectionSocket.send(Local_Webpage_Arabic)
        else:
            Error(client_ip, client_port)

    # css file request
    # request: /css/example.css
    elif request.startswith("/css/"):
        css_file_name = request.split('/')[2] # get the image name (for example: main.css)
        if css_file_name in css: # check if the file exists
            path = './css/' + css_file_name # path of the css file
            with open(path,'r') as cssFile:
                cssFile = cssFile.read()
            send_response(200,'text/css') # 200 -> OK , the content is css
            connectionSocket.send(cssFile.encode()) # send the file to the client through the socket (encoding from strings to byte)
        else:
            Error(client_ip, client_port)
```

Figure 51 Python Server Code 8

This part of the code handles requests for other HTML and CSS files. For HTML file requests, the server checks if the requested file exists in the html directory. If the file is found, it is read and sent back to the client with a "200 OK" response. If the file is not found, a "404 Not Found" error page is sent instead. Similarly, for CSS file requests, the server checks if the requested CSS file is available in the css directory. If the file exists, it is read and sent to the client with a "200 OK" response and a content type of "text/css". If the CSS file is not found, the server responds with a "404 Not Found" error page.

```python
    # images request
    # request: /imgs/example.png
    elif request.startswith("/imgs/"):
        image_file_name = request.split('/')[2] # get the image name (for example: background.png)
        if image_file_name in images:
            path = './imgs/' + image_file_name # path of the image
            response_type = 'image/' + path.split('.')[-1] # path.split('.')[-1] returns the the extension (for example: png)
            with open(path,'rb') as image:
                image = image.read()
            send_response(200, response_type)
            connectionSocket.send(image) # send the image to the client through the socket
        else:
            Error(client_ip, client_port)

    # videos requests (similar to image request)
    # request: request: /videos/example.mp4
    elif request.startswith("/videos/"):
        video_file_name = request.split('/')[2] # get the video file name
        if video_file_name in videos:
            type = 'video/' + video_file_name.split('.')[-1] # get the video extension
            path = './videos/' + video_file_name
            with open(path, 'rb') as video:
                video = video.read()
            send_response(200, type)
            connectionSocket.send(video)
        else:
            Error(client_ip, client_port)
```

Figure 52Python Server Code 9

This section of the code handles requests for images and videos. For image requests, it checks if the requested image exists in the images directory. If the file is found, the server determines the appropriate response type based on the image file's extension (e.g., PNG or JPEG). The image is then read from the server and sent back to the client with a "200 OK" response. If the image is not found, an error page is returned instead. Similarly, for video requests, the server checks if the requested video file exists in the videos directory. If the file is available, it reads and sends the video to the client with the corresponding content type based on the video file's extension. If the video is not found, an error page is returned.

```
    #if the client made any request that does not exist
    else:
        try:
            object = request.split('/')[1]
            if object in files or object in html or object in css or object in images or object in videos:
                type = object.split('.')[-1]
                if object in html:
                    object = './html/' + object
                elif object in css:
                    object = './css/' + object
                elif object in images:
                    object = './imgs/' + object
                elif object in videos:
                    object = './videos/' + object

                if type == 'html':
                    send_response(200, 'text/html')
                    with open(object, 'r') as file:
                        file = file.read()
                    connectionSocket.send(file.encode())
                elif type == 'css':
                    send_response(200, 'text/css')
                    with open(object, 'rb') as file:
                        file = file.read()
                    connectionSocket.send(file)
                elif type == 'jpg':
                    send_response(200, 'image/jpg')
                    with open(object, 'rb') as file:
                        file = file.read()
                    connectionSocket.send(file)
                elif type == 'png':
                    send_response(200, 'image/png')
                    with open(object, 'rb') as file:
                        file = file.read()
                    connectionSocket.send(file)
```

Figure 53 Python Server Code 10

```
                    connectionSocket.send(file)
                elif type == 'png':
                    send_response(200, 'image/png')
                    with open(object, 'rb') as file:
                        file = file.read()
                    connectionSocket.send(file)
                elif type == 'mp4':
                    send_response(200, 'video/mp4')
                    with open(object, 'rb') as file:
                        file = file.read()
                    connectionSocket.send(file)
                else:
                    send_response(200, 'text/html') # default
                    with open(object, 'rb') as file:
                        file = file.read()
                    connectionSocket.send(file)
            else:
                Error(client_ip,client_port)
        except IndexError:
            print('Bad Request')

    connectionSocket.close()
```

Figure 54 Python Server Code 11

This above of the code handles requests for any files that do not match the predefined categories of HTML, CSS, images, or videos. When the client makes a request for a file, the server checks if the requested file exists in any of these categories (files, html, css, images, or videos). If the file is found, the server determines the appropriate file path based on the type of file and reads it from the appropriate directory (./html/ for HTML files, ./css/ for CSS files, etc.). Depending on the file type (HTML, CSS, image, or video), the server sends the file back with the correct content type (e.g., text/html for HTML, image/jpg for images, video/mp4 for videos). If the file is not found in any of these categories, the server sends an error page (Error 404) to the client. If there's an issue with the request, such as an invalid file path, it prints "Bad Request" in the console. Finally, the connection to the client is closed.

Server Functionality Testing

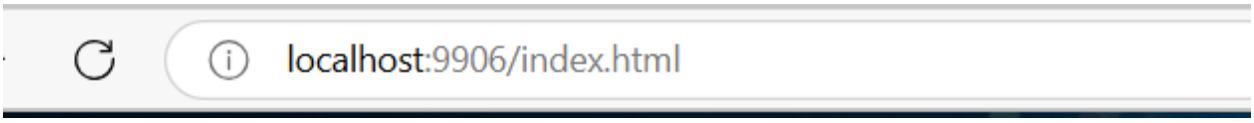


Figure 55 Browser Request for main English webpage

For the client (browser) request: '/', '/index.html', '/main_en.html' or '/en', The server receives a request for the main English webpage, and then receives a request for each object in this page: (main_en.css, Qasim.png, taleed.png, PacketSwitching.jpg, CircuitSwitching.jpg, background.jpg, member_background.jpg).

```
Web Server is listening on port 9906 ...
Got connection from IP: 127.0.0.1, Port: 63121, Server Port: 9906
The Request is : /index.html
GET /index.html HTTP/1.1
Host: localhost:9906
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
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/135.0.0.0 Safari/537.36 Edg/135.0.0.0
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: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63121 | Server Port: 9906
```

Figure 56 index.html request

```
Got connection from IP: 127.0.0.1, Port: 63129, Server Port: 9906
The Request is : /static/css/main_en.css
GET /static/css/main_en.css HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: text/css,*/*;q=0.1
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: style
Referer: http://localhost:9906/index.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63129 | Server Port: 9906
```

Figure 57 main_en.css request

```
Got connection from IP: 127.0.0.1, Port: 63130, Server Port: 9906
The Request is : /static/images/Qasim.png
GET /static/images/Qasim.png HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9906/index.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63130 | Server Port: 9906
Got connection from IP: 127.0.0.1, Port: 63131, Server Port: 9906
The Request is : /static/images/taleed.png
GET /static/images/taleed.png HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9906/index.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63131 | Server Port: 9906
```

Figure 58 members photos requests

```
Got connection from IP: 127.0.0.1, Port: 63134, Server Port: 9906
The Request is : /static/images/PacketSwitching.jpg
GET /static/images/PacketSwitching.jpg HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9906/index.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63134 | Server Port: 9906
Got connection from IP: 127.0.0.1, Port: 63135, Server Port: 9906
The Request is : /static/images/CircuitSwitching.jpg
GET /static/images/CircuitSwitching.jpg HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9906/index.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63135 | Server Port: 9906
```

Figure 59 Switching photos requests

```
Got connection from IP: 127.0.0.1, Port: 63136, Server Port: 9906
The Request is : /static/images/background.jpg
GET /static/images/background.jpg HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9906/static/css/main_en.css
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63136 | Server Port: 9906
Got connection from IP: 127.0.0.1, Port: 63137, Server Port: 9906
The Request is : /static/images/member_background.jpg
GET /static/images/member_background.jpg HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9906/static/css/main_en.css
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63137 | Server Port: 9906
```

Figure 60 Background photos requests

The server response for these requests is the html file for the main English webpage, and all objects in this page. The browser (client) displays these objects to the user (the Main English Webpage) as shown in the figure below:

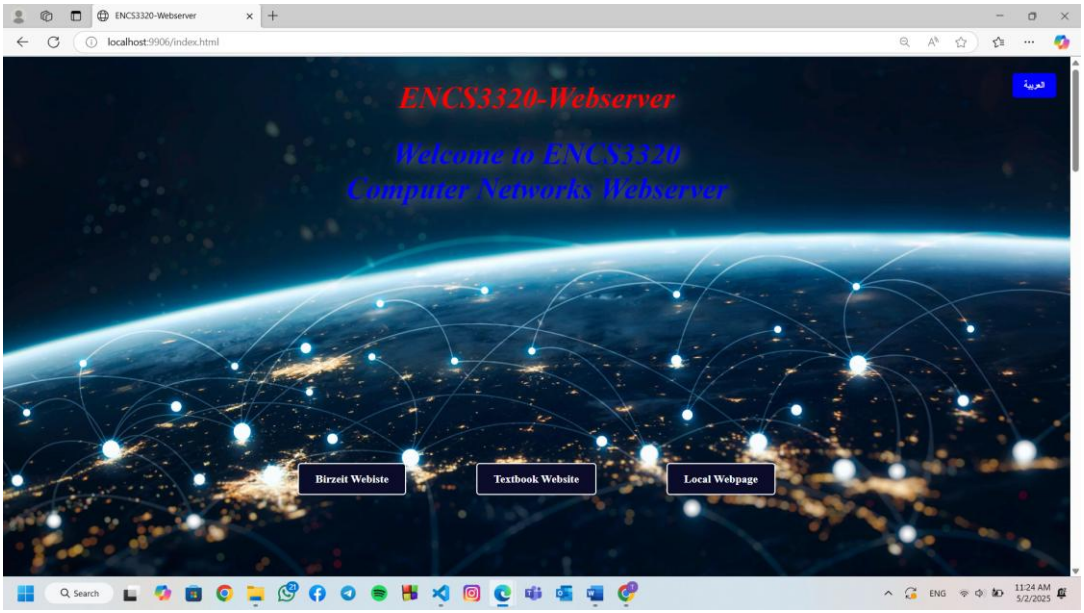


Figure 61 Response for the main English webpage

Requests to /ar or /main_ar.html work the same as English page requests, the server sends the Arabic main page, and the browser loads its linked files like CSS and images.

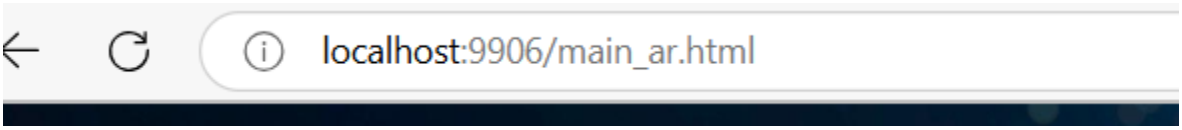


Figure 62 Browser Request for main Arabic webpage

