Faculty of Engineering and Technology Department of Electrical and Computer Engineering

ENCS3320 – Computer Networks

Project #1 (Socket Programming)

Prepared by:

Alaa Faraj	**1220808**	Section:**2**
Adnan Odah	**1220175**	Section:4
Abedalrheem Fialah	**1220216**	Section:**2**

Date: 11/5/2025

Abstract:

This project explored socket programming and computer networking by implementing practical networking tasks designed to apply theoretical knowledge through real-world application. The project included three main tasks, analyzing basic network commands and capturing DNS traffic using Wireshark, developing a web server that listens on a specific port and responds to HTTP requests with structured HTML content, and creating a client-server multiplayer game that utilizes both TCP and UDP protocols to handle setup and gameplay communication. The implementation was done using the Python programming language and its standard socket library to build TCP and UDP connections. Additional technologies such as HTML, CSS, and Wireshark were also used to complete the required components. We successfully developed a functioning web server, and a real-time guessing game based on core networking principles and hybrid socket communication. Overall, the project provided valuable insight into the principles and practical challenges of network programming, applying essential skills for future development in distributed systems.

Table of Contents

Table of Figures:

List of Tables:

1. Theory:

1.1 Network Diagnostic Tool:

A network diagnostics tool is a troubleshooting solution that encompasses the process of evaluating, analyzing, and resolving issues within a network infrastructure. A network diagnostics tool can perform complex tasks, from troubleshooting operating systems to simple checks like ensuring cables are properly connected. The end goal is to mitigate performance issues within an organization's networks and ensure the smooth functioning of all connected systems and services. This is a fundamental step in network management and is crucial for organizations to maintain efficient and reliable connectivity. [1]

In this project, we used **basic command-line tools**, which are built into most operating systems and are used for quick diagnostics. Below are the key tools used:

A. **Ipconfig**:

Displays the TCP/IP network configuration of the device, including information such as IPv4 address, IPv6 address, subnet mask, and default gateway.

ipconfig/all: "/all" subcommand (also called switches) added to "ipconfig" command, this shows additional details such as MAC address, DNS servers, and DHCP information.

B. **Ping** <server>:

Sends ICMP echo request packets to a specified server to test connectivity and measure round-trip time.

Example:

ping google.com

Send 4 packets by default and reports on whether replies are received, along with latency.

C. **telnet**:

used to connect to remote servers via a specified port.

Example:

telnet example.com 80

If successful, this confirms that the host is reachable and that the port is open.

D. **treacert/traceroute** <server>:

Trace the path packets take from your device to a destination server, showing each intermediate hop and its response time.

E. **nslookup** <server>:

Queries the Domain Name System (DNS) to obtain domain name or IP address mapping, or other DNS records.

1.2 Wireshark:

Wireshark is a powerful and widely-used network protocol analyzer that captures packets—discrete units of data—traveling over a network connection, such as between your computer and the internet. It allows IT professionals and cybersecurity experts to inspect network traffic at a granular level, whether in real-time or through offline analysis. This tool is essential for diagnosing performance issues, identifying suspicious activity, and gaining visibility into the underlying behavior of a network. With Wireshark, users can trace connections, dissect the contents of suspect transactions, and pinpoint bursts of traffic or anomalies that may indicate deeper problems. Its robust filtering capabilities allow zooming in on specific packets, making it invaluable for troubleshooting, network analysis, and improving network security. Wireshark is an indispensable part of any IT professional's toolkit—and using it effectively requires both knowledge and precision. [2]

1.3 Socket Programming:

Socket programming enables two nodes on a network to establish communication with each other. one socket (typically the server) listens for incoming connections on a specific IP address and port, while the other socket (the client) initiates the connection. The server creates a listening socket, and the client connects to it to begin data exchange. This approach is widely used in various applications such as instant messaging, real-time document collaboration, binary data streaming, and online media platforms. [3]

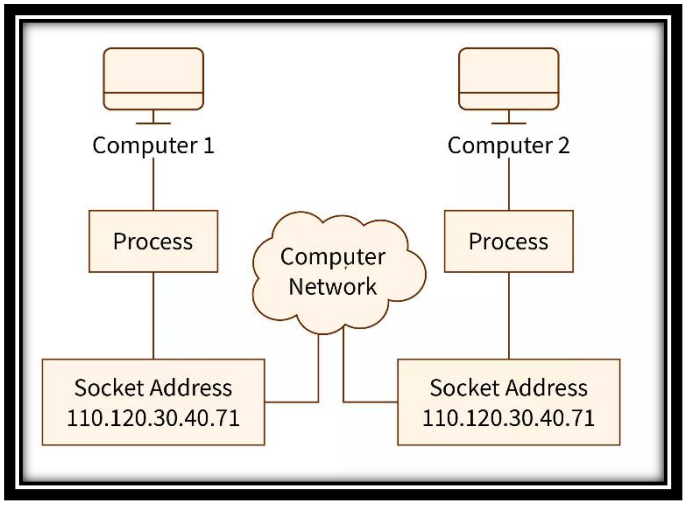


Figure 1:socket programming diagram [4].

To understand how socket programming works in practice, it's important to break down the essential components that enable communication between two networked devices:

- **Socket**:

  one endpoint of a two-way communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication takes place.

- **IP Address**:

  This identifies the host (server or client) on the network. It's essential for locating the device you want to communicate with.

- **Port Number**:

  A port number specifies the particular service or application on the host that the socket is communicating with (e.g., port 80 for HTTP,).

- **Protocol (TCP/UDP):**

  - **TCP (Transmission Control Protocol):**

    Provides reliable, connection-oriented communication with error checking and flow control.

  - **UDP (User Datagram Protocol):**

    Offers faster, connectionless communication with less overhead, butwithout guaranteed delivery.

- **Socket Programing with TCP (Transmission Control Protocol):**

The Transmission Control Protocol (TCP) is a connection-oriented protocol that ensures reliable data exchange between devices over a network. Operating at the transport layer of the OSI model, TCP establishes a connection between sender and receiver before transmitting data. It guarantees data integrity by using error checking, sequencing, and retransmitting lost packets. TCP is essential for applications where reliability is critical, such as web browsing, file transfers, and email, and it works with the Internet Protocol (IP) to route data across the network. [5]

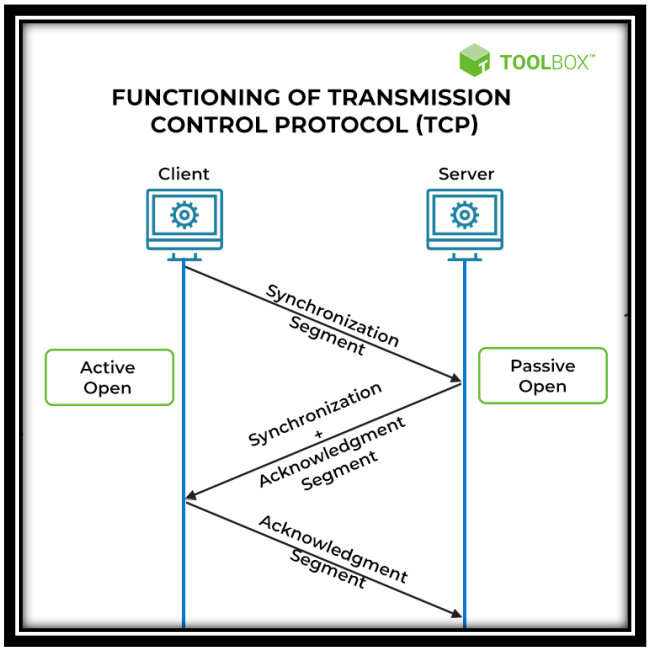


Figure 2:TCP connection [5].

TCP socket programming involves creating a reliable communication channel between a client and a server over a network. In this model, the server listens for incoming connections on a specific port, while the client initiates a connection to the server. Once the connection is established, both parties can exchange data in a reliable, ordered manner, ensuring that messages are received correctly.

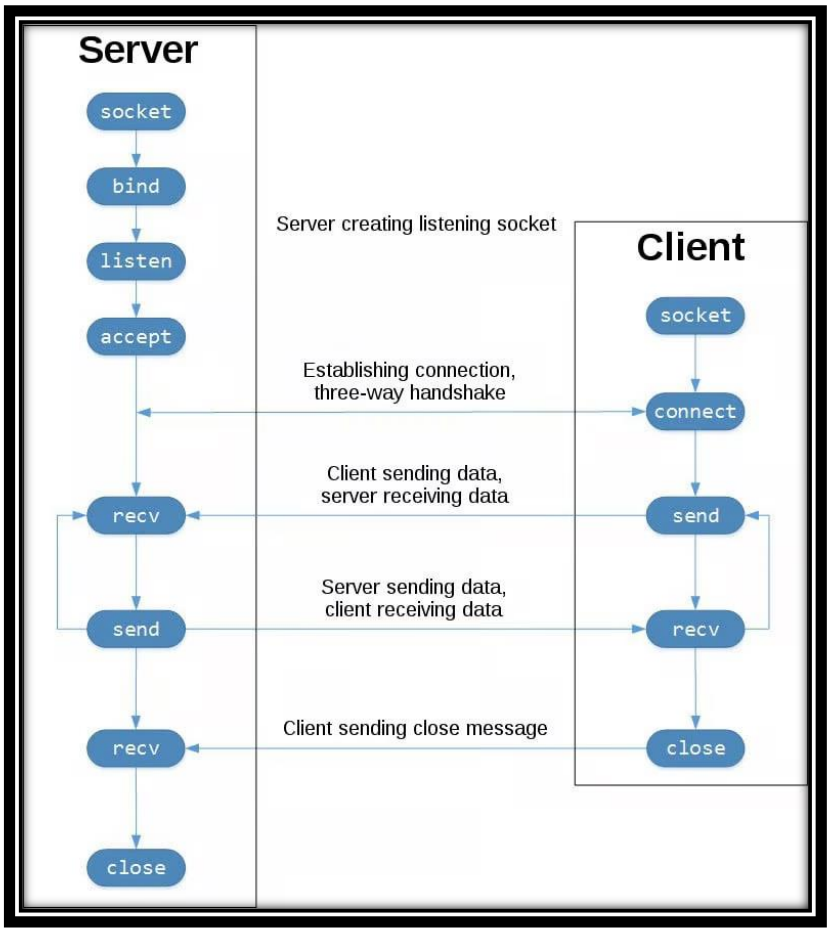


Figure 3:TCP socket Flow [6].

- **Socket Programing with UDP:**

The User Datagram Protocol (UDP) is a connectionless, message-based communication protocol that enables devices and applications to transmit data across a network without guaranteeing delivery or order. It is ideal for real-time applications and broadcasting, where speed is prioritized over reliability. [5]

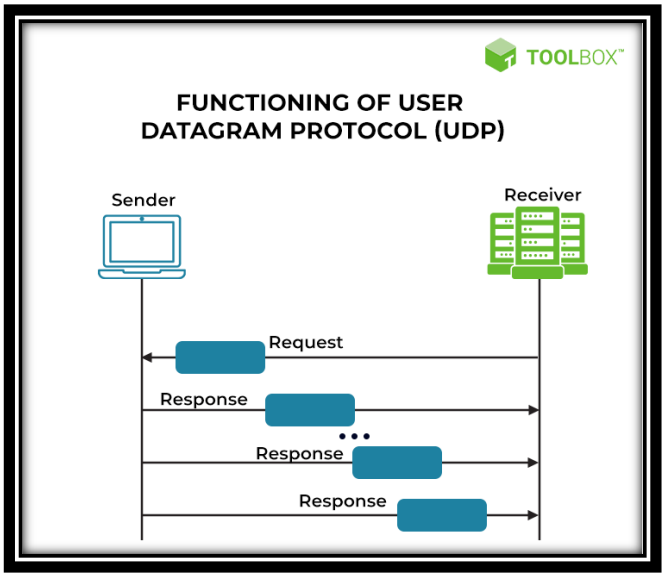


Figure 4:UDP Connection [5].

UDP socket programming enables communication between a client and a server without establishing a dedicated connection. In this model, data is sent in discrete packets called datagrams, which are transmitted independently and may arrive out of order or not at all. The server typically binds to a specific port and listens for incoming datagrams, while the client sends data directly to the server's IP and port, making UDP fast and suitable for time-sensitive applications where occasional data loss is acceptable.

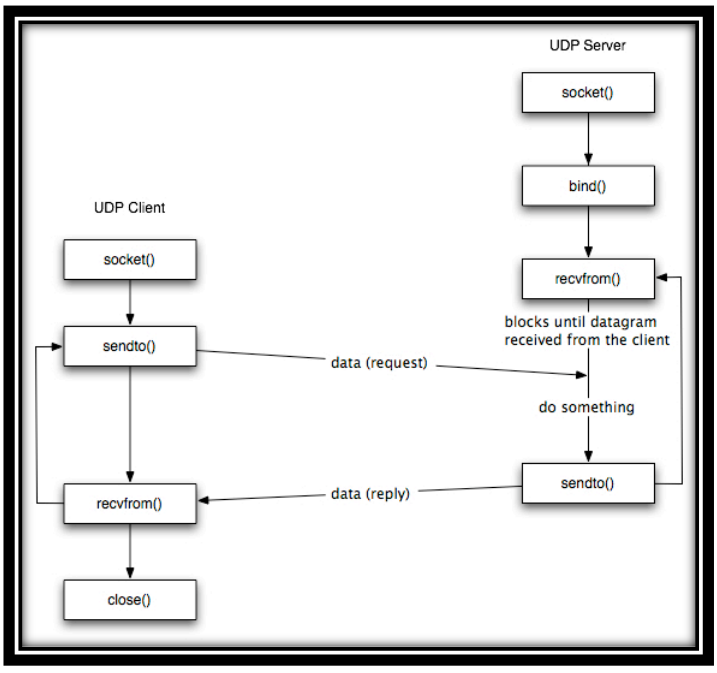


Figure 5:UDP Socket Flow [7].

- **TCP vs UDP**:

Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are two core transport layer protocols used for data communication over networks. TCP is a **connection-oriented protocol** that ensures reliable, ordered, and error-checked delivery of data between applications. It's ideal for situations where accuracy and data integrity are essential, such as web browsing, file transfers, and emails. On the other hand, UDP is a **connectionless protocol** that sends packets without guaranteeing delivery or order, making it much faster and better suited for real-time applications like video streaming, online gaming, and voice calls. Understanding the trade-offs between TCP and UDP is crucial for selecting the appropriate protocol depending on the application's performance and reliability requirements.

Table 1:TCP vs UDP table.