```
Got connection from IP: 127.0.0.1, Port: 63226, Server Port: 9906
The Request is : /static/css/main_ar.css
GET /static/css/main_ar.css HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: text/css,*/*;q=0.1
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: style
Referer: http://localhost:9906/main_ar.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63226 | Server Port: 9906
Got connection from IP: 127.0.0.1, Port: 63230, Server Port: 9906
The Request is : /static/images/Qasim.png
GET /static/images/Qasim.png HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9906/main_ar.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63230 | Server Port: 9906
```

Figure 63 Part of browser requests

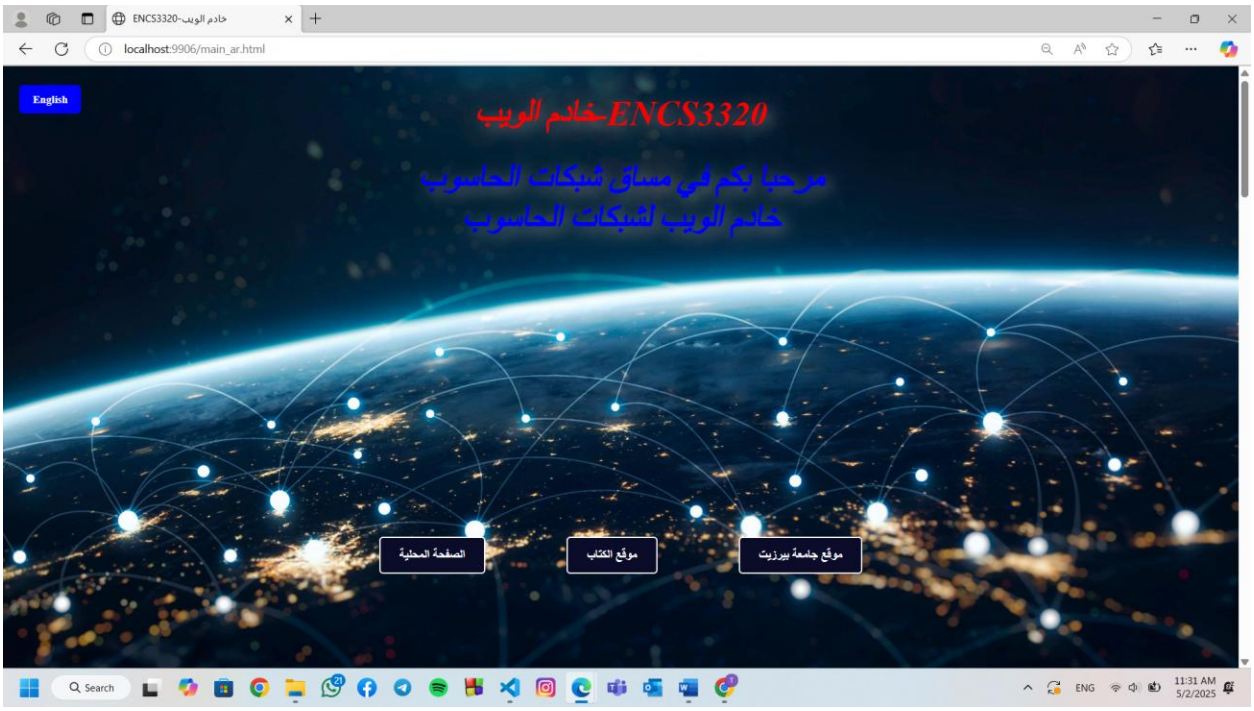


Figure 64 Response for the main Arabic webpage

When the client (browser) requests a file that doesn't exist on the server, the server responds with a 404 Not Found status and sends an error webpage (as shown in the figure below). The browser tab shows "Error 404," and the page displays "The File Not Found!" along with the client's IP address and port number.

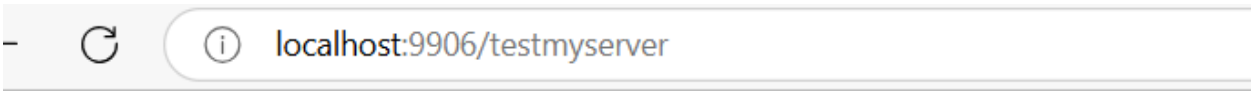


Figure 65 invalid URL request

```
Got connection from IP: 127.0.0.1, Port: 63241, Server Port: 9906
The Request is : /testmyserver
GET /testmyserver HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
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/135.0.0.0 Safari/537.36 Edg/135.0.0.0
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: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 404 Not Found | Client IP: 127.0.0.1 | Client Port: 63241 | Server Port: 9906
```

Figure 66 Request for invalid file

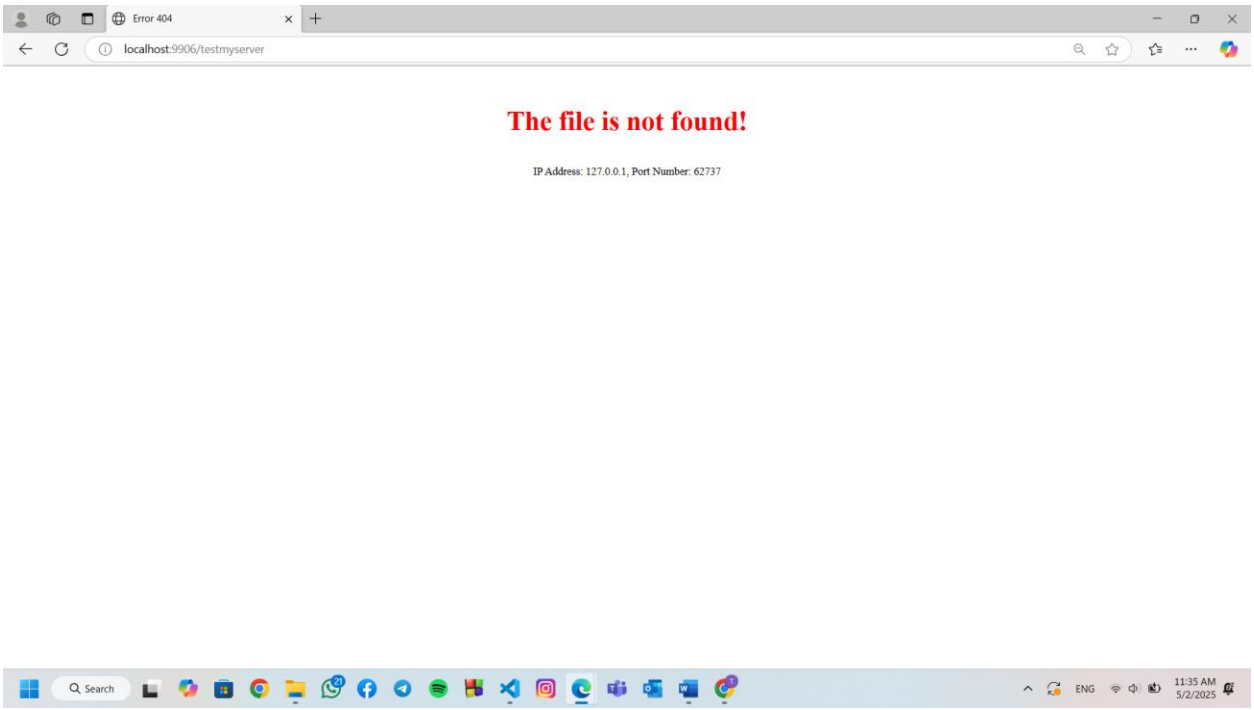


Figure 67 Response for invalid Request

When the client clicks on the **Birzeit** button, the browser opens the official Birzeit University website (https://www.birzeit.edu/) in a new tab. Similarly, clicking the **Textbook** button directs the user to the textbook's official website (https://gaia.cs.umass.edu/kurose_ross/index.php). Both links are handled using standard HTML hyperlinks, allowing users to access external resources easily.

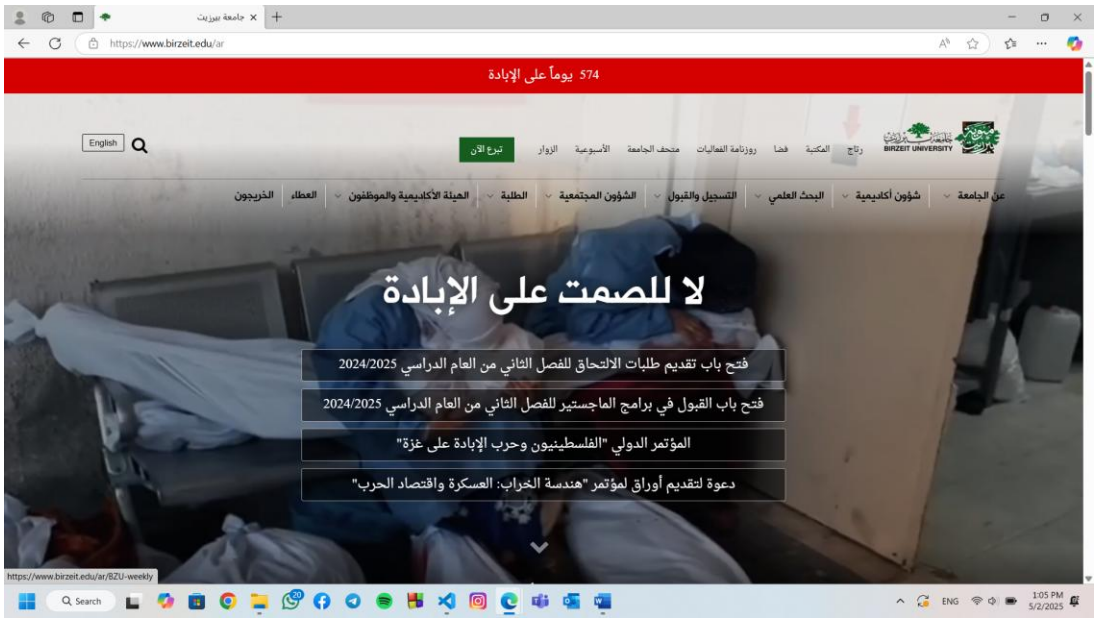


Figure 68 Birzeit Website

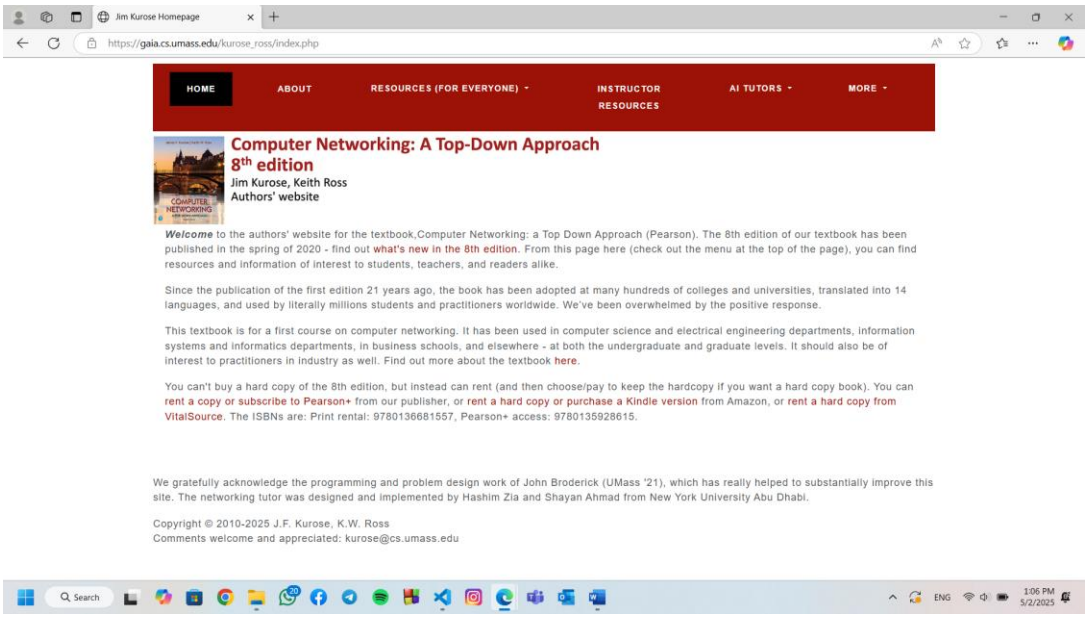


Figure 69 Textbook Website

When the user clicks on the **Local Webpage** button, the server receives requests for the HTML file of the page and all the associated objects it includes.