	TCP	**UDP**
Connection Type	Connection-oriented	Connectionless
Reliability	Reliable	Unreliable
Speed	Slower due to overhead of connection and checks	Faster due to minimal overhead
Data Transfer	continuous flow of data	sends individual packets
Overhead	Higher	Lower
Packet Acknowledgment	Yes	No

1.4 Web Server:

A web server combines both hardware and software components to deliver website content over the internet.

- Hardware: A physical machine that stores website files (HTML, CSS, JS, images) and connects to the internet.
- Software: An HTTP server that processes browser requests, locates files, and sends responses (e.g., web pages or error messages).

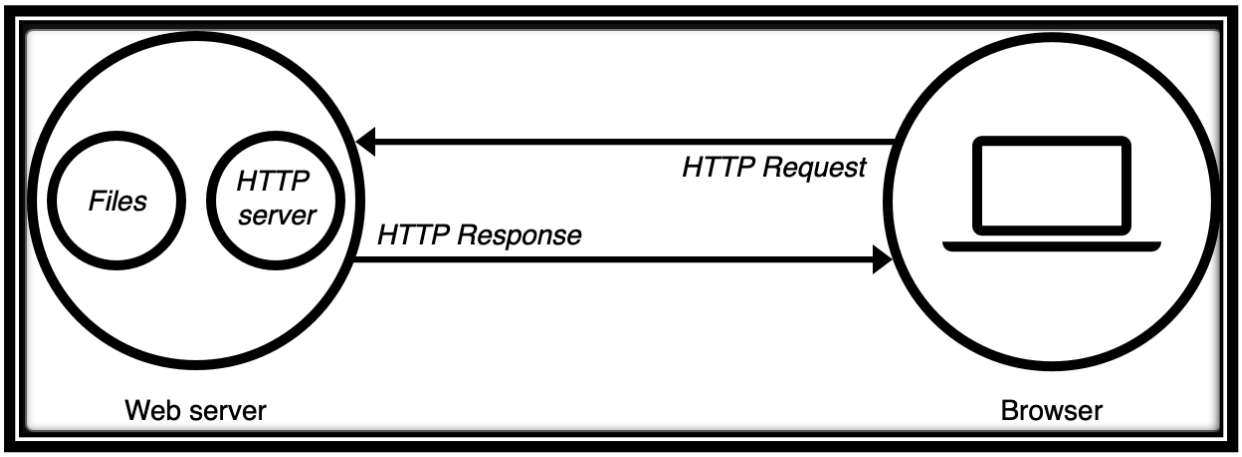


Figure 6:Web server and browser connection [8].

Server Response and Incoming Requests:

When a client (usually a web browser) sends a request to a web server, it typically uses the HTTP (HyperText Transfer Protocol) to communicate. The process involves the following steps:

- **Incoming Requests**: A client sends a request, typically in the form of a URL, asking the web server for specific content (like an HTML page, image, or data). This request includes details such as the type of content desired, any parameters (like query strings), and additional headers that specify things like browser type or accepted content formats.
- Server Processing: The web server receives the incoming request and processes it. If the requested file exists and can be served, the server prepares the response by locating the file, processing any required dynamic content (such as executing server-side scripts), and formatting it for the client.

- **Server Response**: After processing the request, the server sends a response back to the client. This response generally includes:
    - **Status Code**: A numeric code indicating the result of the request (e.g., 200 OK for successful requests, 404 Not Found for missing content).
    - **Headers**: Additional information about the response, such as the content type (Content-Type: text/html), caching policies, and more.
    - **Body**: The actual content requested by the client, such as HTML, CSS, JavaScript, images, or JSON data.

The web server continues to respond to incoming requests, delivering requested files or error messages until the connection is closed. This entire process forms the backbone of how the web operates, ensuring that users can access content efficiently.

2. Procedure:

2.1 Task 1 – Network Commands and Wireshark:

2.1.1 ipconfig /all:

Execute the **ipconfig /all** command on your computer and locate the IP address, subnet mask, default gateway, and DNS server addresses for your main network

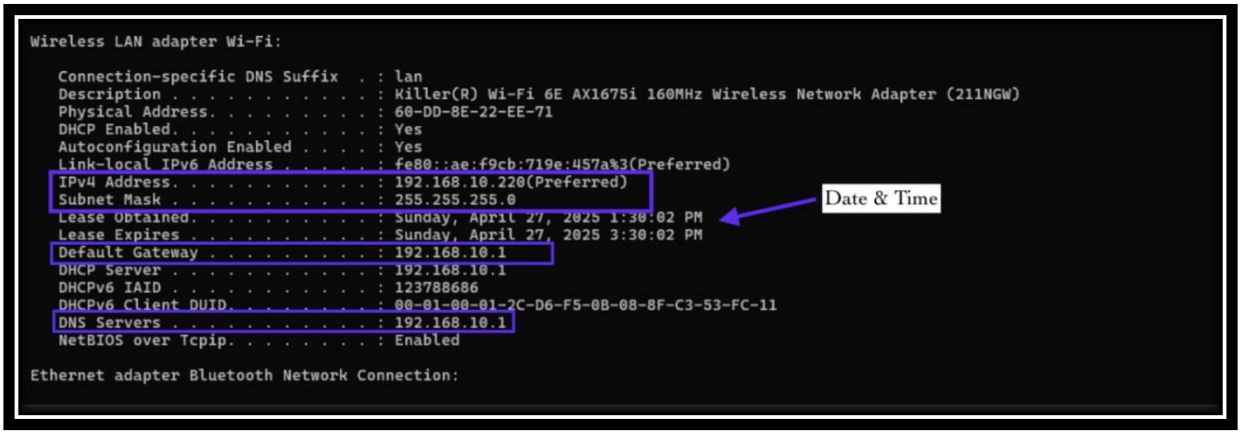


*Figure 7: **ipconfig /all** command.*

interface.

To explain the results:

- IPv4 address (192.168.10.220) is a local IP given by the router. It identifies your device on the local network so it can communicate with others and access the internet.
- Subnet mask (255.255.255.0) is set by the router. It splits the IP address into network and device parts to check if other devices are on the same local network.
- Default gateway (192.168.10.1) is assigned by the router. It forwards data from your device to destinations outside the local network, like the internet.
- DNS server (192.168.10.1), typically provided by the router, converts domain names like google.com into IP addresses.

2.1.2 Ping:

Send a **ping** request to a device within your local network, such as from a laptop to a smartphone connected to the same wireless network.

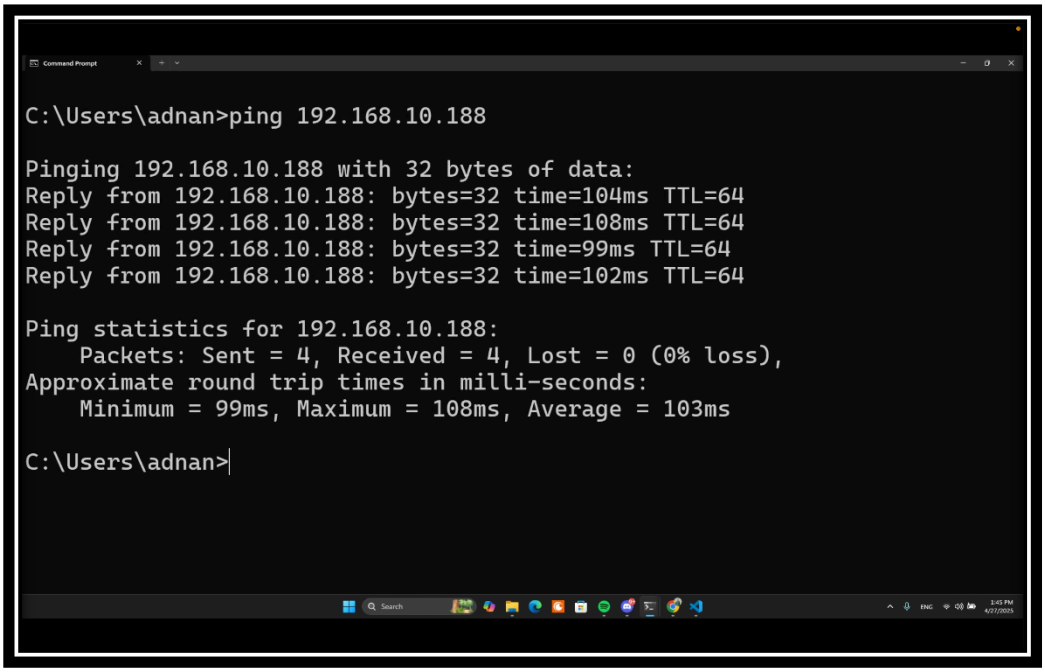


Figure 8: Ping command.

The ping command is used to test the connectivity between your computer and another device on a network:

- All 4 ping requests received replies (sent = 4, received = 4), indicating that the target device (the mobile phone with IP 192.168.10.188) is online and reachable.
- 0% loss which confirms stable communication between the two devices.
- Round Trip Time (RTT) : Minimum = 99ms, Maximum = 108ms, and Average of 103 ms.
- Time To Live (TTL) : 64, which means that the packet likely came from device on the same local network which is correct.

2.1.3 Ping gaia.cs.umass.edu:

Ping gaia.cs.umass.edu, and using the results, provide a brief explanation of whether you believe the response originates from.

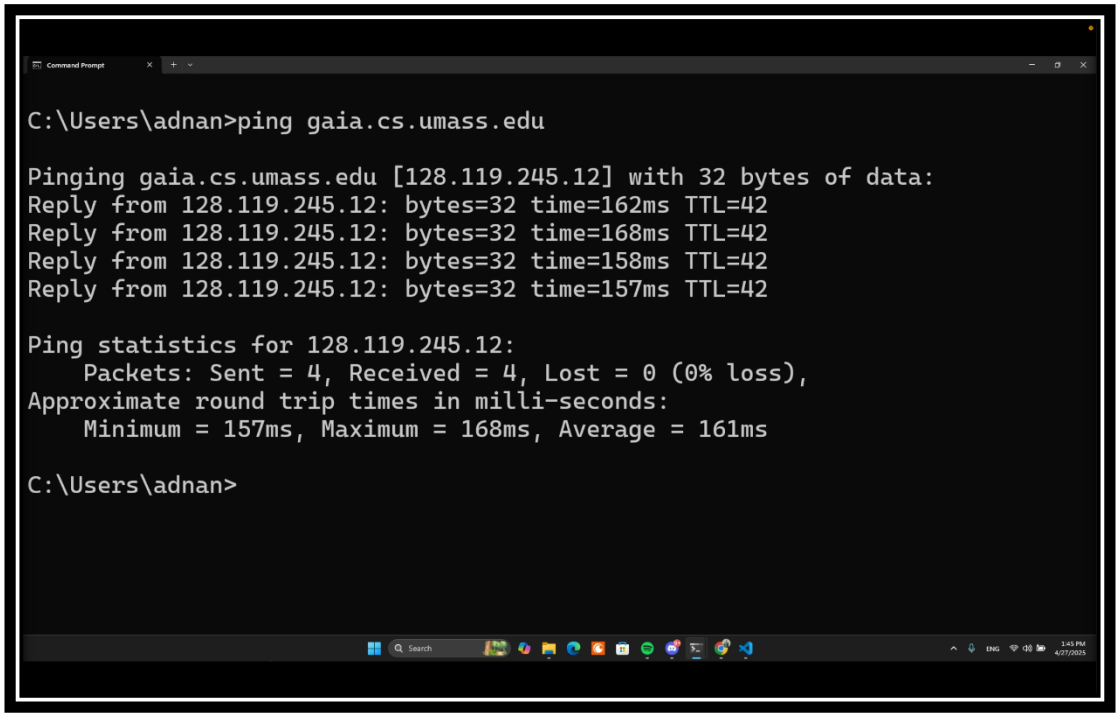


Figure 9: Ping gaia.cs.umass.edu.

The ping command tests connectivity to the remote server **gaia.cs.umass.edu** (IP: 128.119.245.12)

- **All 4 requests received replies** (Sent = 4, Received = 4), confirming the server is reachable.
- **0% packet loss**, indicating a stable connection.
- Round-Trip Time (RTT): Minimum = 157ms, Maximum = 168ms, and Average of 161ms.
- Time To Live (TTL) : 42, which means that the server is multiple hops away (likely over the internet).

2.1.4 tracert/traceroute:

Run **tracert/traceroute** gaia.cs.umass.edu to see the route it takes.

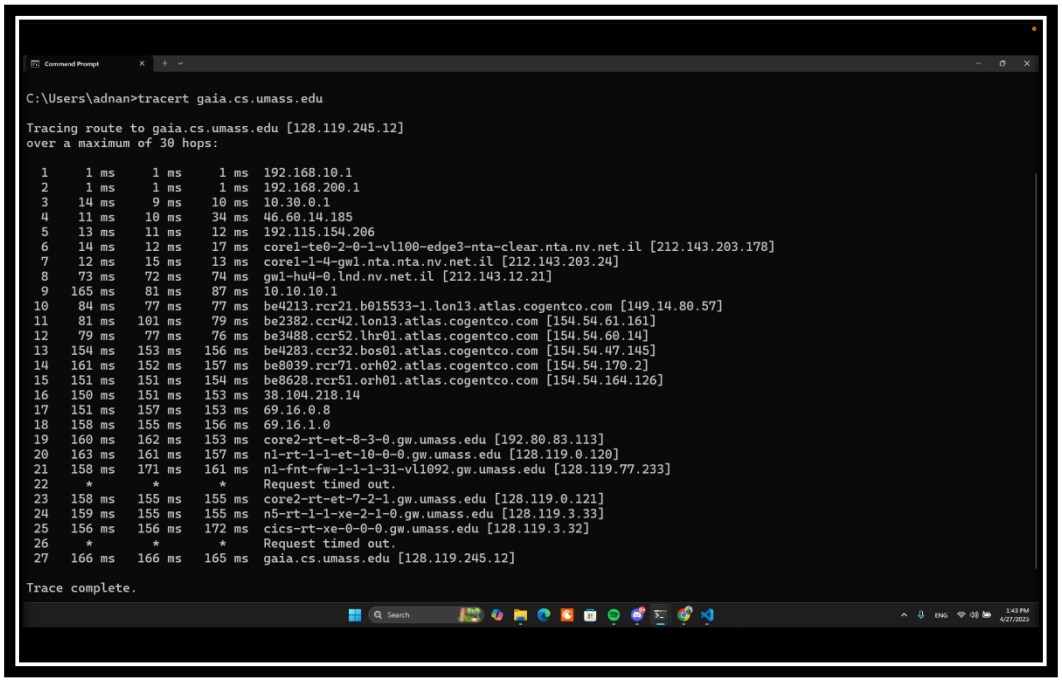


Figure 10: tracert/traceroute command.