```
Got connection from IP: 127.0.0.1, Port: 63661, Server Port: 9906
BAD REQUEST
Got connection from IP: 127.0.0.1, Port: 63746, Server Port: 9906
The Request is : /html/mySite_1220006_en.html
GET /html/mySite_1220006_en.html HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
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/135.0.0.0 Safari/537.36 Edg/135.0.0.0
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: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:9906/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63746 | Server Port: 9906
```

Figure 70 mySite_1220006_en.html request

```
Got connection from IP: 127.0.0.1, Port: 63748, Server Port: 9906
The Request is : /static/css/mySite_1220006_en.css
GET /static/css/mySite_1220006_en.css HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: text/css,*/*;q=0.1
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: style
Referer: http://localhost:9906/html/mySite_1220006_en.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63748 | Server Port: 9906

Got connection from IP: 127.0.0.1, Port: 63751, Server Port: 9906
The Request is : /static/images/background.jpg
GET /static/images/background.jpg HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9906/static/css/mySite_1220006_en.css
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63751 | Server Port: 9906
```

Figure 71 Local Webpage objects' requests

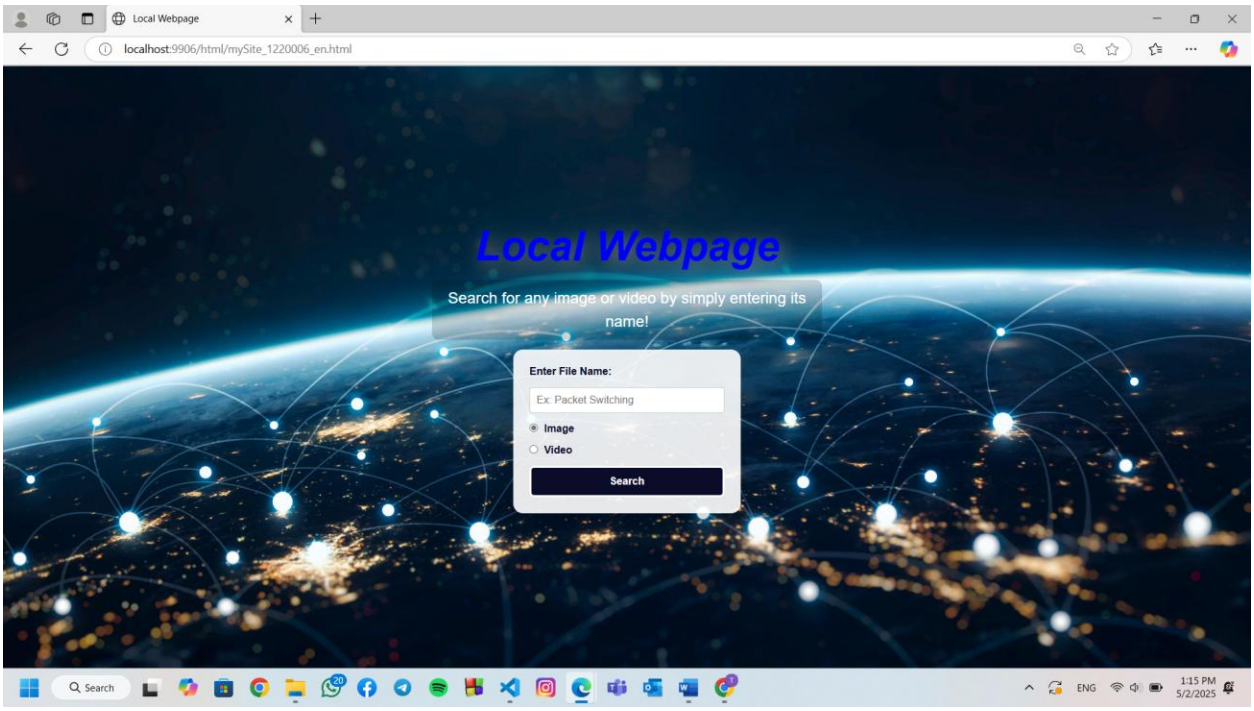


Figure 72 Response for the local Webpage

The user types the name of the image they want from the server and clicks submit. The server then processes the request and either returns the image if it exists or redirects the user to Google if it doesn't.

Figure 73 Search for image in local Webpage

```
Got connection from IP: 127.0.0.1, Port: 63790, Server Port: 9906
The Request is : /html/request_handler?material=CircuitSwitching&type=image
GET /html/request_handler?material=CircuitSwitching&type=image HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
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/135.0.0.0 Safari/537.36 Edg/135.0.0.0
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: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:9906/html/mySite_1220006_en.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63790 | Server Port: 9906
```

Figure 74 Request for image from local Webpage

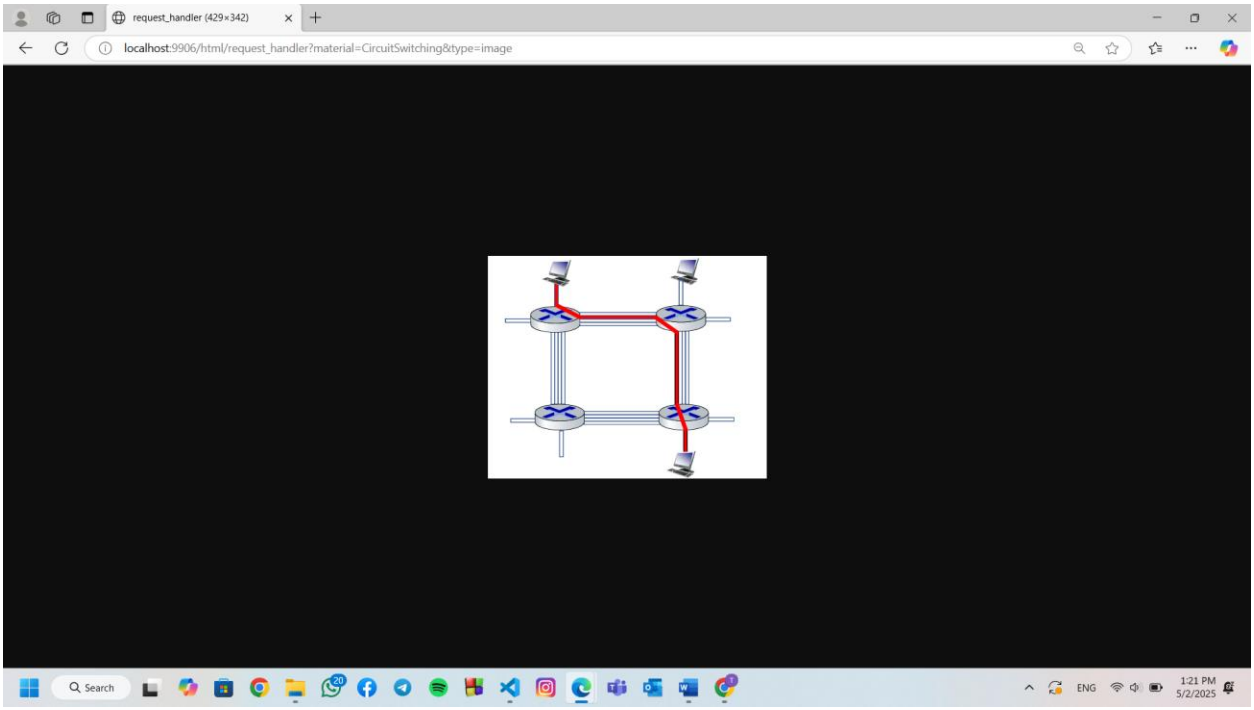


Figure 75 Response for the image

The Response of the server includes status code of (200 OK) as shown in request figure above with content type (image/jpg), and the requested image.

If the requested image, such as one named "Networks", is not found on the server, the server responds by redirecting the client to Google to view image search results related to "Networks". Along with this redirection, the server sends a response with the status code **307 Temporary Redirect**, indicating that the client should look elsewhere for the requested content.

Figure 76 Search for unavailable image

```
Got connection from IP: 127.0.0.1, Port: 63813, Server Port: 9906
The Request is : /html/request_handler?material=Networks&type=image
GET /html/request_handler?material=Networks&type=image HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
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/135.0.0.0 Safari/537.36 Edg/135.0.0.0
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: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:9906/html/mySite_1220006_en.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 307 Temporary Redirect | Client IP: 127.0.0.1 | Client Port: 63813 | Server Port: 9906

Redirect to Google
```

Figure 77 Request for unavailable image

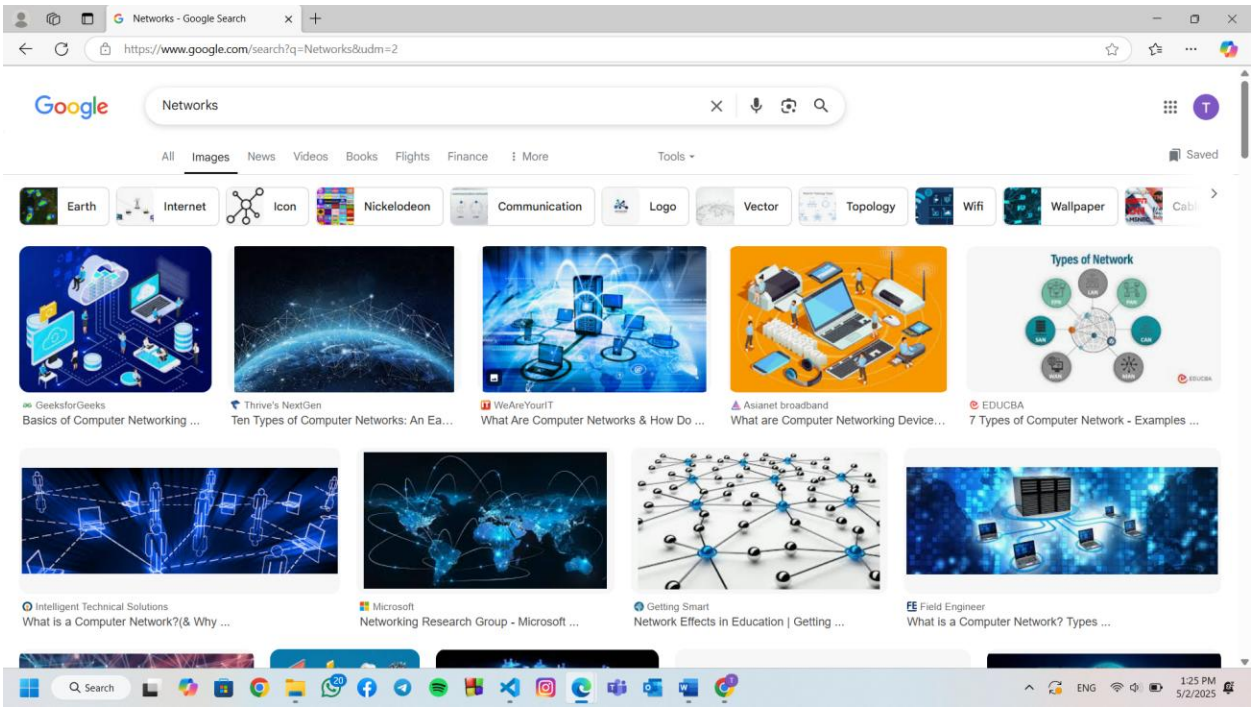


Figure 78 Response for unavailable image

Same for Videos:

Figure 79 Search for video in local Webpage

```
Got connection from IP: 127.0.0.1, Port: 63845, Server Port: 9906
The Request is : /html/request_handler?material=Switching&type=video
GET /html/request_handler?material=Switching&type=video HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua-platform: "Windows"
Accept-Encoding: identity;q=1, *;q=0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.0.0 Safari/537.36 Edg/135.0.0.0
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
sec-ch-ua-mobile: ?0
Accept: */*
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: video
Referer: http://localhost:9906/html/request_handler?material=Switching&type=video
Accept-Language: en-US,en;q=0.9,ar;q=0.8
Range: bytes=0-


Response Sent: HTTP/1.1 200 OK | Client IP: 127.0.0.1 | Client Port: 63845 | Server Port: 9906
```

Figure 80 Request for video from local Webpage

Figure 81 Response for the Video

If the requested video is not available on the server, the server redirects the client to YouTube to view search results related to that video.

Figure 82 Search for unavailable video

```
Got connection from IP: 127.0.0.1, Port: 63846, Server Port: 9906
BAD REQUEST
Got connection from IP: 127.0.0.1, Port: 63876, Server Port: 9906
The Request is : /html/request_handler?material=DNS&type=video
GET /html/request_handler?material=DNS&type=video HTTP/1.1
Host: localhost:9906
Connection: keep-alive
sec-ch-ua: "Microsoft Edge";v="135", "Not-A.Brand";v="8", "Chromium";v="135"
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/135.0.0.0 Safari/537.36 Edg/135.0.0.0
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: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:9906/html/mySite_1220006_en.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