Tracert command to gaia.cs.umass.edu (IP:128.119.245.12) shows the route taken across 27 hops, starting with fast local network responses (1ms), passing through international ISP nodes (peaking at 170ms in the U.S.), and finally reaching the destination with 165ms latency; some hops (22, 26) timed out due to firewalls blocking ICMP, and the last IP Address represents the destination host and where the packet reaches its final destination, However, the trace completed successfully, confirming connectivity

2.1.5 nslookup:

Execute the **nslookup** command to retrieve the Domain Name System (DNS) information for gaia.cs.umass.edu.

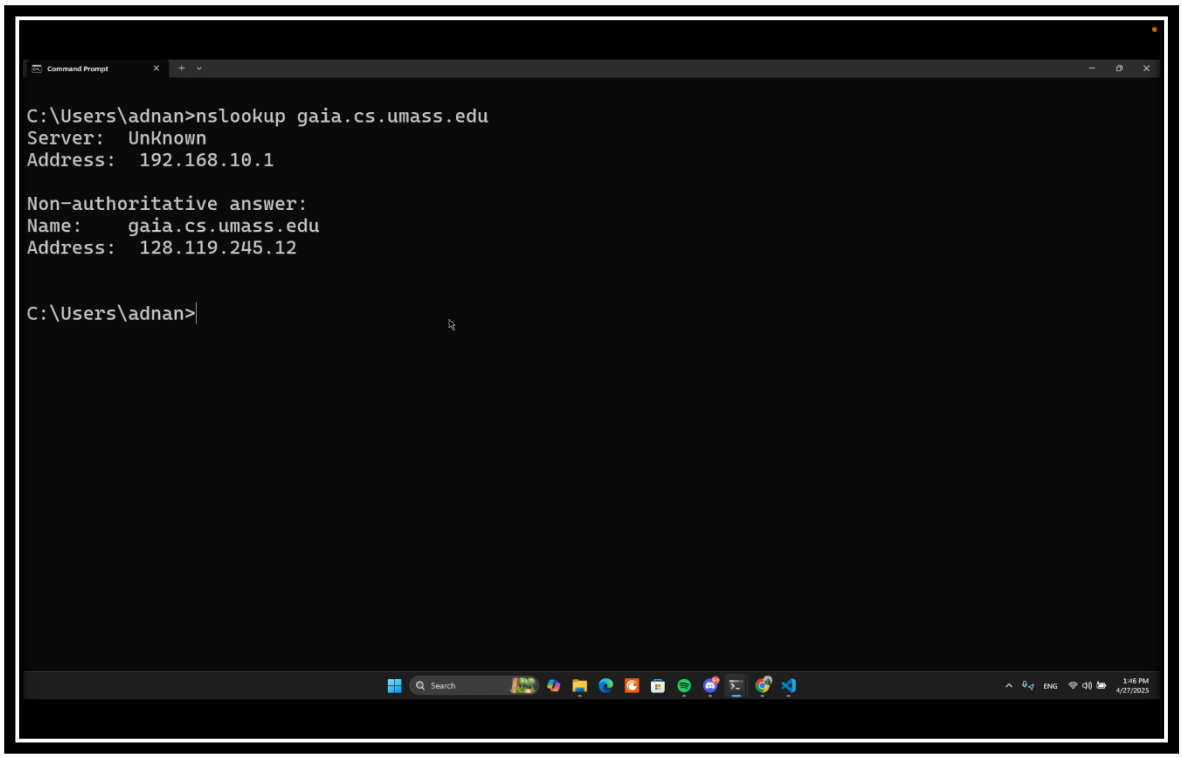


Figure 11: nslookup command

The nslookup command for gaia.cs.umass.edu successfully resolved the domain to IP address 128.119.245.12 using the local DNS server at 192.168.10.1, with non-authoritative answer indicating the response came from the DNS cache rather than the authoritative nameserver for the domain.

2.1.6 telnet:

Use **telnet** to try connecting to gaia.cs.umass.edu at port 80.

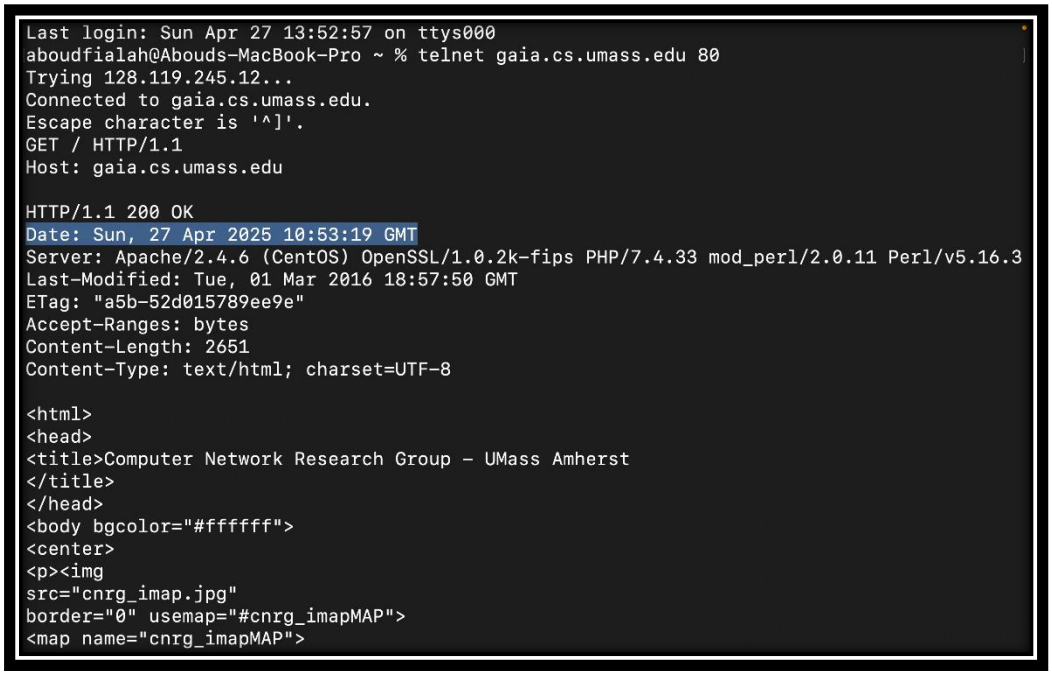
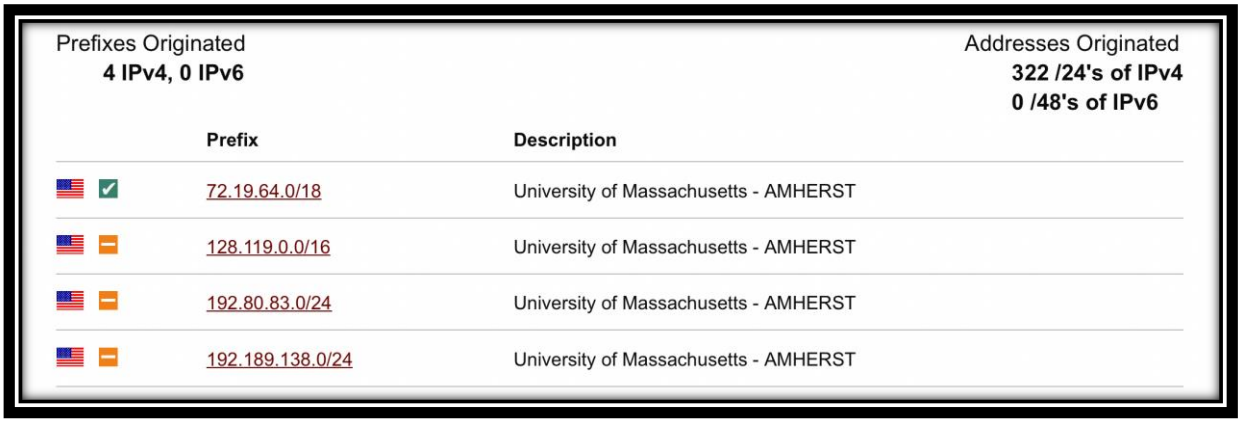


Figure 12: telnet command.

The telnet gaia.cs.umass.edu 80 commands successfully established a connection to the web server at IP address 128.119.245.12, after which a manual HTTP request (GET / HTTP/1.1) was sent to retrieve the homepage. The server responded with a 200 OK status code and returned its HTTP headers. The response included the HTML source code of the "Computer Network Research Group" website from UMass Amherst, containing the page title, a background image (cnrg_imap.jpg), and an image map for navigation. This test confirms the web server is operational and exposes technical details about its configuration, demonstrating how raw HTTP requests work.

2.1.7 details about the autonomous system (AS):

Provide details about the autonomous system (AS) number, number of IP addresses, prefixes, peers, and the name of Tier 1 ISP(s) associated with gaia.cs.umass.edu. Hint, you can use any online tool (e.g., www.bgpview.io or https://bgp.tools/).



Prefixes Originated
4 IPv4, 0 IPv6

Addresses Originated
322 /24's of IPv4
0 /48's of IPv6

Prefix	Description
72.19.64.0/18	University of Massachusetts - AMHERST
128.119.0.0/16	University of Massachusetts - AMHERST
192.80.83.0/24	University of Massachusetts - AMHERST
192.189.138.0/24	University of Massachusetts - AMHERST

Figure 13: IP Addresses and prefixes.

The figure above displays the IPv4 network information for the University of Massachusetts - Amherst. It shows the university's allocated IP address ranges, known as prefixes, which define their network space.

First, the IPv4 addresses listed here represent the university's public IP allocations. These are the addresses used for servers, devices, and services connected to the internet, including the host gaia.cs.umass.edu (128.119.245.12).

Next, the prefixes (such as 128.119.0.0/16) indicate how these IP addresses are grouped into larger blocks. The /16, /18, and /24 notations determine the size of each network segment, with smaller numbers meaning larger IP ranges.

The Figure displays the autonomous system number for the University of Massachusetts Amherst's network. This unique identifier allows the university to manage its own internet route and connectivity.

```
ASHandle:       AS1249
OrgID:          UNIVER-6
ASName:         FIVE-COLLEGES-AS
ASNumber:       1249
RegDate:        1991-04-18
Updated:        2024-11-06
TechHandle:     PG138-ARIN
Source:         ARIN
```

Figure 14:ASNumber.

This Figure lists the Tier 1 ISPs connected to the University of Massachusetts Amherst's network (AS1249). Tier 1 ISPs are the backbone of the internet, providing global connectivity without paying for transit.

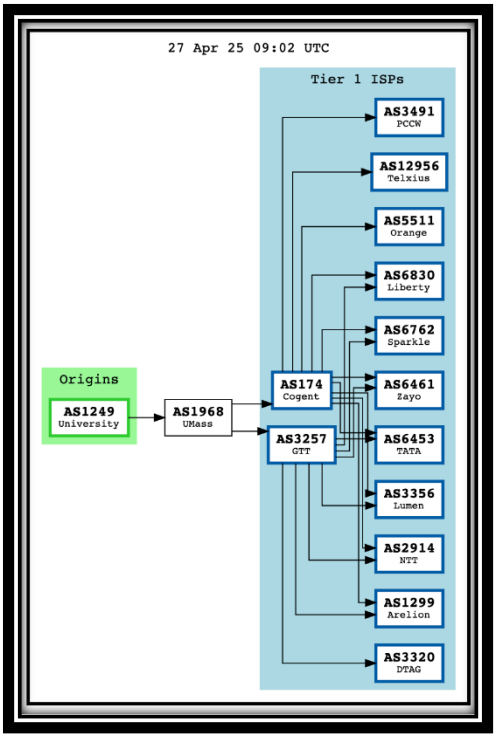


Figure 16: Tier 1 ISPs.

Peers

	ASN	Description	IPv4	IPv6
	AS1968	UMASSNET	✔	✔
	AS168	University of Massachusetts - AMHERST	✘	✔

Figure 15:Peers

The figure shows peering connections for UMass Amherst's network. Peering is when two networks directly exchange traffic instead of routing through third parties. Here, AS1968 (UMASSNET) peers with both IPv4 and IPv6, while another UMass network (AS168) only supports IPv6 peering. This setup improves connectivity and reduces costs by enabling direct data exchange between the university's own networks and partners.

2.1.8 Wireshark:

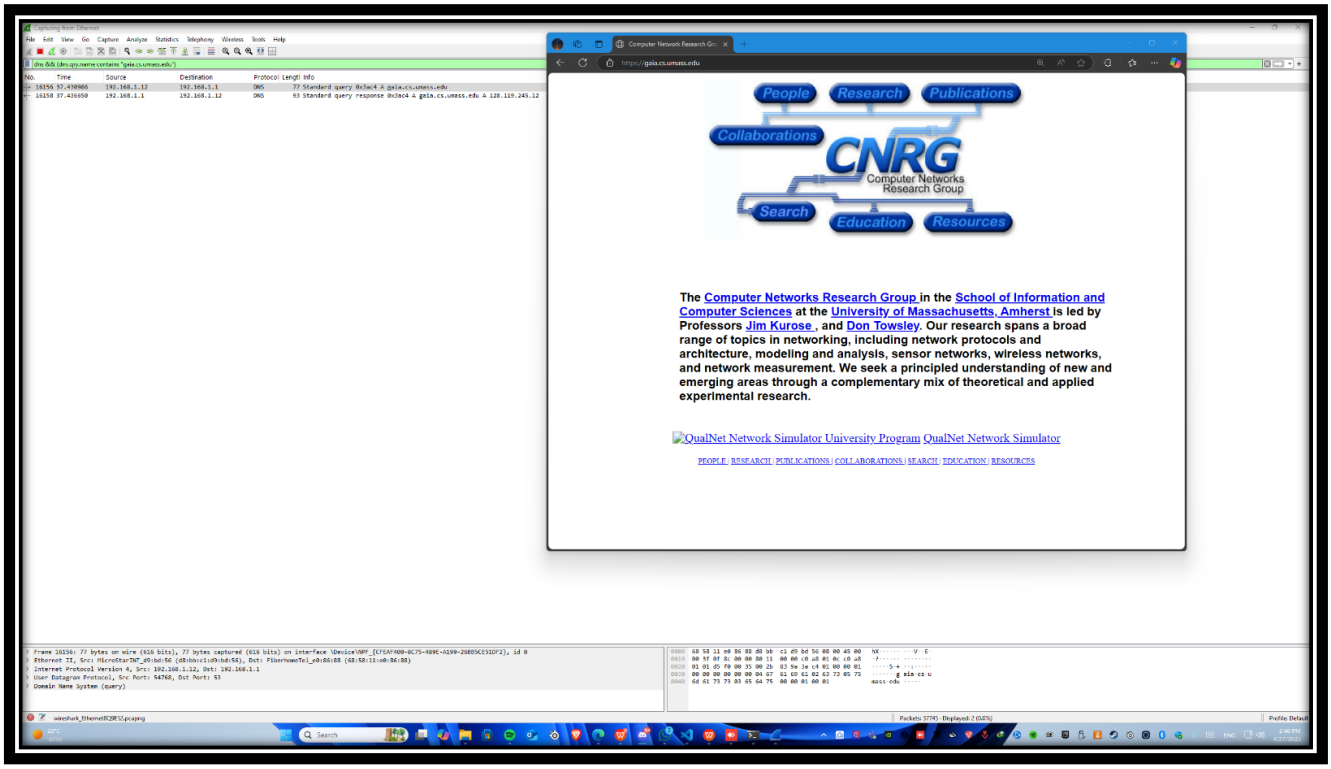


Figure 19:Wireshark.

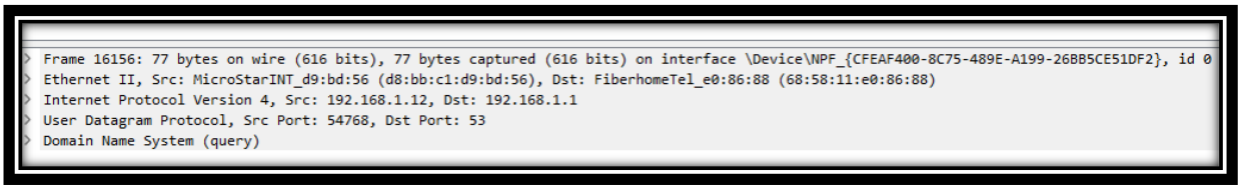
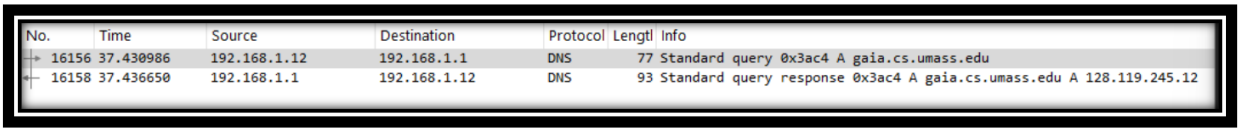


Figure 18:UDP info



No.	Time	Source	Destination	Protocol	Lengtl	Info
16156	37.430986	192.168.1.12	192.168.1.1	DNS		77 Standard query 0x3ac4 A gaia.cs.umass.edu
16158	37.436650	192.168.1.1	192.168.1.12	DNS		93 Standard query response 0x3ac4 A gaia.cs.umass.edu A 128.119.245.12

Figure 17: Request and Response.

Use **Wireshark** to capture a DNS query and response for gaia.cs.umass.edu.

The figure above shows how we managed to request using Web Browser.

The Wireshark capture shows a DNS query where my computer (192.168.1.12) asks my local DNS server (192.168.1.1) for the IP address of "gaia.cs.umass.edu". The DNS server quickly responds (in about 5.6ms) with the correct IPv4 address 128.119.245.12. This exchange uses UDP port 53, the standard port for DNS traffic. The successful resolution confirms that the domain name is properly configured in the DNS system and that my local network's DNS service is working correctly.

2.2 Task 2:

2.2.1 Socket Setup:

To set up the socket we import the socket library and in the following figure is the code to setup.

```
def run_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, __value: 1)
    server_socket.bind((HOST, PORT))
    server_socket.listen(5)
    print(f"Server listening on {HOST}:{PORT}")
    while True:
        client_socket, client_address = server_socket.accept()
        handle_client(client_socket, client_address)
```

Figure 20:socket setup code.

we initialize the socket using:

server_socket = socket. socket (socket.AF_INET, socket. SOCK_STREAM)

here the AF_INET Specifies the use of IPv4 addressing and the SOCK_STREAM Sets up a TCP connection for reliable data transfer

for the bind used to bind the server to a specific IP address and port using the Server_socket. Bind.

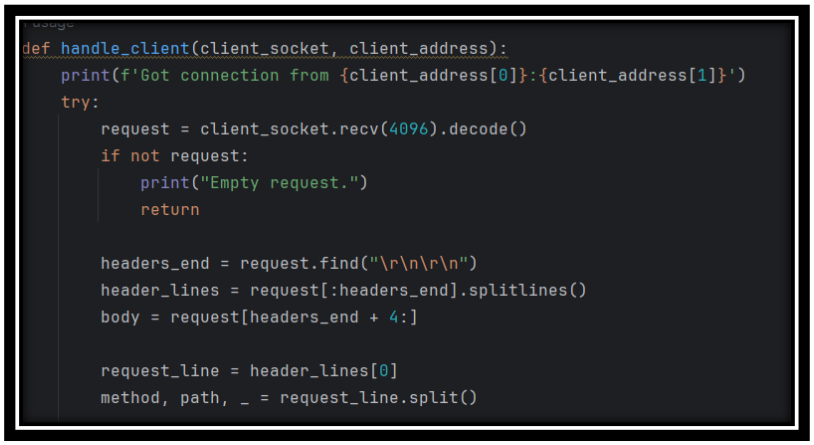
server_socket. bind (HOST, PORT))

```
HOST = "0.0.0.0"            #IP address that listens to all available networks
PORT = 9926                 #PORT Number (Aboud Fialah Student ID: 1220216)
```

Figure 21:IP Address and PORT number

Then using server_socket. listen (5) the server listens for incoming connections.

- **Request Handling**:



```
def handle_client(client_socket, client_address):
    print(f'Got connection from {client_address[0]}:{client_address[1]}')
    try:
        request = client_socket.recv(4096).decode()
        if not request:
            print("Empty request.")
            return

        headers_end = request.find("\r\n\r\n")
        header_lines = request[:headers_end].splitlines()
        body = request[headers_end + 4:]

        request_line = header_lines[0]
        method, path, _ = request_line.split()
```

Figure 22: handle the request code.

To handle any request, we have two types of request POST and GET request, the first thing is to accept the incoming client connections all the time using accept ().

The server receives HTTP requests from the client using:

- **Receiving Requests:**

  request = client_socket. recv (4096). Decode ()

  The number 4096 is the buffer size in bytes for incoming data, and then the reserved data from the request is decoded from bytes to a string using decode ().

- **Analysis the requests:**

  To analysis the requests we start by finding the request headers end and then split the line of the request and take the body of the request.

  After that we went to know the request type using these two lines

  request_line = header_lines[0]

  method, path, _ = request_line.split()

  Method will be like GET or POST and the path will be like /main_en.html

To handle the different Request Types like HTML files, CSS files, Videos and Images the server identifies the type of resource requested.

```
if method.upper() == "POST":
    form_data = parse_qs(body)# 1220808 122175 1220216
    filename = form_data.get("filename", [""])[0]
    filetype = form_data.get("filetype", [""])[0]
    print(filename)
    if filetype and filename:
        if filetype == "image":
            try:
                filename = filename.replace("/img/", "")
                print(filename)
                with open(f"./templates/img/{filename}", "rb") as f:
                    content = f.read()
                response_headers = (
                    "HTTP/1.1 200 OK\r\n"
                    "Content-Type: image/png\r\n"
                    f"Content-Length: {len(content)}\r\n"
                    "\r\n"
                ).encode()
                print(request)
                client_socket.sendall(response_headers + content)
                return
            except FileNotFoundError:
                response_body = (
                    f"<h1>{filetype.capitalize()} '{filename}' not found.</h1>"
                    f"<p>Click <a href='https://www.google.com/search?q={filename}+{filetype}'>here</a> to search on Google.</p>"
                )
        elif filetype == "video":
            try:
                with open(f"./templates/videos/{filename}", "rb") as f:
                    content = f.read()
                response_headers = (
                    "HTTP/1.1 200 OK\r\n"
                    "Content-Type: video/mp4\r\n"
                    f"Content-Length: {len(content)}\r\n"
                    "\r\n"
                ).encode()
                print(request)
                client_socket.sendall(response_headers + content)
                return
            except FileNotFoundError:
                response_body = (
                    f"<h1>{filetype.capitalize()} '{filename}' not found.</h1>"
                    f"<p>Click <a href='https://www.google.com/search?q={filename}+{filetype}'>here</a> to search on Google.</p>"
                )
```

Figure 23:post request for images and video code.

We have the next two steps to serve the client:

- **Dynamic Files Serving:**

  In this part the server r identifies the requested resource path, files are opened in the appropriate mode.

- **Sending Files:**

  After The server identifies the requested resource path, the server sends a response header (like HTTP/1.1 200 OK) followed by the file content using client_socket.send().

To load the Images and Videos we used the next format:

Here we can see that if the method is post, the first thing we do is take the file name and the file type

filename = form_data.get("filename", [""])[0]

filetype = form_data.get("filetype", [""])[0]

After that we checked the file type if its image open the image from the directory then give

```
response_headers = (
    "HTTP/1.1 200 OK\r\n"
    "Content-Type: image/png\r\n"
    f"Content-Length: {len(content)}\r\n"
    "\r\n"
).encode()
```

Then using client_socket.sendall(response_headers + content) we sent the response headers to the client

To handle a file request, we can see the following figure in the code:

```
if method.upper() == "GET":
    path = path.rstrip("?")
    if path in ["/", "/en", "/index.html", "/main_en.html"]:
        file_path = "./templates/main_en.html"
    elif path in ["/ar", "/main_ar.html"]:
        file_path = "./templates/main_ar.html"
    elif path == "/mySite_STDID_en.html":
        file_path = "./templates/mySite_1220175_en.html"
    elif path == "/mySite_STDID_ar.html":
        file_path = "./templates/mySite_1220175_ar.html"
    elif path == "/css/main_en.css":
        file_path = "./css/main_en.css"
    elif path == "/css/main_ar.css":
        file_path = "./css/main_ar.css"
    elif path == "/css/supporting":
        file_path = "./css/supporting.css"
    else:
        file_path = path
    if file_path:
```

Figure 24: check the path.

Here we checked the file type and then took the path of it.

- CSS: we searched for the CSS file see the next figure, if the file exists return a response with 200 OK and the Content-Type: text/css, if the file does not exist return a response with HTTP/1.1 404 Not Found and Content-Type: text/css.

```
if file_path.endswith(".css"):#1220808
    try:
        with open(f"./templates{path}", "r") as f:
            content = f.read()
        response = (
            "HTTP/1.1 200 OK\r\n"
            "Content-Type: text/css\r\n"
            f"Content-Length: {len(content.encode())}\r\n"
            "\r\n"
            f"{content}"
        )
        client_socket.sendall(response.encode())
    except FileNotFoundError:
        response_body = "<h1>CSS file not found.</h1>"
        response = (
            "HTTP/1.1 404 Not Found\r\n"
            "Content-Type: text/css\r\n"
            f"Content-Length: {len(response_body.encode())}\r\n"
            "\r\n"
            f"{response_body}"
        )
        client_socket.sendall(response.encode())
    return
```

Figure 25:handle CSS request file.

- HTML: to open the html file, we searched the directory for that file if the file exists then we return a response with HTTP/1.1 200 OK and Content-Type: text/html then using client_socket.sendall(response.encode()) we sent the response to the client to open the html after that the client start sending a request for the images and the CSS file.

- Images: for the images on the page, we used the following code to handle the request.

```
if file_path.endswith(".png"):#1220808
    try:
        file_path = file_path.replace( __old: "/img/",  __new: "")
        file_path = unquote(file_path)
        print(file_path)
        with open(f"./templates/img/{file_path}", "rb") as f:
            content = f.read()
        response_headers = (
            "HTTP/1.1 200 OK\r\n"
            "Content-Type: image/png\r\n"
            f"Content-Length: {len(content)}\r\n"
            "\r\n"
        ).encode()
        print(request)
        client_socket.sendall(response_headers + content)
        return
    except FileNotFoundError:
        response_body = "<h1>img not found.</h1>"
    response = (
        "HTTP/1.1 404 Not Found\r\n"
        "Content-Type: image/png\r\n"
        f"Content-Length: {len(content)}\r\n"
        "\r\n"
        f"{response_body}"
    )
    client_socket.sendall(response.encode())
    return
```

Figure 26: handle image request in the web.

- **HTML and CSS**:

  In this part we successfully complete the implementation of the main English(main_en.html), main Arabic(main_ar.html) and two supporting pages Arabic and English (mySite_1220175_en, mySite_1220175_ar), then using CSS file we mad these pages having a good user interface design.

  - The team members section with names, id's, and images which they are implemented as follow.

```
<header class="header_text">
    <h1>ENCS3320-Webserver</h1>
    <h2>Welcome to ENCS3320 - Computer Networks Webserver</h2>
</header>
<body class="body">
    <div class="part1">
        <div class="teamInFo">
            <p>Team Member</p>
            <div class="teme_member">
                <div class="card">
                    <img src="img/Alaa1220808.png" alt="" class="theimage"  method="post" action="/alaa.png">
                    <h3>Alaa Faraj</h3>
                    <h>Skills:coding(C,Java,Python) and frontend development</h>
                    <h>hobby: playing video games</h>
                </div>
                <div class="card">
                    <img src="img/Adnan1220175.png" alt="" class="theimage">
                    <h>Skills: coding(C,Java,Python) and frontend development</h>
                    <h>hobby: Gym</h>
                </div>
                <div class="card">
                    <img src="img/Aboud1220216 .png" alt="" class="theimage">
                    <h>Skills: coding(C,Java,Python) and frontend development</h>
                    <h>hobby: swimming</h>
                </div>
            </div>
        </div>
    </div>
```

Figure 27:HTML code for the team.

In the following figure we can see the team members and the header text that contains ENCS3320-Webserver and Welcome to ENCS3320 - Computer Networks Webserver.