Response Sent: HTTP/1.1 307 Temporary Redirect | Client IP: 127.0.0.1 | Client Port: 63876 | Server Port: 9906
Redirect to Youtube
```

Figure 83 Request for unavailable video

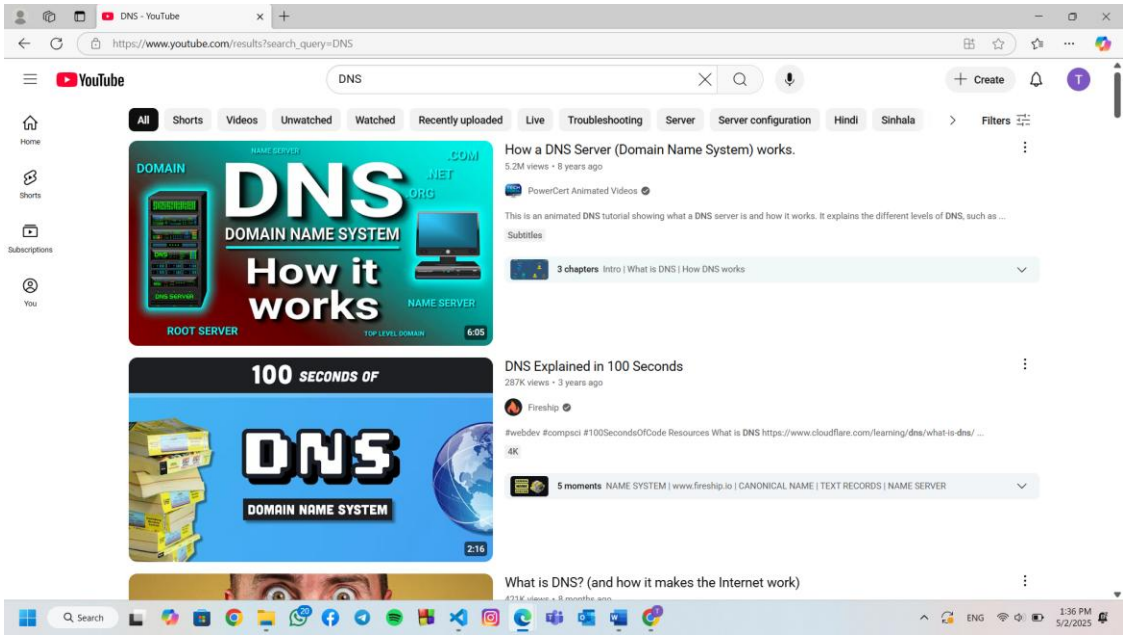


Figure 84 Response for unavailable video

When the client (browser) requests a file like *mySite_1220006_en.css*, the server checks if the file exists in the CSS directory. If it exists, it will respond with the file and a status code of **200 OK**; otherwise, it returns an error response.

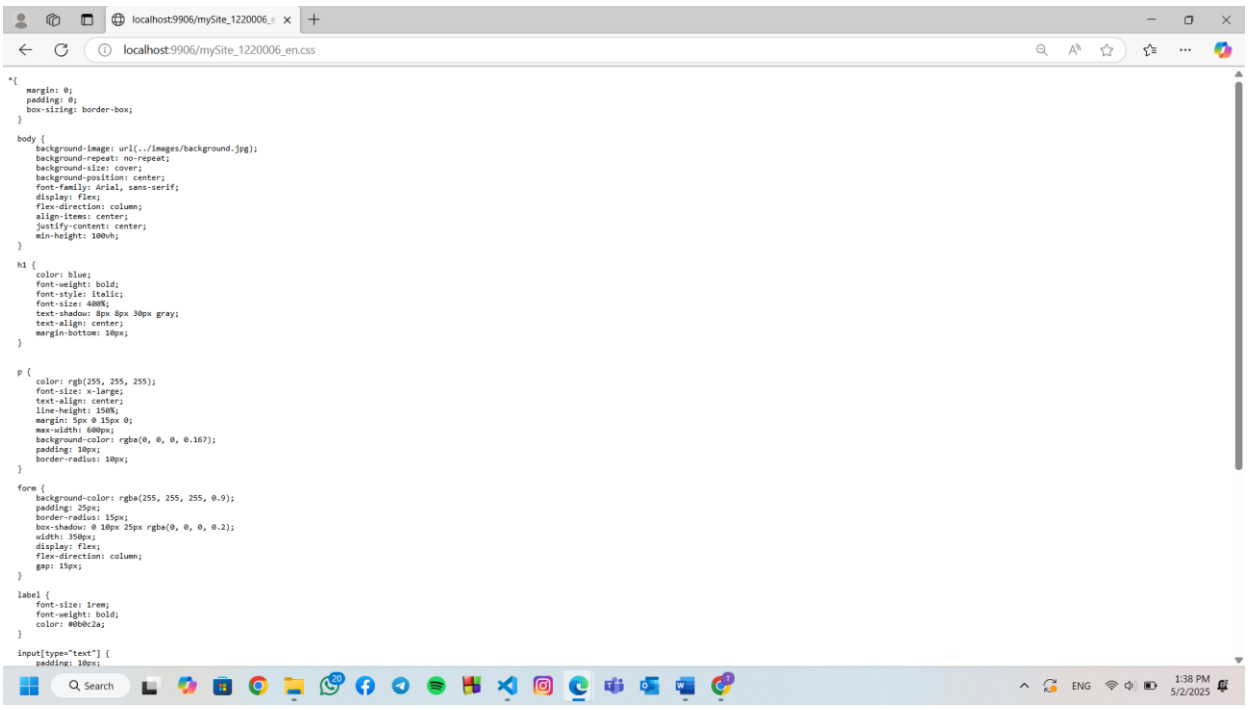


Figure 85 Response for css file request

Another device on the same local network requests a webpage from the server:

```
Got connection from IP: 192.168.1.101, Port: 59905, Server Port: 9906
The Request is : /
GET / HTTP/1.1
Host: 192.168.1.126:9906
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 18_3_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/18.3 Mobile/15E148 Safari/604.1
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Priority: u=0, i
Accept-Encoding: gzip, deflate
Connection: keep-alive


Response Sent: HTTP/1.1 200 OK | Client IP: 192.168.1.101 | Client Port: 59905 | Server Port: 9906

Got connection from IP: 192.168.1.101, Port: 59906, Server Port: 9906
The Request is : /static/css/main_en.css
GET /static/css/main_en.css HTTP/1.1
Host: 192.168.1.126:9906
Referer: http://192.168.1.126:9906/
Accept: text/css,*/*;q=0.1
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 18_3_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/18.3 Mobile/15E148 Safari/604.1
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Priority: u=1, i
Accept-Encoding: gzip, deflate
Connection: keep-alive