- Here we have a clearly organized presentation of a topic from the first chapter of the textbook, about Network security and we add an image for this topic.

```
<div class="part2">
    <div class="topic_card_1">
        <h5 class="topic">Network security</h5>
    </div>
    <div class="info-card">
        <div class="topic_card">
            <h6>introduction</h6>
            <p class="introduction">The internet was originally built with little regard for security, based on a trust model of friendly users on an open network.
                When threats became apparent, protocol designers were compelled to adapt and add security measures at every layer.
                Today, it is crucial to focus on understanding how attackers can exploit networks, how to defend against attacks, and how to design network
                architectures that are secure and resilient by nature </p>
        </div>

        <div class="topic_card">
            <h6>Bad guys: Malware</h6>
            <p>Malware can infect a host in different ways:</p>
            <ul class="Bad-guys-type">
                <li><b>Virus:</b> A self-replicating infection caused by receiving or executing an object (like an email attachment).</li>
                <li><b>Worm:</b> A self-replicating infection caused by passively receiving an object that executes itself.</li>
                <li><b>Spyware malware:</b> Records keystrokes, visited websites, and uploads information to a collection site.</li>
                <li><b>Botnet:</b> Infected hosts are enrolled and used for spam or distributed denial of service (DDoS) attacks.</li>
            </ul>
        </div>

        <div class="topic_card">
            <h6>Denial of Service (DoS)</h6>
            <p>A DoS attack happens when attackers flood a resource (like a server or bandwidth) with bogus traffic, making it unavailable for legitimate users.</p>
            <ul class="Bad-guys-type">
                <li><b>Select Target:</b> The attacker picks a server or service to attack.</li>
                <li><b>Break into Hosts:</b>They compromise multiple machines across the network (creating a botnet).</li>
                <li><b>Send Packets:</b> These compromised hosts then send massive amounts of traffic to the target, overwhelming it.</li>
            </ul>
        </div>
    </div>
    <img src="./img/Screenshot 2025-04-27 155903.png" alt="" class="topic_img">
</div>
```

Figure 28:HTML topic code.

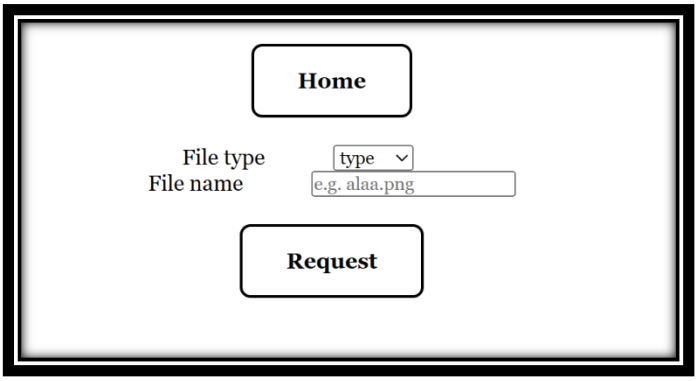
- The last thing in the main_en and main_ar we have the links for Textbook website and Birzeit website and a link to local html file mySite_1220175_en for the English version and a link to switch from the English to Arabic.

```
<footer class="footer" >
    <div class="externalLinks">
        <a href="https://gaia.cs.umass.edu/kurose_ross/index.php" target="_blank">
          <button class="btn liquid">Textbook</button>
        </a>
        <a href="https://www.birzeit.edu" target="_blank">
          <button class="btn liquid">Birzeit</button>
        </a>
        <form method="get" action="/mySite_1220175_en.html">
            <button class="btn liquid">supporting</button>
        </form>
        <form method="get" action="/main_ar.html">
            <button class="btn liquid">Arabic version</button>
        </form>

    </div>
```

Figure 29:HTML links code.

After implementing the main page, we implement the supporting page (mySite_1220175_en, mySite_1220175_ar), that contains a form to search for an image or a video, we can choose the type of data and then search in the figure below we can see the form and the html code.




```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="./supporting.css">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Request File</title>
</head>
<body>
    <header class="header">
        <div>
            <form method="get" action="http://localhost:9926/main_en.html">
                <button class="btn liquid" type="submit">Home</button>
            </form>
        </div>
    </header>

    <div class="container">
        <form id="requestForm" method="post" action="http://localhost:9926" target="_blank">
            <div class="div_input">
                <label for="filetype">File type</label>
                <select name="filetype" id="filetype" class="form-control" required>
                    <option value="" disabled selected>type</option>
                    <option value="image">Image</option>
                    <option value="video">Video</option>
                </select>
                <i class="zmdi zmdi-caret-down" style="font-size: 17px"></i>
            </div>

            <div class="div_input">
                <label for="filename">File name</label>
                <input type="text" name="filename" id="filename" placeholder="e.g. alaa.png" required>
            </div>

            <div class="div_input">
                <button class="btn liquid" type="submit">Request</button>
            </div>
        </form>
    </div>
</body>
<footer>
</footer>
</html>
```

Figure 30:search form code.

The supporting page check if the image or video is on the main page if the image or video are available the web will open the data that we searched for it in this case we have two possible answers from the supporting page if the data exists or not.

- The data exists:

  Here we need to the type of the searched data if it image or video then in the python code (server side) see the following figure the code check if the data in our web site or not if it exists the server will return a response with HTTP/1.1 200 OK and Content-Type: image/png as in the figure below.

```python
filename = filename.replace("/img/", "")
print(filename)
with open(f"./templates/img/{filename}", "rb") as f:
    content = f.read()
response_headers = (
    "HTTP/1.1 200 OK\r\n"
    "Content-Type: image/png\r\n"
    f"Content-Length: {len(content)}\r\n"
    "\r\n"
).encode()
print(request)
client_socket.sendall(response_headers + content)
return
```

Figure 31:the data exists response.

- The data does not exist:
  If the requested file is unavailable the server response with the following format of the response headers for the image and the same for the video:

```
except FileNotFoundError:
    redirect_url = f"https://www.google.com/search?tbm=isch&q={filename}"
    response_headers = (
        "HTTP/1.1 307 Temporary Redirect\r\n"
        f"Location: {redirect_url}\r\n"
        "Content-Length: 0\r\n"
        "Connection: Close\r\n"
        "\r\n"
    ).encode()
    client_socket.sendall(response_headers)
    print(request)
    return
```

Figure 32: the data (image) dose not exists response.

Here if the file is not found in the try section, except FileNotFoundError: we constructed google image search URL that take the entered image name and concatenate it with the URL to provide a ready link.

Then create the HTTP response and tells the client that the image or the video is not found and redirect to a different URL.

The same happens with the video but with different URL:

```
except FileNotFoundError:
    redirect_url = f"https://www.google.com/search?tbm=vid&q={filename}"
    response_headers = (
        "HTTP/1.1 307 Temporary Redirect\r\n"
        f"Location: {redirect_url}\r\n"
        "Content-Length: 0\r\n"
        "Connection: Close\r\n"
        "\r\n"
    ).encode()
    client_socket.sendall(response_headers)
    print(request)
    return
```

Figure 33: the data (video) does not exists response.

- **Test the code and run the cases:**

  - **Load the HTML web with all the obj:**

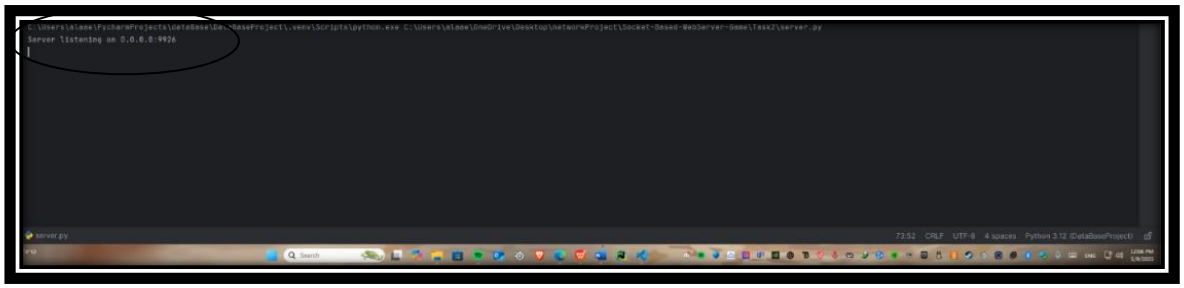


Figure 34:start the server.

When we start the server, we will have the following message in the terminal.

- Here we see that the server starts to lessen on the **address 0.0.0.0** and on port **9926**. After that when we enter the browser **localhost:9926** a request to the main_en.html will be generated in the following figure we can see the paths of main_en.html.

```
if path in ["/", "/en", "/index.html", "/main_en.html"]:
    file_path = "./templates/main_en.html"
```

Figure 35: main_en.html paths.

1. **On the same pc:**

To see the response to the request see the figure below this request.

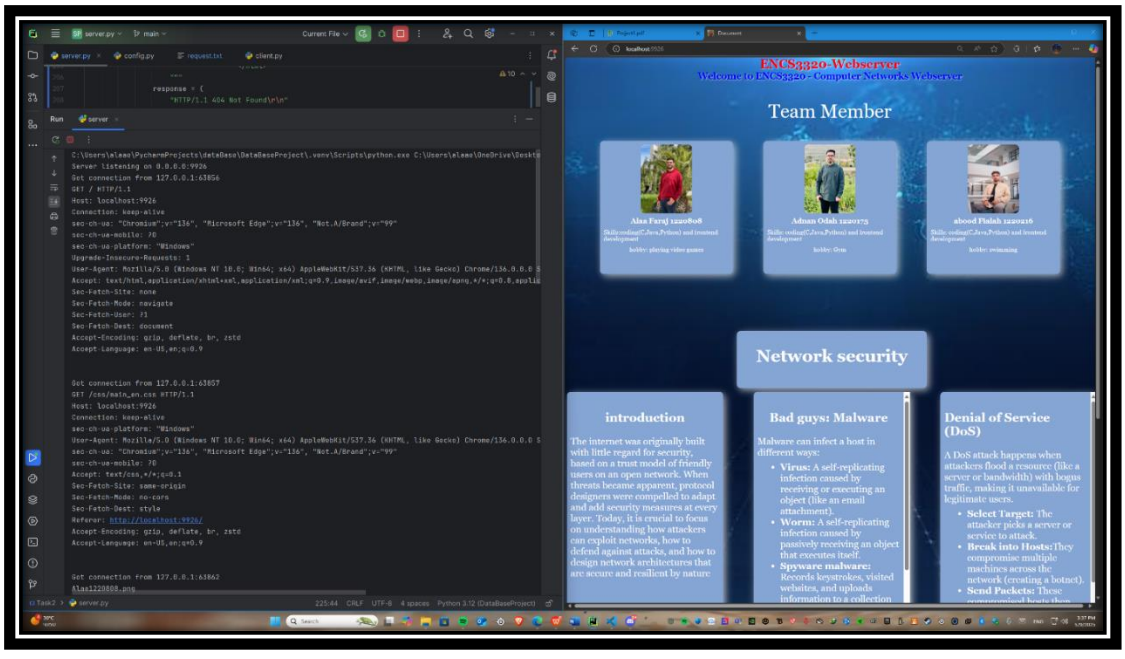


Figure 36:open main_en.

❖ To see the response closely:

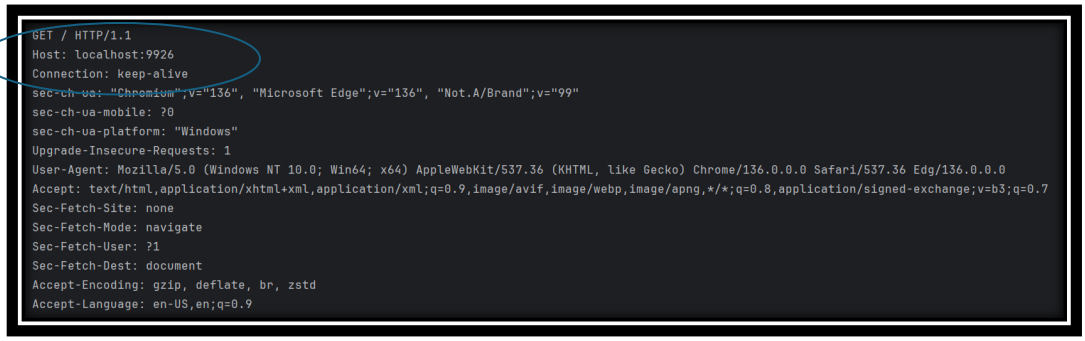


Figure 37: html file request

❖ Now it has started requesting all the file on the page:

Figure 38: CSS file request.

❖ Now the client has started requesting the images on the web page as follows:

I.Alaa Faraj image :

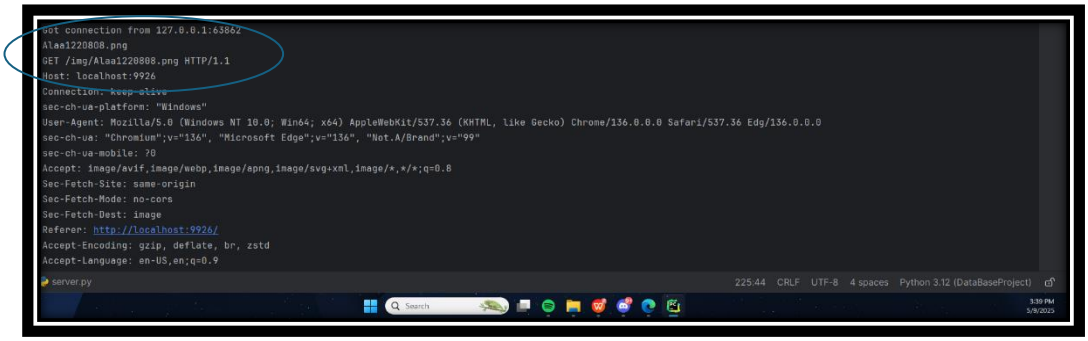


Figure 39: request Alaa Faraj image

II.Adnan Odeh image:

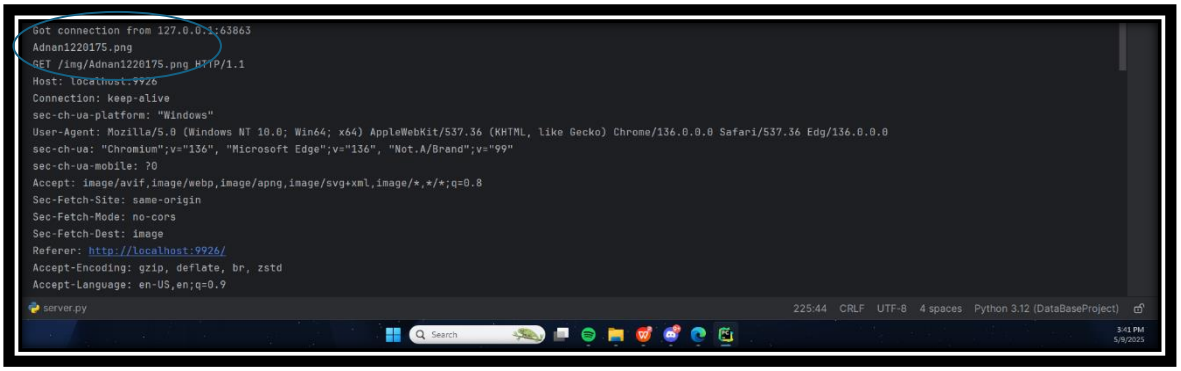


Figure 40: request Adnan Odeh image

III. Abood Fialah image:

Figure 41: request Abood Fialah image.

IV. The topic image:

Figure 42: request topic image.

V. The background image:

Figure 43: request Background image.

In our code we took the file name from the request message and searched for it, if it exists on our web page then the server sent it to the client.

- **on different device (mobile):**

  here we open the web site on different device as shown in the next figure the web successfully loads and all the element, we managed to do that by using the IP address of the pc that the server run on it and use the PORT number **9926**

Figure 44:main_en.html on different device

Here we can see that the server got a connection from 192.168.1.6:62223 and this is IP Address and the porn number of the mobile

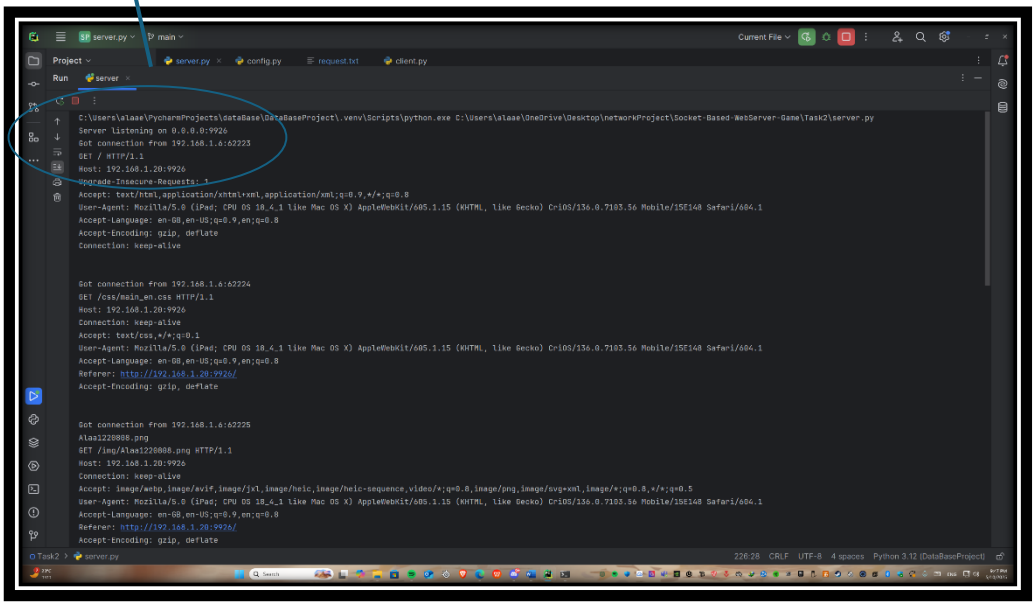


Figure 45:element request and page.

- **Supporting page Sample Run(**mySite_1220175_en**):**

  After running the main page as the previous sample run, we start testing the supporting page mySite_1220175_en, in this page we will insert in the form below some the type of the data (image or video).

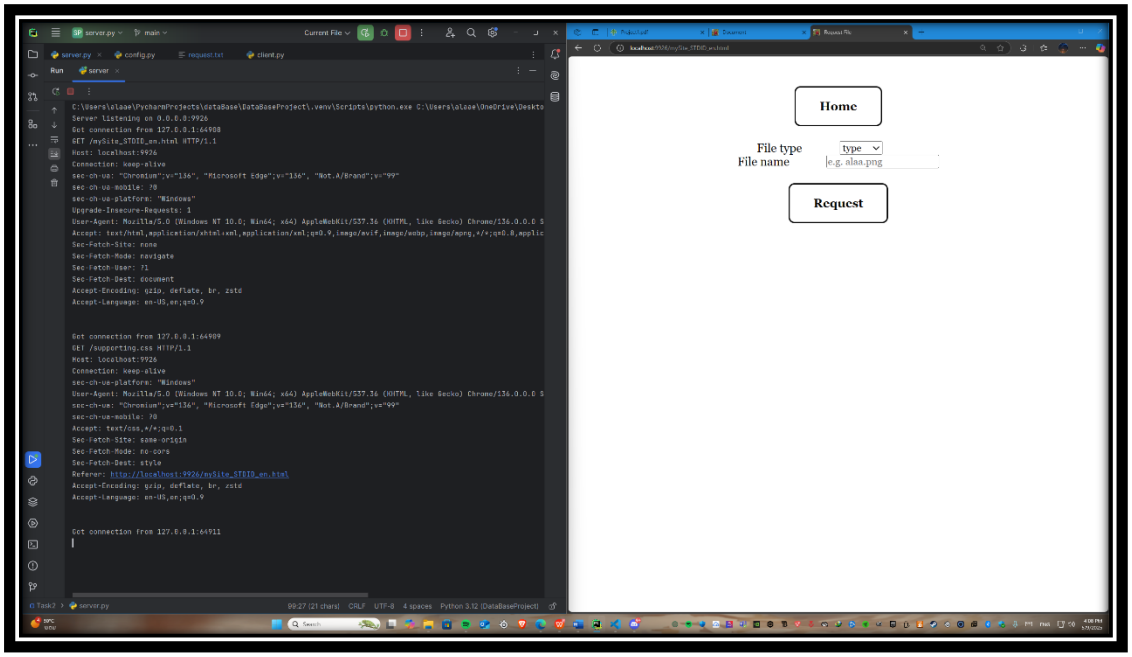


Figure 46: request mySite_1220175_en.

In figure 25 we can see the request to the mySite_1220175_en page, to test this page we will search for an image for example we will search for Alaa1220808.png and for an unknown image like book.png.

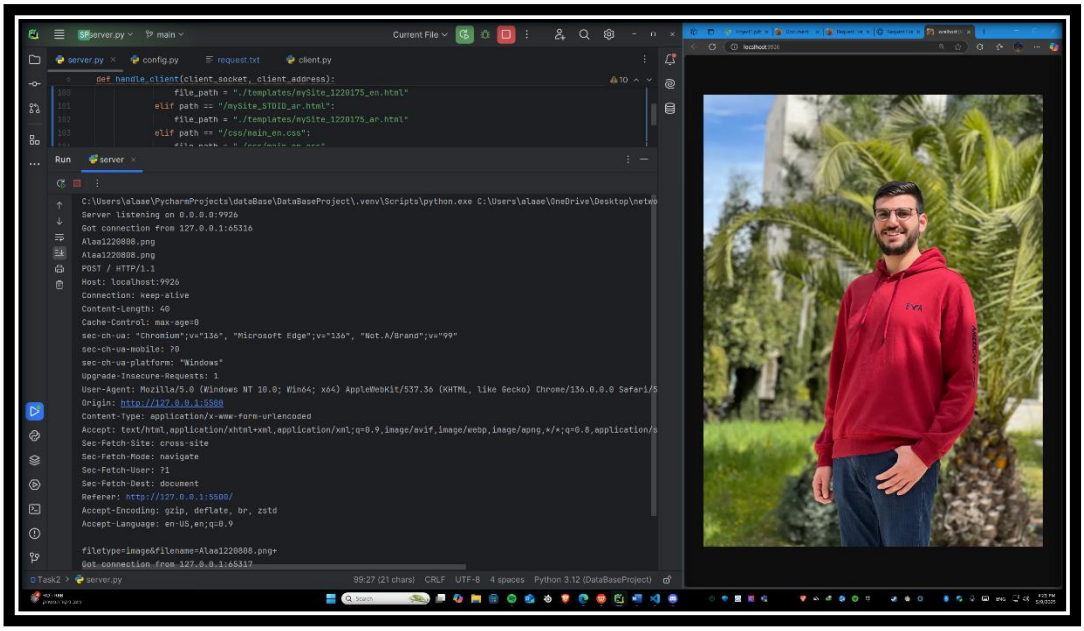


Figure 47: request Alaa1220808.png

By looking at figurer 26 we request the image, the server will chick if the obj is found or not, the output was successfully shown as in figurer 26.

Now we searched for an unknown image, for example book.png.

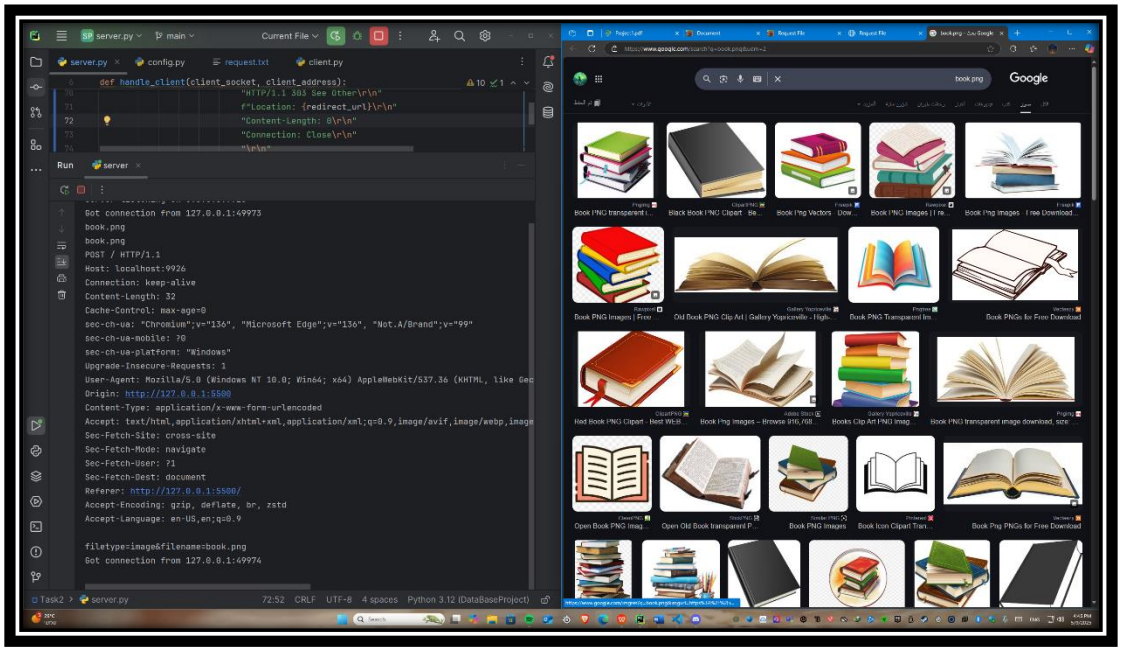


Figure 48: request book.png.

Here the server does not find the image, so the server takes me to google and search for book.png.

In case we searched for a video is the video exists the server will return the video to the client as follows:

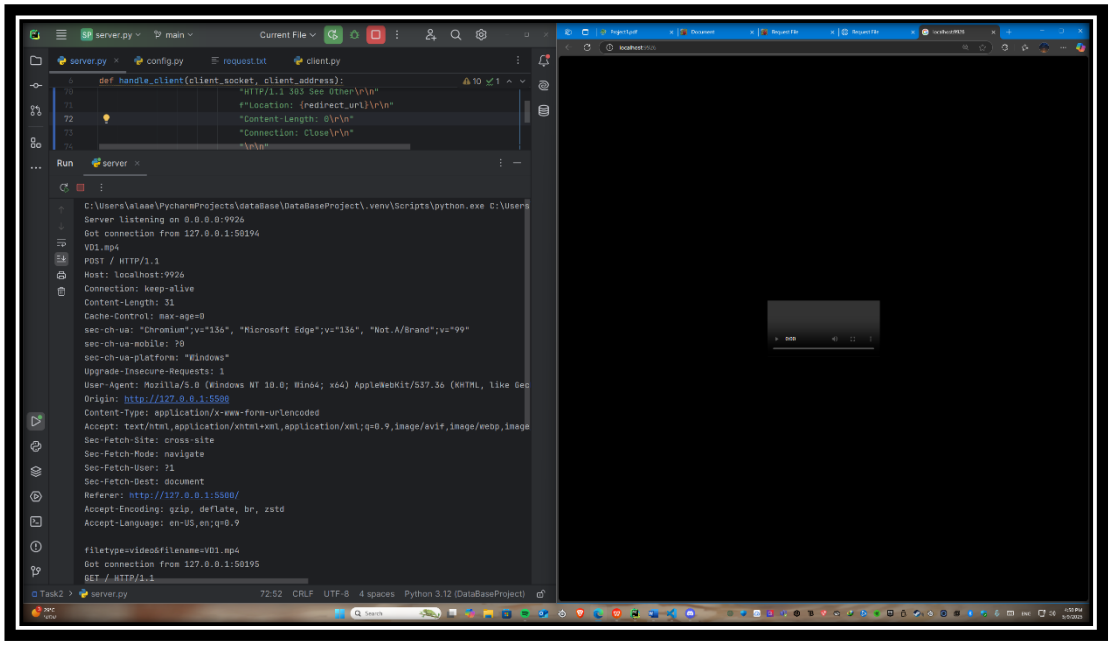


Figure 49:request a video.

And if the server does not find the video the server will do the same as he done for the image.

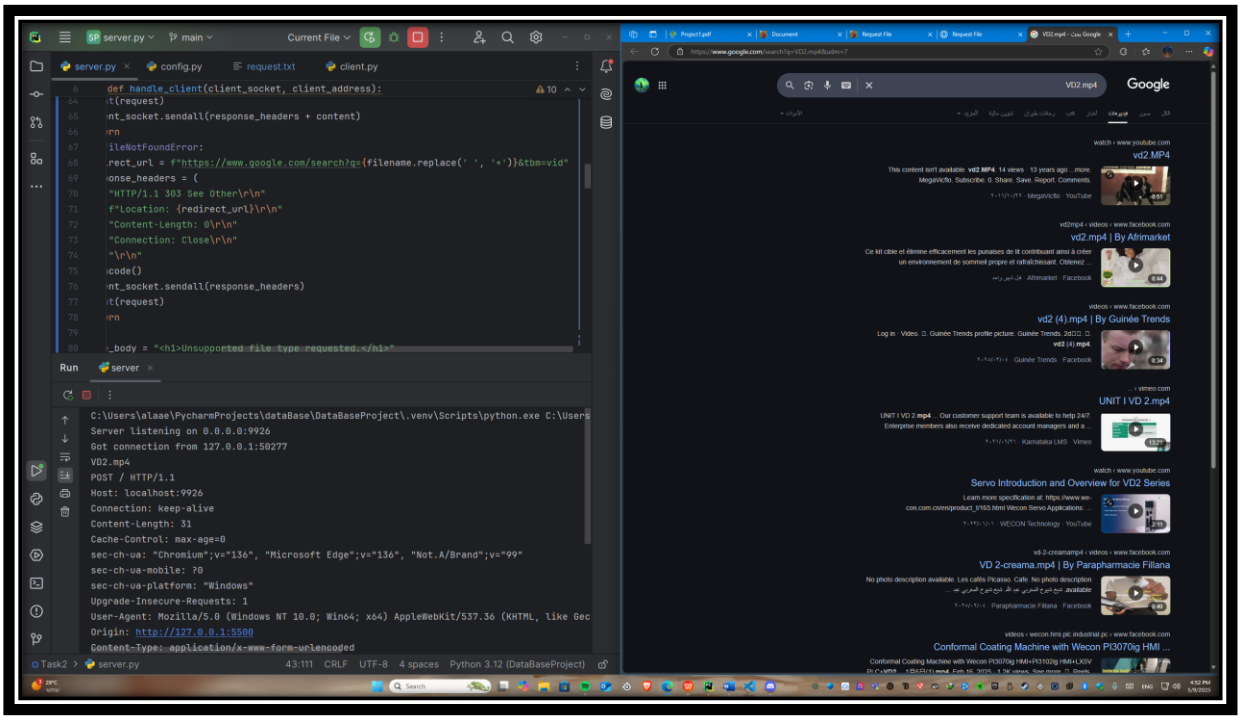


Figure 50: video does not exists.