Response Sent: HTTP/1.1 200 OK | Client IP: 192.168.1.101 | Client Port: 59906 | Server Port: 9906
```

Figure 86 Request for a webpage from another device

Figure 87 Server Response to Iphone device request

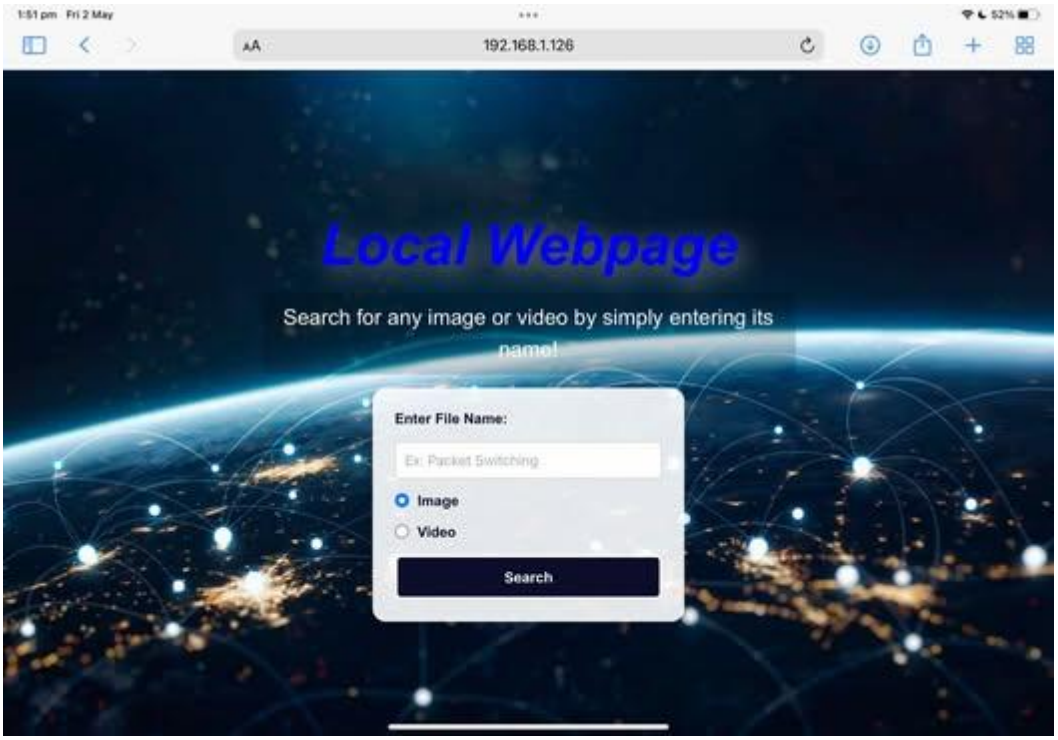


Figure 88 Server Response to IPAD device request

TASK 3: TCP/UDP Client-Server Game Using Socket Programming

In this task, we have implemented a hybrid network guessing game where the server use TCP for reliable control messages in the game setup and final results, and uses UDP for fast, real-time feedbacks during the gameplay. The game supports multiple players, each one trying to guess a randomly number generated by the server before the others.

Game Workflow

The server opens TCP and UDP sockets with port numbers 6000 and 6001, respectively. The server waits for at least two clients (players) to connect to the TCP socket by entering a unique name. Once two players connected, the game begins, and all players simultaneously enter their guess to the server via UDP. For each guess, the server will response with a feedback indicating whether the guessed number that the player entered is higher, lower, or equal to the target number. The first player that guesses the target number correctly wins the game. The server finally sends the final results for each client via TCP.

The total gameplay duration is 60 seconds and players must try to guess the number within this time or the game will end with no winner. Each player has a time of 10 seconds to send his guessed number, and numbers must be within the range defined by the server.

In case a player disconnects from the server during the gameplay, other players will be notified and asked if they want to continue guessing or end the game.

Players will play the game in rounds, each round is as described above, and after the results are announced, a new game begins.

Python Server Code

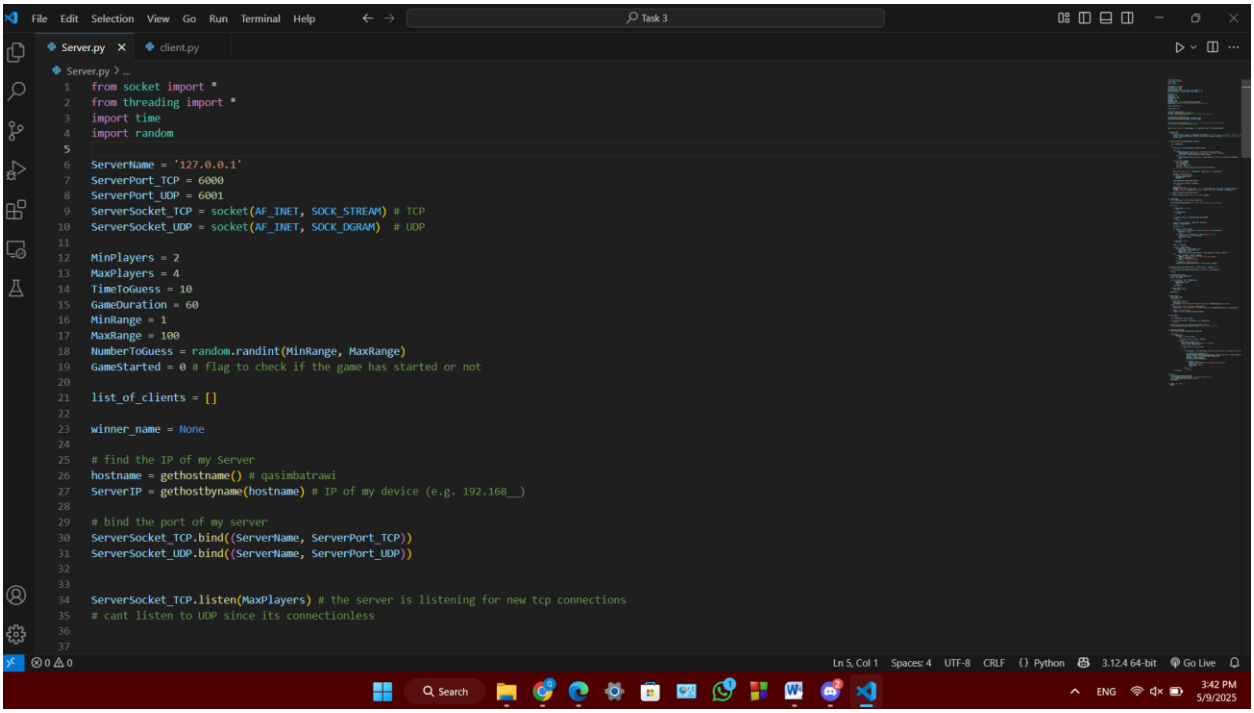


Figure 89 Server Code 1

The above figure shows the main parameters that we have used for our server, which includes the server name (localhost), the port number of TCP (6000) and UDP (6001), the server sockets TCP (for reliable connection) and UDP (for fast messaging), where we have used the socket() function to create them. The target number which is randomly generated using random() function. Each socket (TCP and UDP) is bound to a specific IP address using bind() method. The TCP socket then starts listening to any client connection, where it can listen to at max four players.

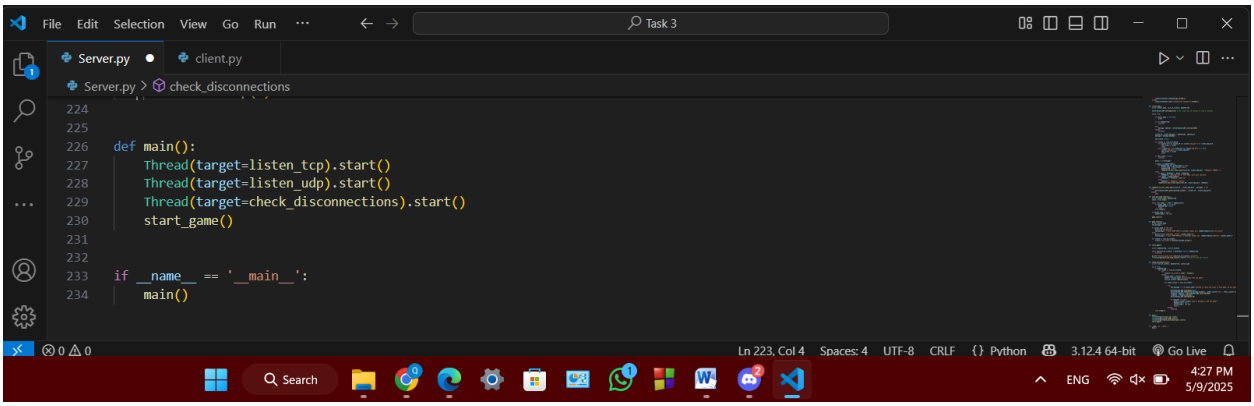


Figure 90 Server Code 2

Figure 88 shows the server's main function, where three threads are created: one for TCP connections, another for UDP fast messaging, and thread to check if some client has disconnected during the gameplay. The main thread will enter strat_game() function. Each will be discussed later.

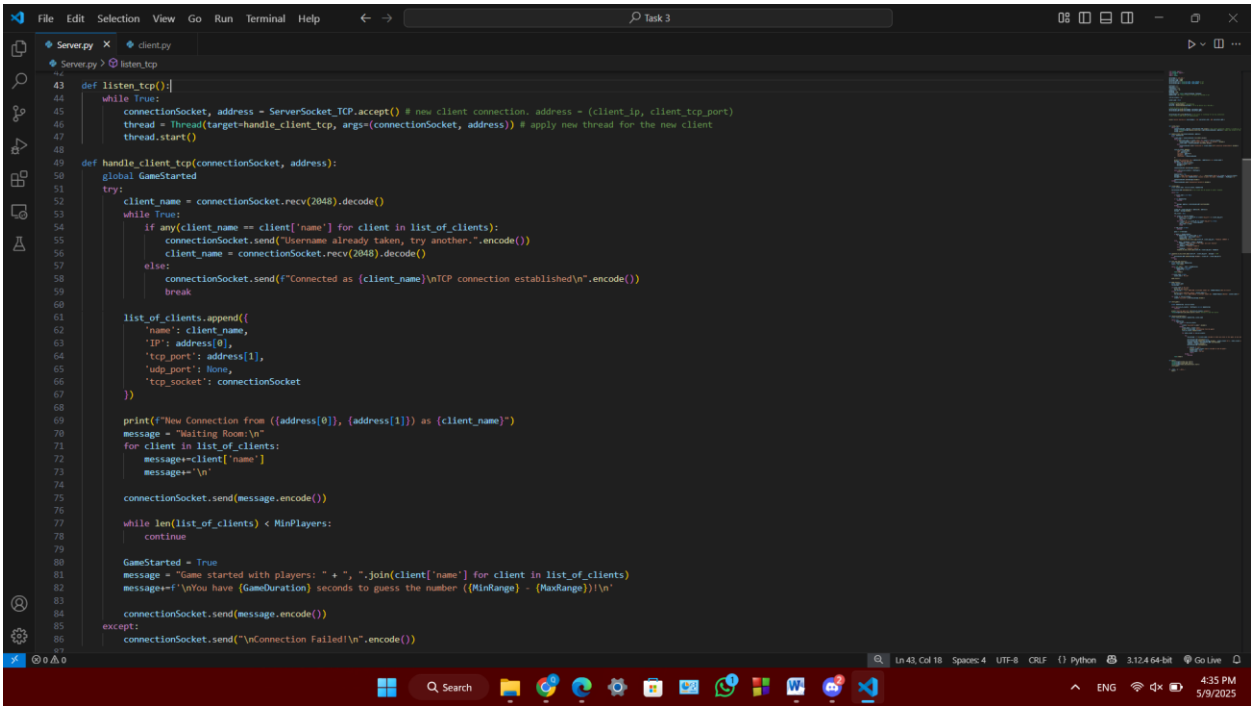


Figure 91 Server Code 3

Figure 89 shows the function that handles clients TCP connection. The server creates a socket of the connected client and receives its address, and then asks the client to enter a player name. The server will ensure that the name is unique; if not, it will send a message to the client via the created socket to enter a new name. After guaranteeing that the name is unique, the server adds the information of the client to the list of clients and sends the waiting room to the client. The server will wait until another client connects to start the game.

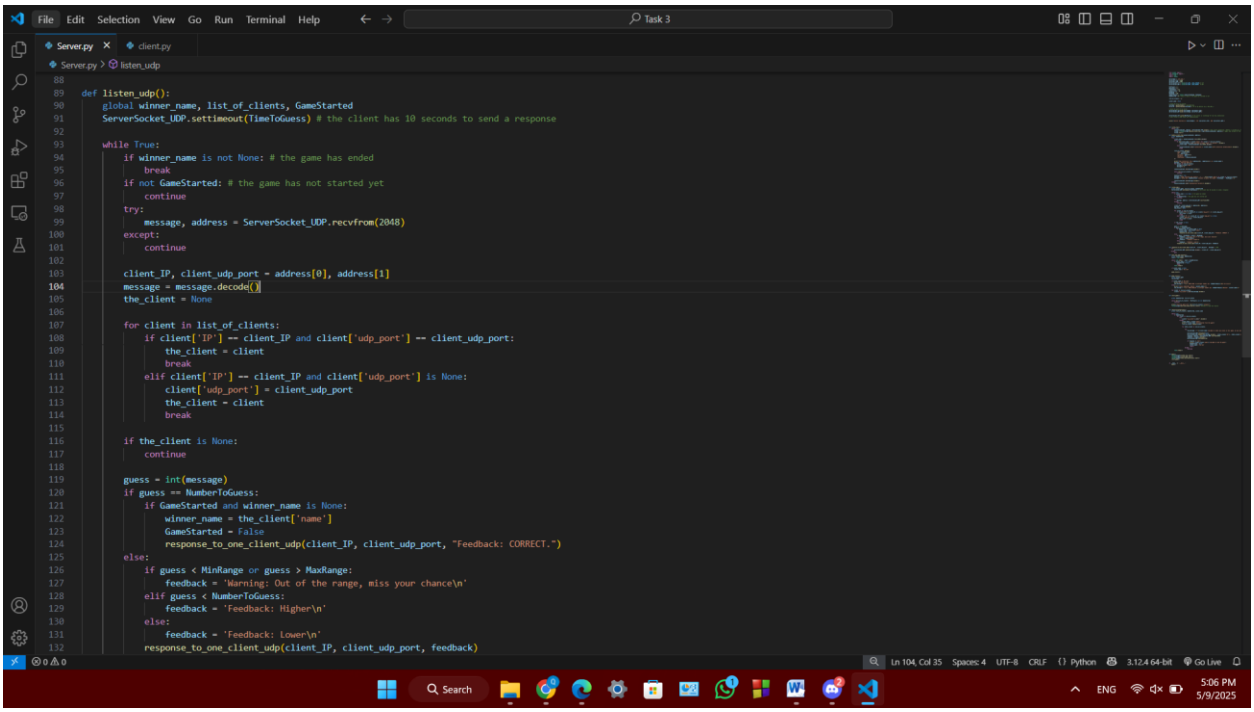


Figure 92 Server Code 4

Once two clients connect to the server, the game will start and the UDP socket will start receive guessing from the clients. As shown in the figure above, UDP socket will receive from each client its guess and it will send a feedback if the client is higher, lower, or equal to the target number.

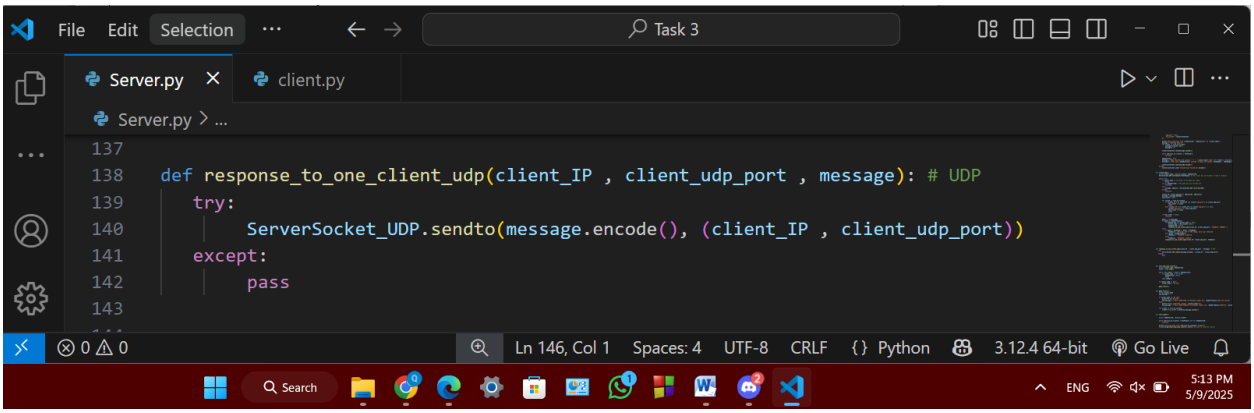


Figure 93 Server Code 5

The figure above shows the function used to send a response to the client. The server sends a feedback to the client using its UDP socket, as we need to specify the client's IP and UDP port number, unlike TCP, where the connection already handles this information.

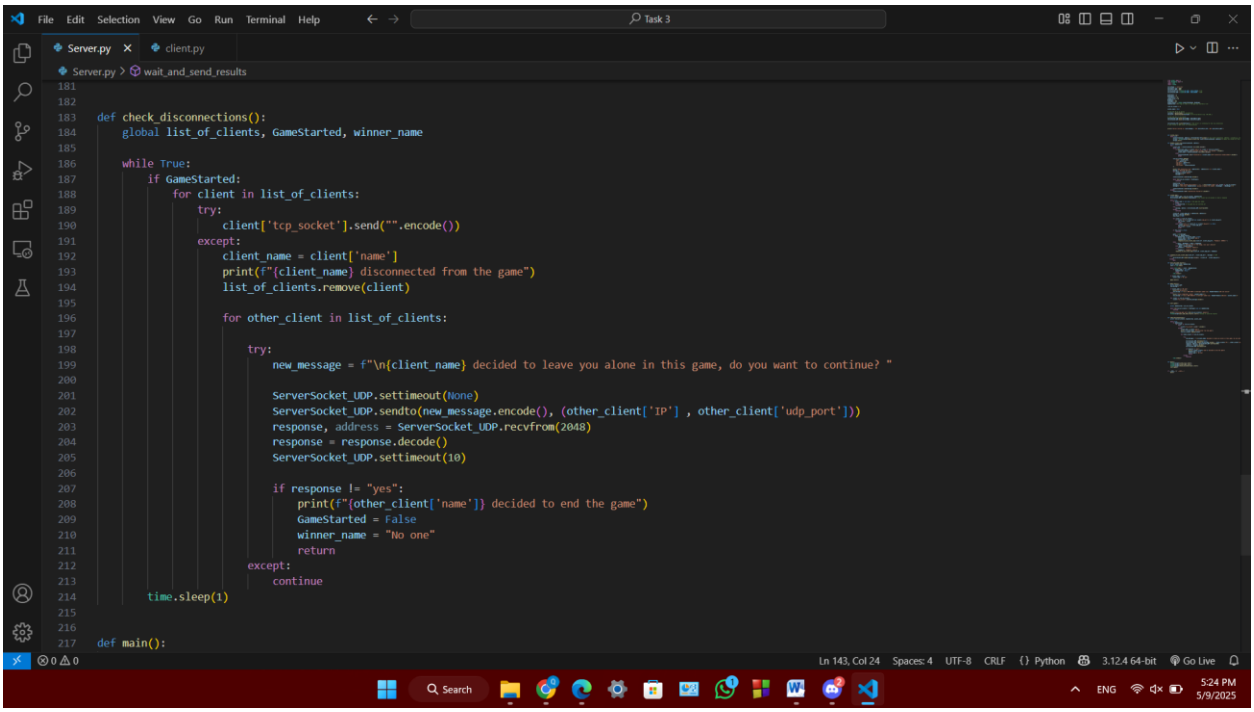
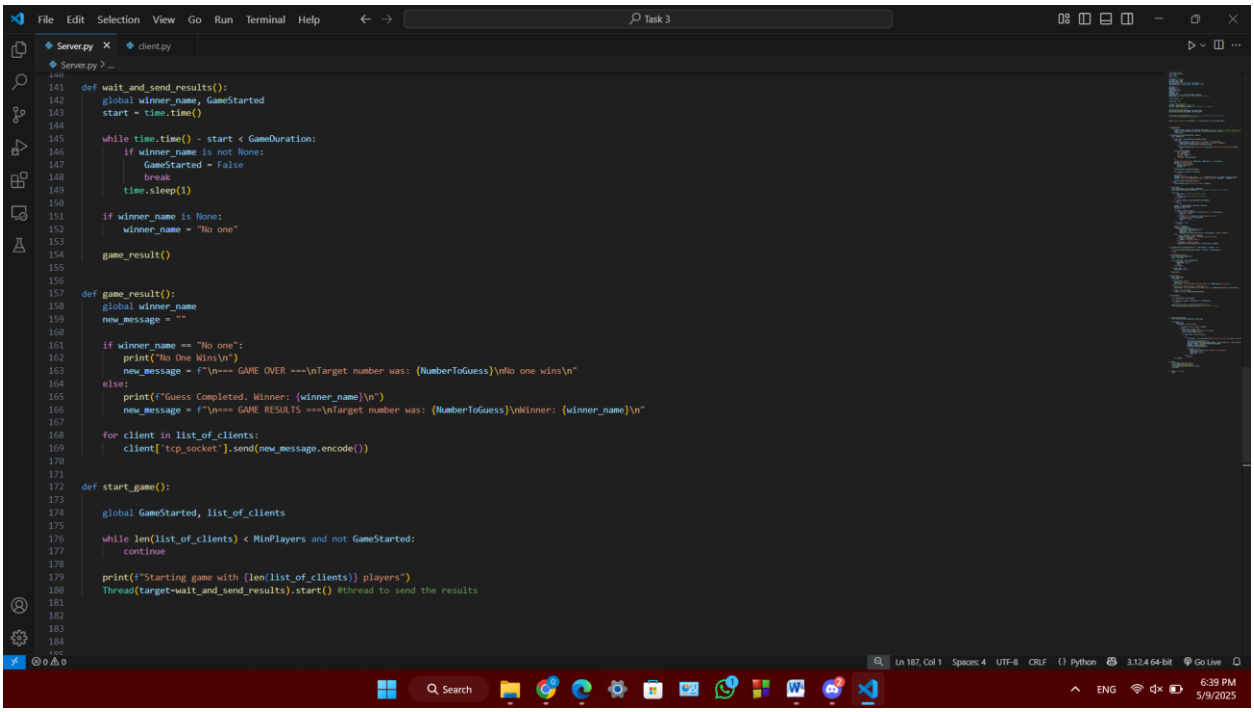


Figure 94 Server Code 6

The figure above shows how the server checks if a client has disconnected. Every second during the game, the server will send an empty message to the client through the TCP socket. If a client did not receive the message, it means that the client has disconnected. The server then asks the remaining client if he wants to continue playing; if not, the game will end with no winner.