2.3 Task 3:

2.3.1 Server Initialization:

The server starts by creating a TCP socket on port 6000 to manage client connections and player registration and it continuously waits for clients to connect the number of clients should be two players minimum and four

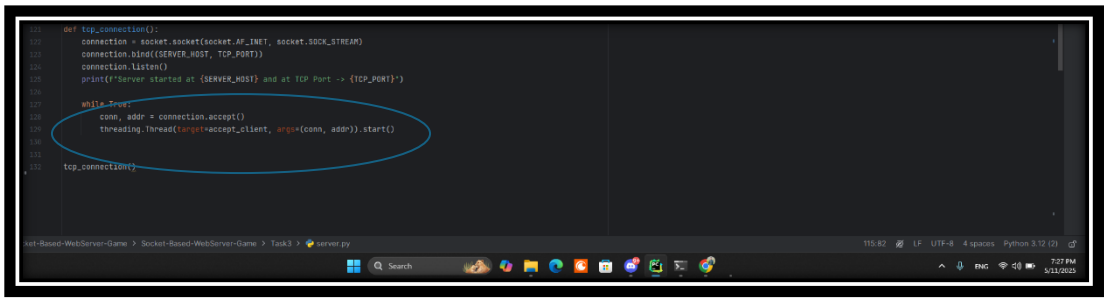


Figure 51:code to accept.

players maximum in the following graph is the code.

2.3.2 Handling Client Registration (TCP):

To handle Inside accept_client () The server expects a message in the format:

"JOIN <username> “, It splits and validates the message to Ensures it starts with "JOIN" and has exactly one username. Then Checks if the username is unique (not already in use).

- If valid, the username is stored in a dictionary: Players[username] = (tcp_conn, udp_addr).

A TCP broadcast is sent to all players notifying them of the new player.

And here we can see that if only one player enters the game, the game want start.

We can see that a player with name Adnan join

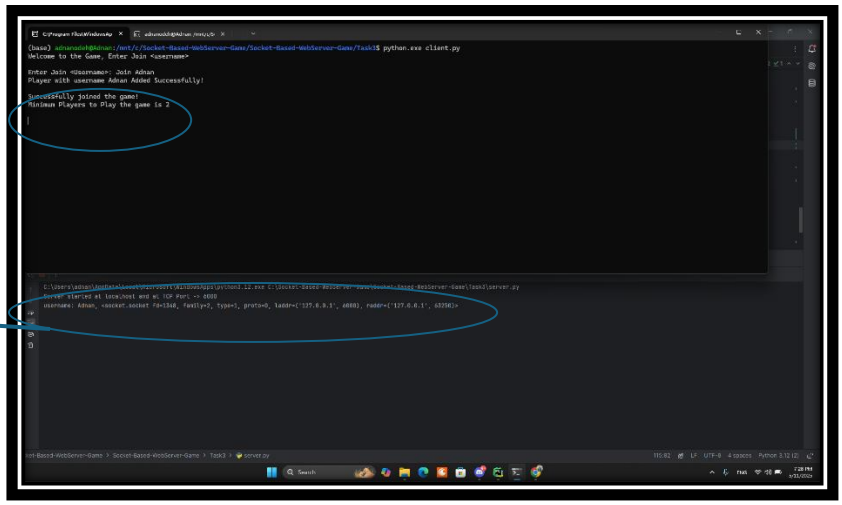


Figure 52:only one player in the game.

To start the game another player needs to join the game:

Here we can see that another player joins the game

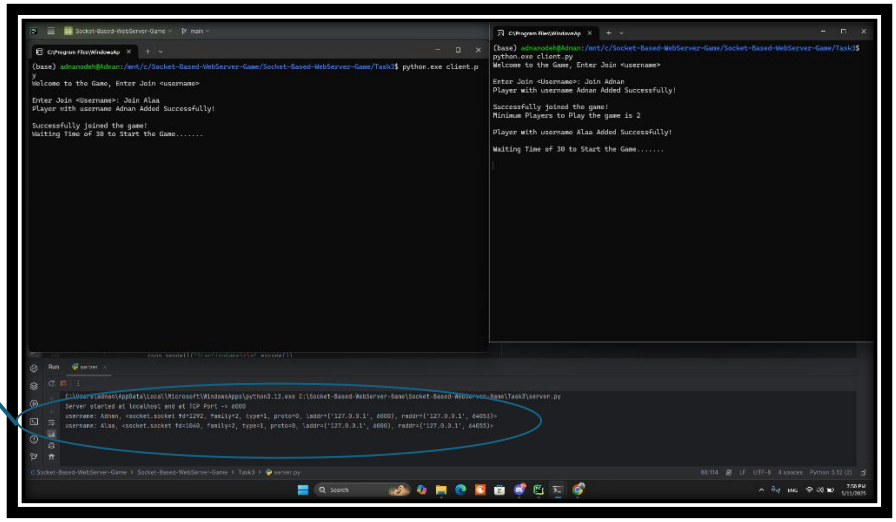


Figure 53:two players in the game.

After joining the second player the server sent to all the players that the game will start after 30 seconds, and the timer will reset after every joint.

In the following figure we can see the max number of players:

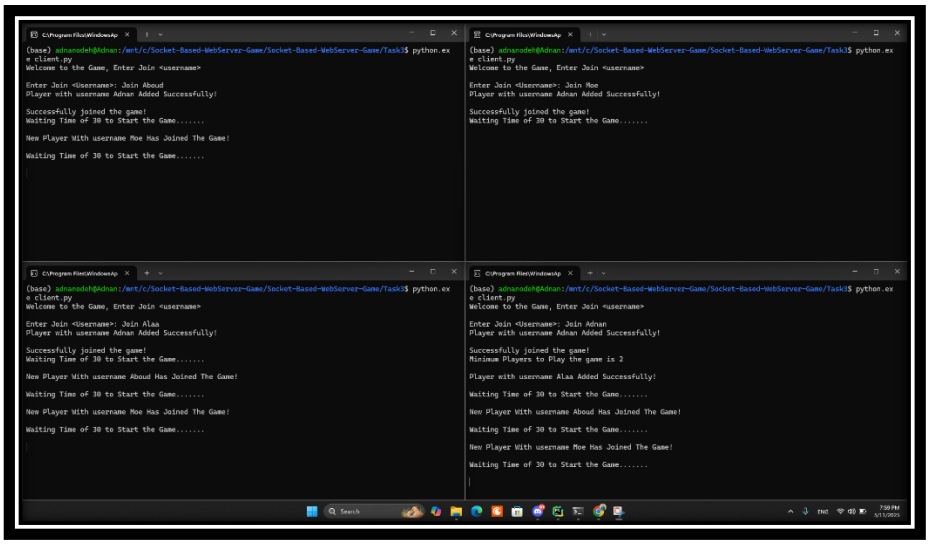


Figure 54:max number of players.

To see the server terminal look at the following figure:

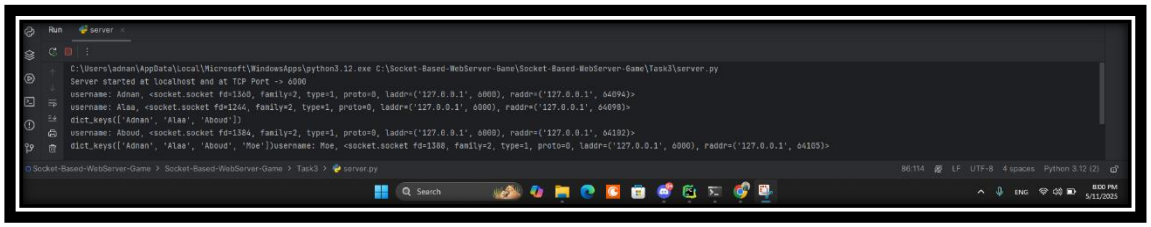
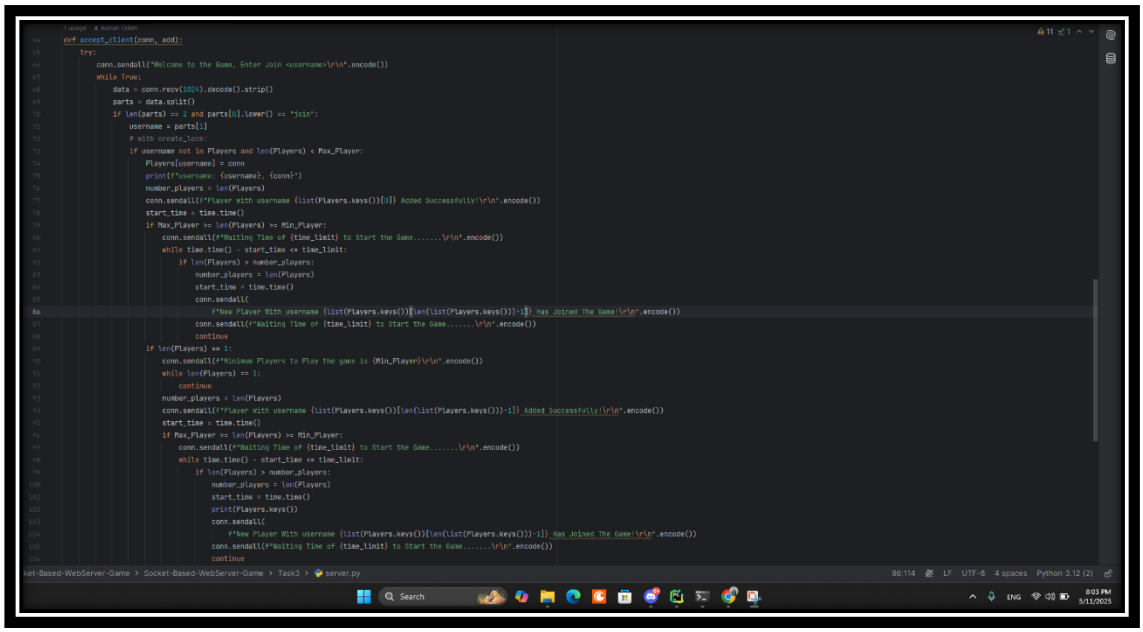
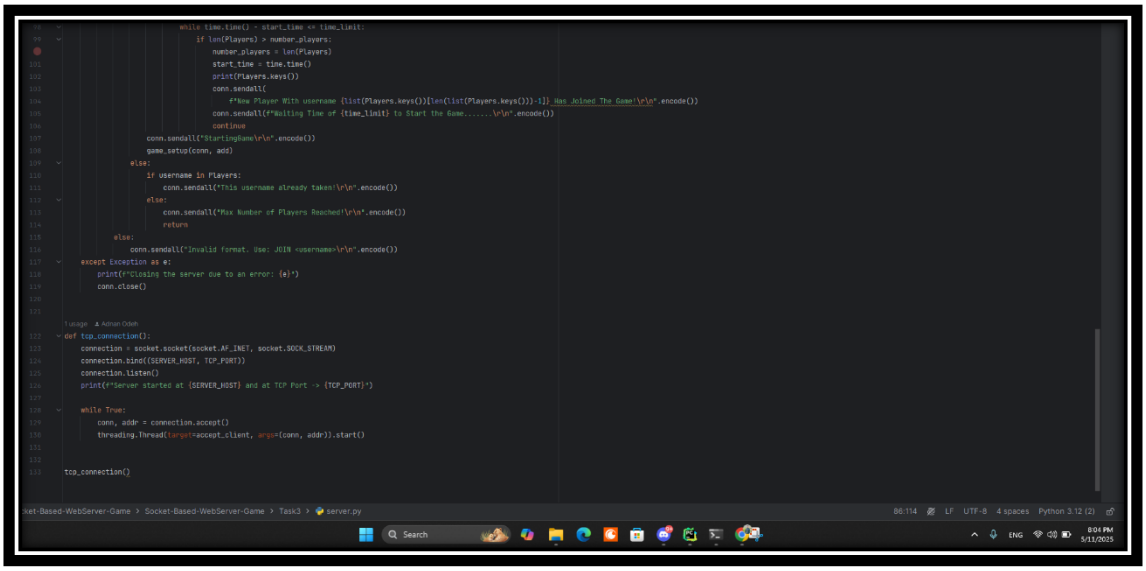


Figure 55:server terminal.