```python
def wait_and_send_results():
    global winner_name, GameStarted
    start = time.time()

    while time.time() - start < GameDuration:
        if winner_name is not None:
            GameStarted = False
            break
        time.sleep(1)

    if winner_name is None:
        winner_name = "No one"

    game_result()


def game_result():
    global winner_name
    new_message = ""

    if winner_name == "No one":
        print("No One Wins\n")
        new_message = f"\n=== GAME OVER ===\nTarget number was: {NumberToGuess}\nNo one wins\n"
    else:
        print(f"Guess Completed. Winner: {winner_name}\n")
        new_message = f"\n=== GAME RESULTS ===\nTarget number was: {NumberToGuess}\nWinner: {winner_name}\n"

    for client in list_of_clients:
        client['tcp_socket'].send(new_message.encode())


def start_game():

    global GameStarted, list_of_clients

    while len(list_of_clients) < MinPlayers and not GameStarted:
        continue

    print(f"Starting game with {len(list_of_clients)} players")
    Thread(target=wait_and_send_results).start() #thread to send the results
```

Figure 95 Server Code 7

As we have mentioned before, the main thread runs the start_game() function. As shown in the figure above, the function creates a new thread when the game starts. This thread will run until the game duration ends, one of the clients wins, or all clients disconnects. Game results will be sent to each client through the TCP socket, indicating the winner and the target number.

Python Client Code

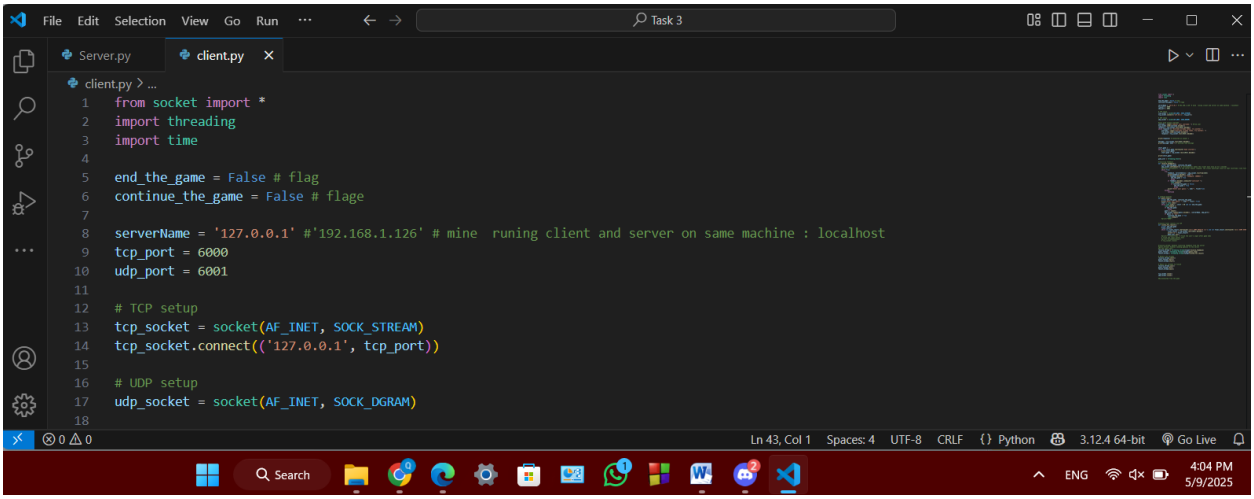


Figure 96 Client Code 1

The figure shows the parameters that we have used for the clients. We have defined two flags: one to determine if the game has ended or not, and the other to determine whether the client want to continue the game or not after another client disconnects. The server name is set to the localhost number to allow connection to the server on the same machine. TCP and UDP ports match those used by the server. TCP and UDP sockets are created for the client, and the TCP socket sends a connection request to the server.

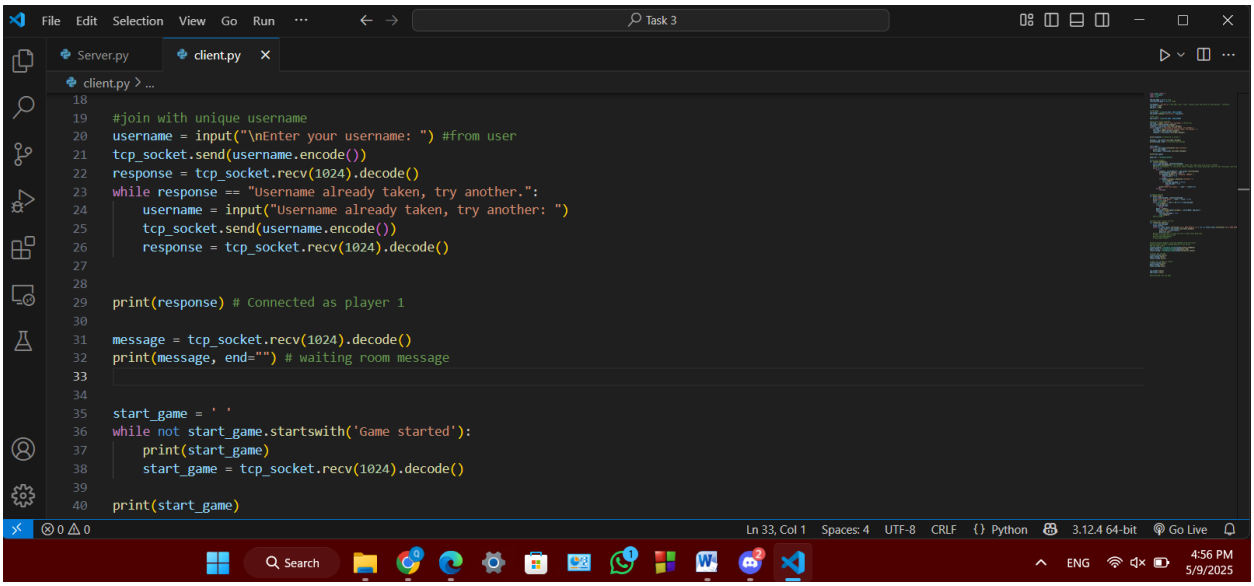


Figure 97 Client Code 2

As shown in the figure above, the client here enters and sends his name to the server using the TCP socket. If the server responds that the name is not unique, the client will keep re-entering and sending a new name until it is unique. Client will then wait for another message from the server indicating the start of game.

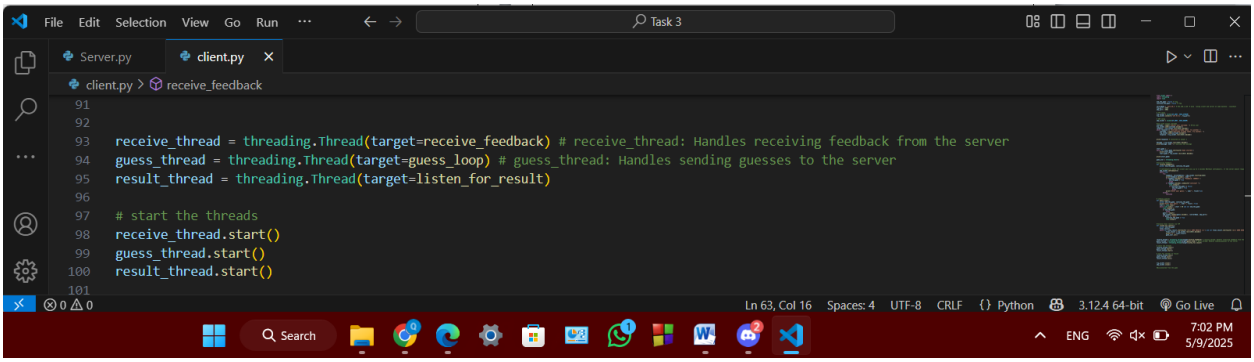


Figure 98 Client Code 3

Once the game starts, three threads will be created: one to handle the feedbacks from the server, another to handle the guess response to the server, and one to receive the final results. Each will be discussed later.

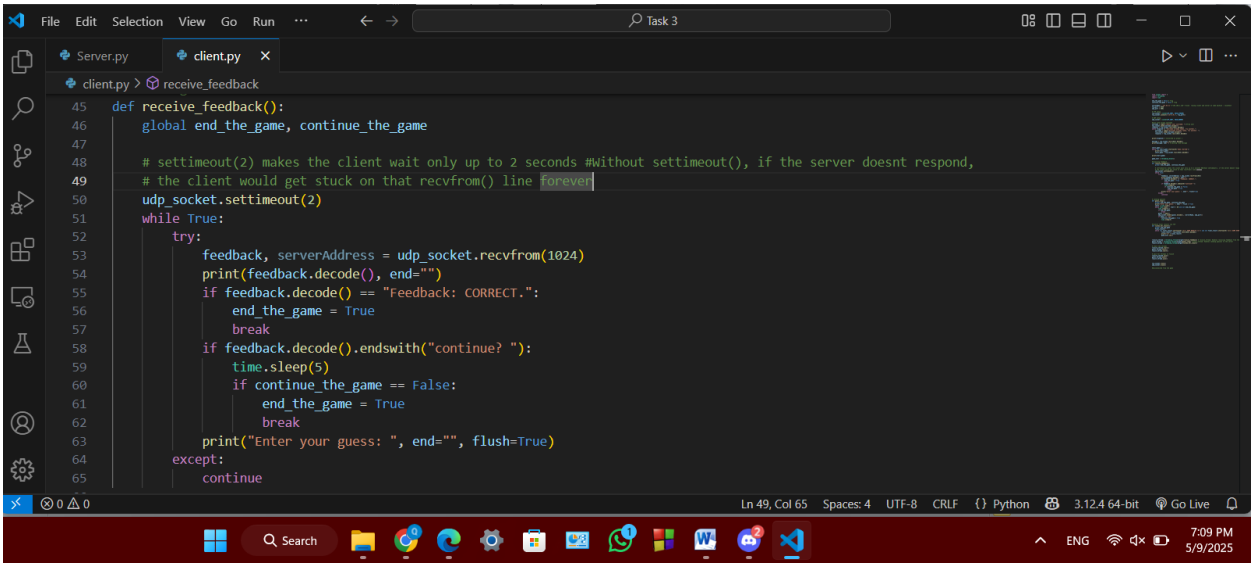


Figure 99 Client Code 4

The figure above shows the how the thread handles the feedbacks from the server. It runs in an infinite loop until it receives a feedback from the server that the last guess was correct, another client has won, or the client decided to leave the game after the other client disconnected.

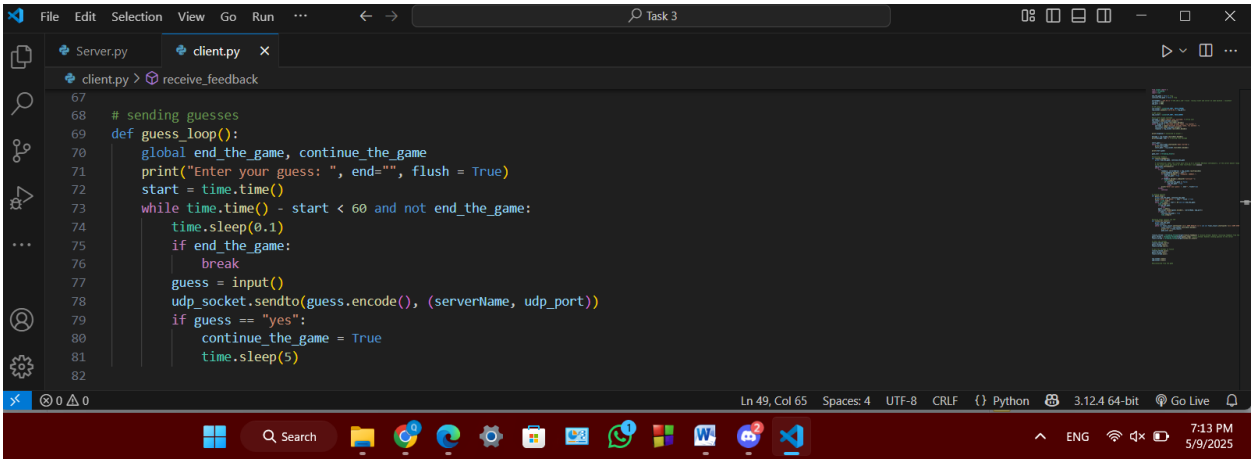


Figure 100 Client Code 5

The thread running the function shown in the figure above handles sending the client's guess to the server using the UDP socket. It also sends the response indicating if the client wants to continue or not after the other client disconnects. A small time delay is added to avoid communication issues.

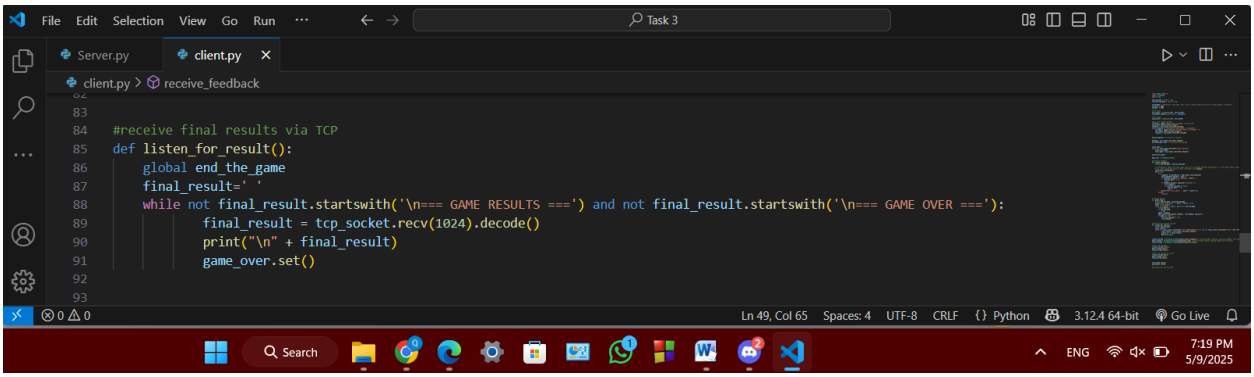


Figure 101 Client Code 6

The final thread, running in the function shown in the figure above, waits for a response from the server on its TCP socket indicating that the game has ended and receives the results of the game.

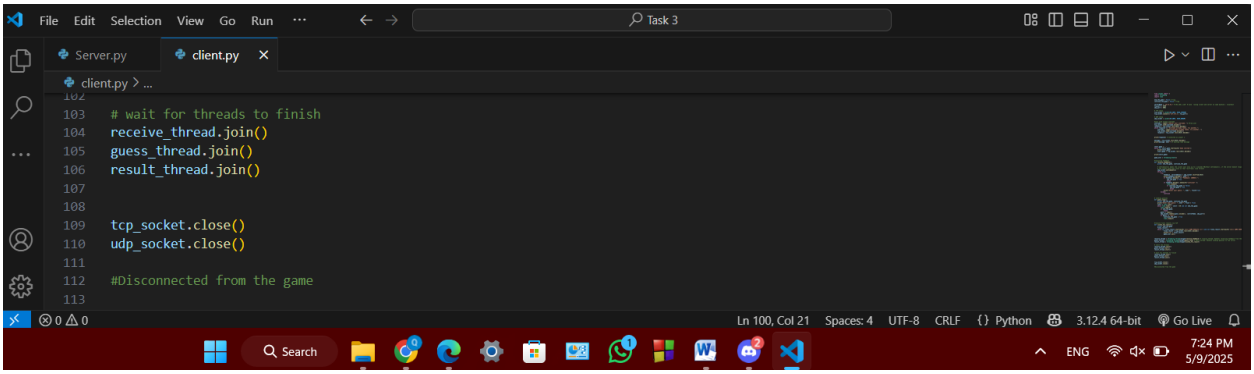


Figure 102 Client Code 7

Finally, the figure above shows the end of the client code, where the main thread waits for other threads to finish and closes the TCP and UDP sockets.

Test Cases

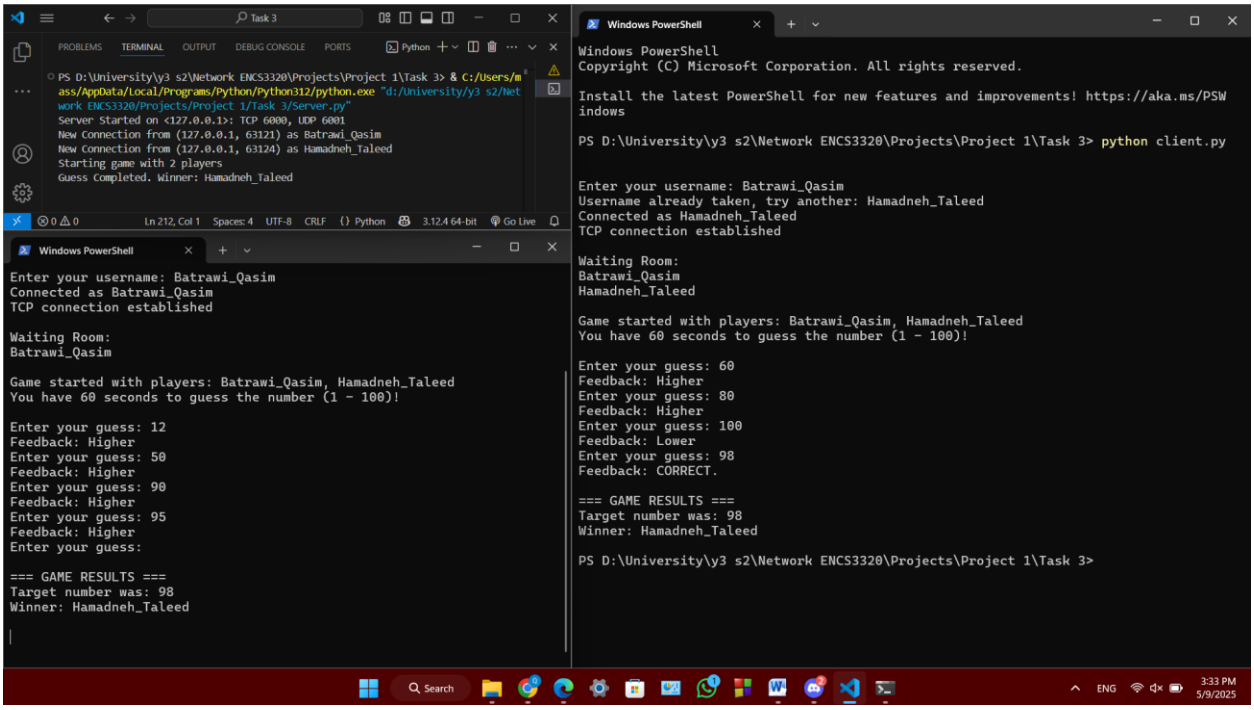


Figure 103 Test Case 1 – Correct Guess & Invalid Name

The above figure shows a gameplay of two clients. The server opens his TCP ad UDP sockets, and waits for clients to connect. The first client has connected with a name **Batrawi_Qasim** and waits for another client to connect. The second client connected and tried to enter the same name as client one, but the server reject it, so she entered the name as **Hamadneh_Taleed**. The game starts and the two clients were trying to guess the number, Hamadneh_Taleed has guessed the number first and wins the game while Batrawi_Qasim was still trying, so the game ends immediately and the server sent the results to both clients.

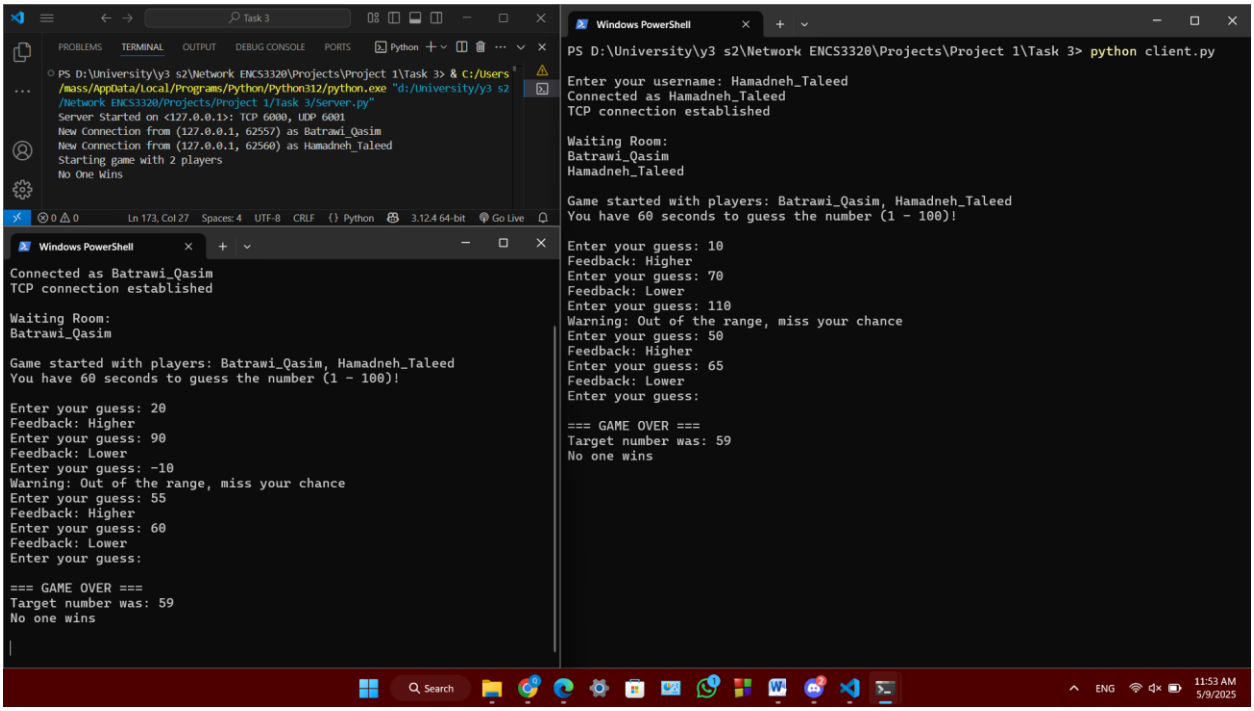


Figure 104 Test Case 2 – Time Out & Invalid Input

The above figure shows another test case where the clients **Batrawi_Qasim** and **Hamadneh_Taleed** were trying to guess the number. However, the 60 seconds game duration ran out before anyone guessed correctly, so the server ended the game for both clients with no winner. During the gameplay, both clients entered numbers that were out of range, and the server responded with a warning.

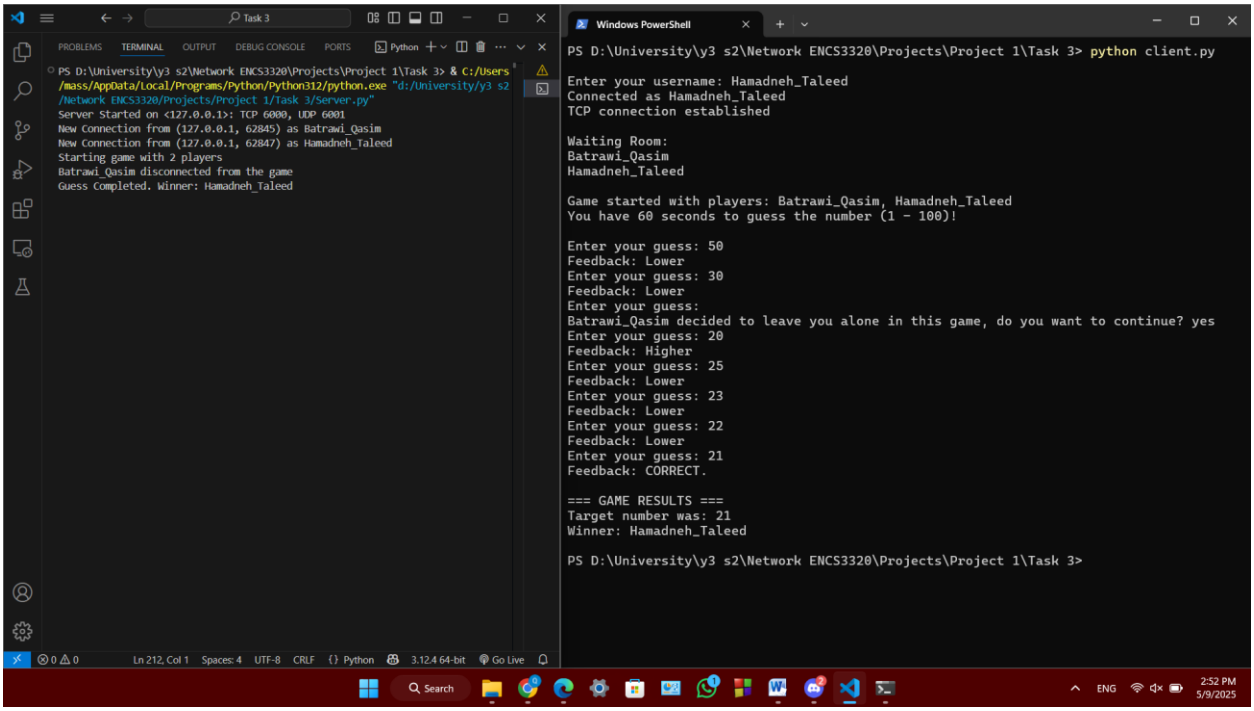


Figure 105 Test Case 3 - Client Disconnection.

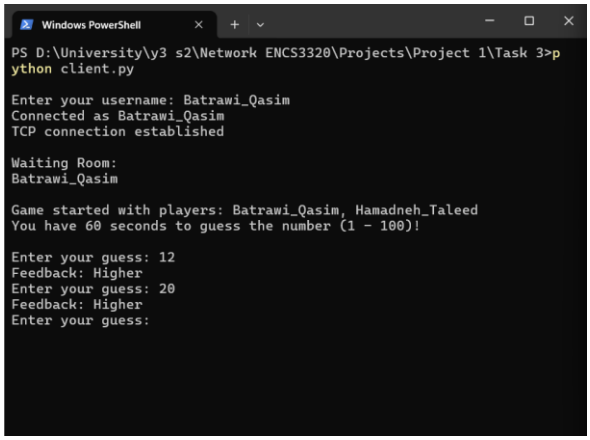


Figure 106 Test Case 3 - Terminal of Disconnected Client

The figure above shows a gameplay of two clients. One of the clients, Batrawi_Qasim, decided to leave the game after his second guess while the game was running. When he disconnected, the server notified the other client, Hamadneh_Taleed, and asked her if she want to continue the game or not, and she decided to continue guessing and she won the game after that.

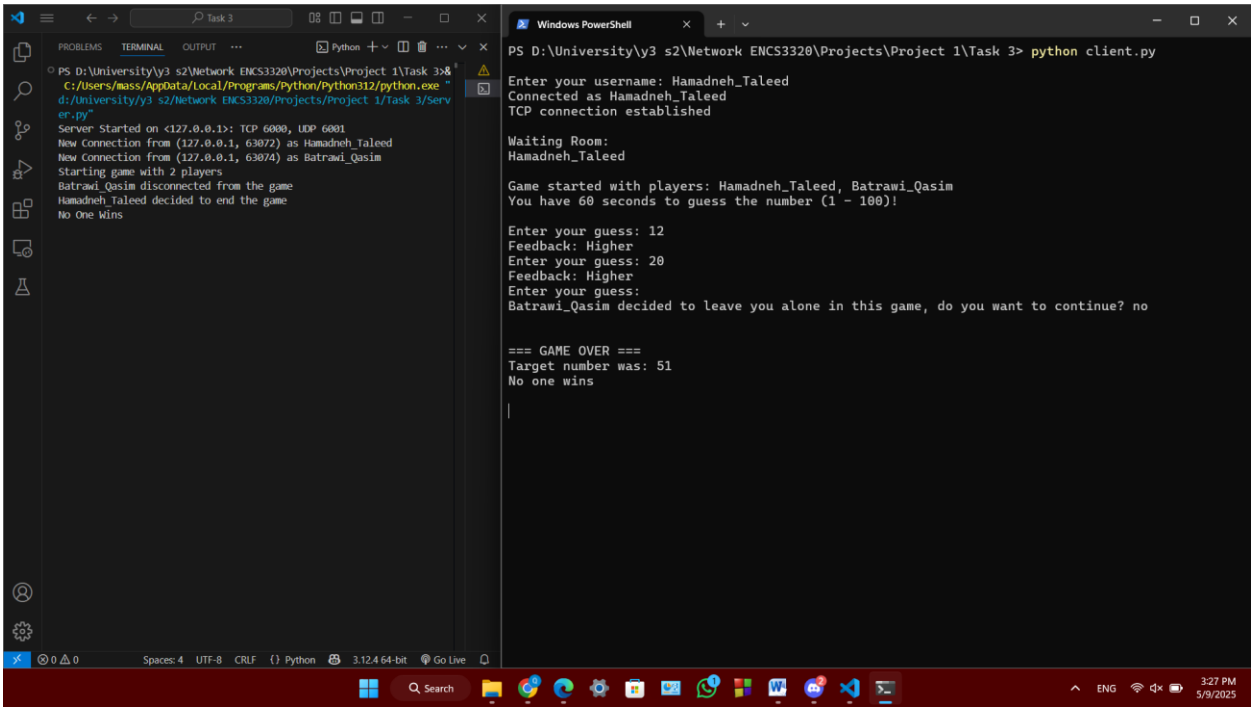


Figure 107 Test Case 4 - Client Disconnection

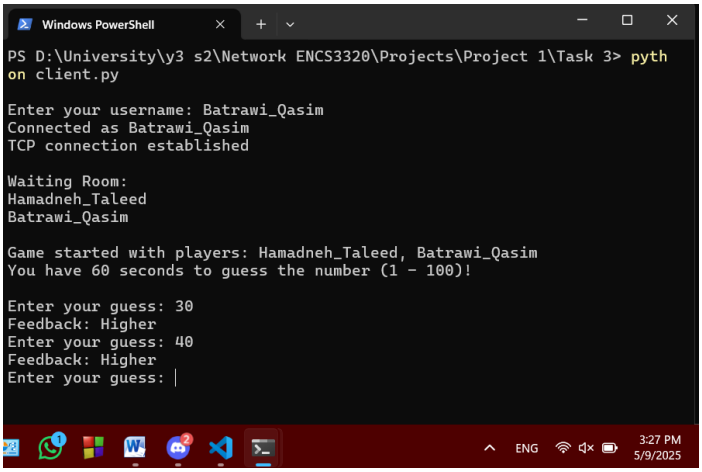


Figure 108 Test Case 4 - Terminal of Disconnected Client

The figure above shows a gameplay of two clients. Similar to the previous game, one of the clients, Batrawi_Qasim, decided to leave the game after his second guess while the game was running. When he disconnected, the server notified the other client, Hamadneh_Taleed, and asked her if he want to continue the game or not, and she decided to end the game with no winner.

Alternative Solutions, Issues, and Limitation

All parts of the project are working as expected. As an alternative approach, we could have used a higher-level framework like Flask to simplify some tasks. During development, we faced some issues with multithreading, especially when multiple threads accessed shared data at the same time. To solve this, we created threads responsible for TCP, and other responsible for UDP. We also used some sleep functions to help with synchronization and timing. Other minor challenges came up along the way, but we were able to fix them through testing and debugging. Overall, everything is running smoothly.

Teamwork

Throughout the project, tasks were divided between Taleed Hamadneh and Qasim Batrawi to ensure balanced collaboration. In Task 1, Taleed handled the first five shell commands, while Qasim completed the remaining commands and conducted Wireshark analysis. For Task 2, Qasim designed the main web page, and Taleed worked on the local web page. Within the server-side implementation of Task 2, Taleed was responsible for developing the connection methods, the function to respond to clients, and for splitting the files into appropriate types such as HTML and images. Qasim focused on retrieving the correct files, managing error responses, and implementing redirection handling. Task 3 was completed together during a Zoom meeting, with both members equally contributing to the planning and implementation. The project report was also divided fairly: Qasim wrote the Task 3 section, while Taleed wrote about Task 2. Task 1, as well as the theory and procedure sections, were completed together. This balanced approach led to efficient teamwork and a well-rounded final submission.

Figure 109 Teamwork chart

References

[1]: James W. Kurose, Keith W. Ross – Computer Networking_A top-Down Approach Pearson.pdf

[2]: Network Component: UDP Socket

[3]: https://www.theknowledgeacademy.com/blog/what-is-wireshark/

[4]: GeeksforGeeks | Your All-in-One Learning Portal