And here we can see the code to accept the client:



```
def accept_client(conn, addr):
    try:
        conn.sendall("Welcome to the Game, Enter Join <username>\r\n".encode())
        while True:
            data = conn.recv(1024).decode().strip()
            parts = data.split()
            if len(parts) == 2 and parts[0].lower() == "join":
                username = parts[1]
                # with create_lock:
                if username not in Players and len(Players) < Max_Player:
                    Players[username] = conn
                    print(f"username: {username}, {conn}")
                    number_players = len(Players)
                    conn.sendall(f"Player with username {list(Players.keys())[0]} Added Successfully!\r\n".encode())
                    start_time = time.time()
                    if Max_Player >= len(Players) >= Min_Player:
                        conn.sendall(f"Waiting Time of {time_limit} to Start the Game.....\r\n".encode())
                        while time.time() - start_time <= time_limit:
                            if len(Players) > number_players:
                                number_players = len(Players)
                                start_time = time.time()
                                conn.sendall(
                                    f"New Player With username {list(Players.keys())[len(list(Players.keys()))-1]} Has Joined The Game!\r\n".encode())
                                conn.sendall(f"Waiting Time of {time_limit} to Start the Game.....\r\n".encode())
                                continue
                    if len(Players) == 1:
                        conn.sendall(f"Minimum Players to Play the game is {Min_Player}\r\n".encode())
                        while len(Players) == 1:
                            continue
                        number_players = len(Players)
                        conn.sendall(f"Player with username {list(Players.keys())[len(list(Players.keys()))-1]} Added Successfully!\r\n".encode())
                        start_time = time.time()
                        if Max_Player >= len(Players) >= Min_Player:
                            conn.sendall(f"Waiting Time of {time_limit} to Start the Game.....\r\n".encode())
                            while time.time() - start_time <= time_limit:
                                if len(Players) > number_players:
                                    number_players = len(Players)
                                    start_time = time.time()
                                    print(Players.keys())
                                    conn.sendall(
                                        f"New Player With username {list(Players.keys())[len(list(Players.keys()))-1]} Has Joined The Game!\r\n".encode())
                                    conn.sendall(f"Waiting Time of {time_limit} to Start the Game.....\r\n".encode())
                                    continue
```



```
                            while time.time() - start_time <= time_limit:
                                if len(Players) > number_players:
                                    number_players = len(Players)
                                    start_time = time.time()
                                    print(Players.keys())
                                    conn.sendall(
                                        f"New Player With username {list(Players.keys())[len(list(Players.keys()))-1]} Has Joined The Game!\r\n".encode())
                                    conn.sendall(f"Waiting Time of {time_limit} to Start the Game.....\r\n".encode())
                                    continue
                    conn.sendall("StartingGame\r\n".encode())
                    game.setup(conn, addr)
                else:
                    if username in Players:
                        conn.sendall("This username already taken!\r\n".encode())
                    else:
                        conn.sendall("Max Number of Players Reached!\r\n".encode())
                        return
            else:
                conn.sendall("Invalid format. Use: JOIN <username>\r\n".encode())
    except Exception as e:
        print(f"Closing the server due to an error: {e}")
        conn.close()


def tcp_connection():
    connection = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    connection.bind((SERVER_HOST, TCP_PORT))
    connection.listen()
    print(f"Server started at {SERVER_HOST} and at TCP Port -> {TCP_PORT}")

    while True:
        conn, addr = connection.accept()
        threading.Thread(target=accept_client, args=(conn, addr)).start()


tcp_connection()
```

Figure 56:code to accept client.

2.3.3 Starting the Game:

Here we can see how the server deals with the client guesses the server check if the client gets the number correctly or if the entered number is higher or lower:

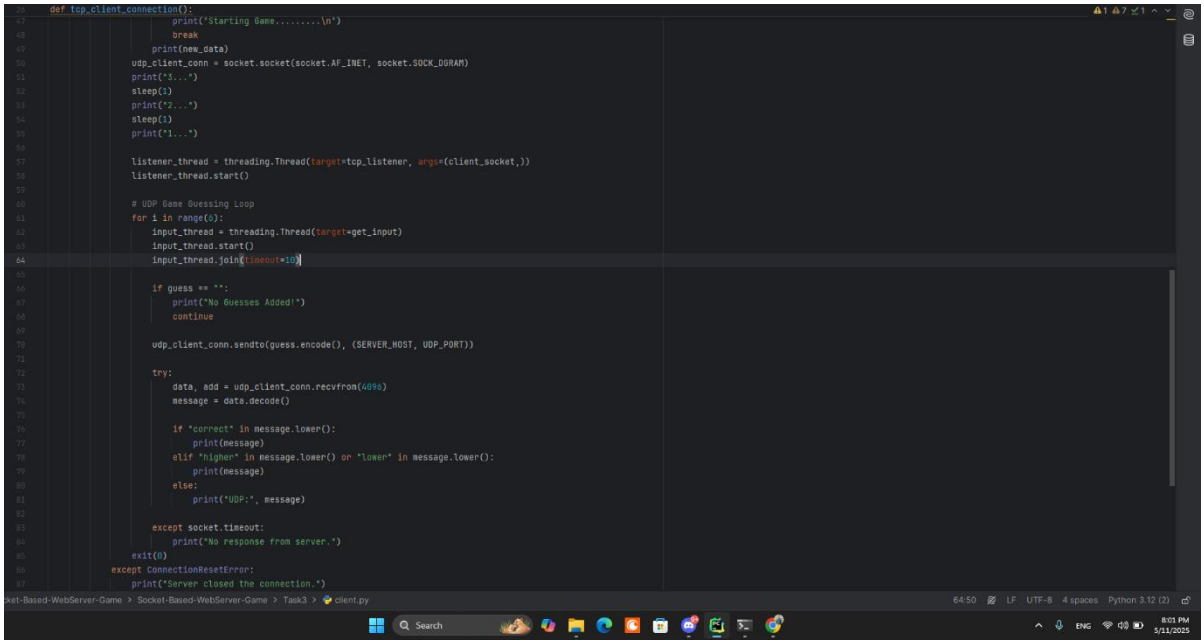


Figure 57:check the guess.

- A UDP socket is initialized on a separate port to handle gameplay.
- A secret number between 1–100 is generated.
- A game_over = False flag is used to indicate when the game ends.

Test case run:

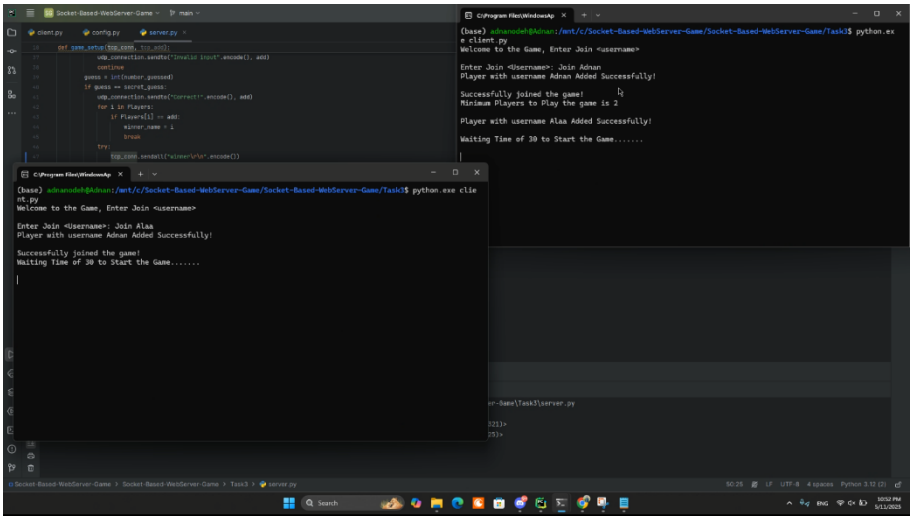


Figure 58:the test start with two client.

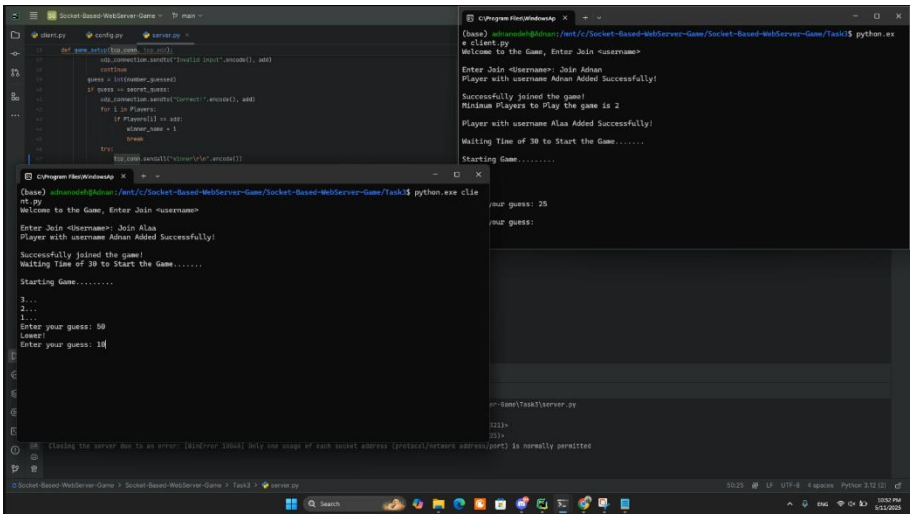


Figure 59:the client starts guessing.

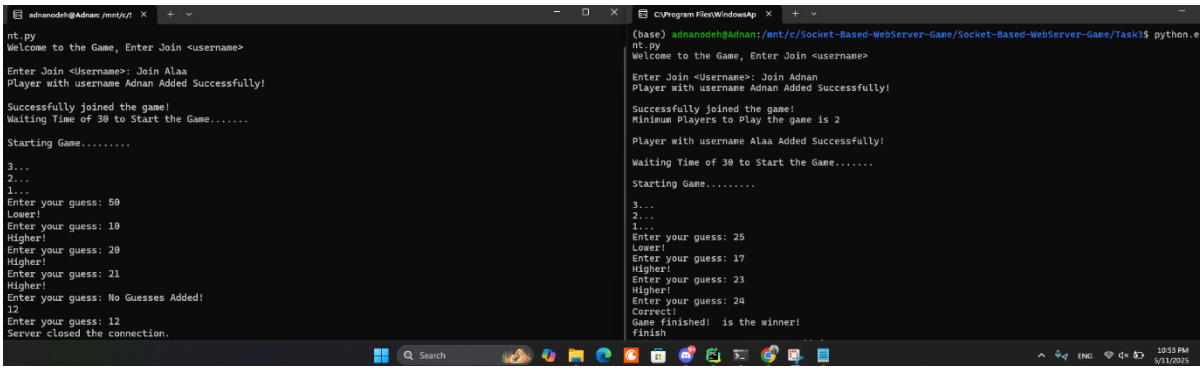


Figure 60:the game finish.

2.3.4 Gameplay Phase (UDP + TCP):

The total length of the game is 60 seconds and in this time the player starts guessing and this happened inside a loop. The server calculates elapsed time. If > 60 seconds, the game ends. Server receives a guest from a player over UDP, The server checks:

- If the guess is not a digit → respond "Invalid input" (UDP).
- If the guess is less than the secret number → respond "Higher!" (UDP).
- If the guess is more → respond "Lower!" (UDP).
- If the guess is correct: Send "Correct!" to that player (UDP).
- Identify the winner's name from the Players dictionary.

This happened in the following code:

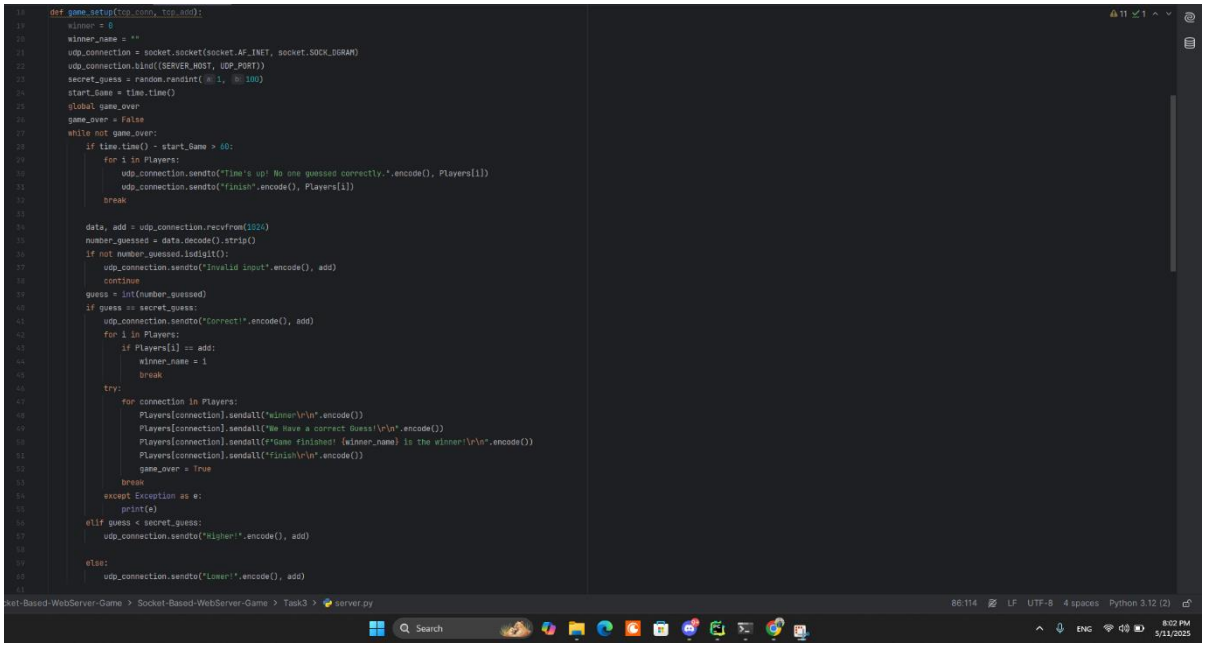


Figure 61:response to the client

By looking at the figure we can see the response that the client reserve from the server.

2.3.5 Game Ending Conditions:

The game ends under two conditions:
A correct guess is made → Server sends winner info via TCP to all clients.

60 seconds pass without a correct guess → Server sends "Time's up!" via UDP and terminates the game.

3. Challenges:

- In task one we had some challenges with Wireshark.
- In task two when we trye to open the web site on another device it does not work when we enter **localhost:9926** instead we used the IP address of the pc.
- In task three when the server sends a massage the server sends it as a stream of bytes and how to handle it in the client side.
- Also, in task three when we opened the UDP socket we let the TCP socket open, the challenge here was how to reuse the TCP socket again.

4. Conclusion:

This project gave us extensive, hands-on experience with the basic principles and practices of computer networking. We were introduced to key networking tools, socket programming concepts, and real-world applications through a set of guided labs. We began by utilizing diagnostic commands and Wireshark to analyze traffic and gain detailed knowledge of DNS resolution. Then, we were able to implement a functional web server that could serve HTML, CSS, images, and videos, and responding to dynamic GET and POST requests. The implementation showcased the ways in which TCP and UDP protocols can be used to design responsive and fault-tolerant client-server systems. Lastly, we developed and deployed a multiplayer guessing game, demonstrating real-time coordination and synchronization using a hybrid TCP/UDP strategy. Overall, this project solidified our understanding of network protocols, client-server interaction, and the practical challenges of building distributed systems. It also enhanced our debugging, collaboration, and system design capabilities—preparing us for advanced topics in computer networks and backend development.

5. Resources:

[1] https://www.ir.com/guides/network-diagnostics
[2] https://www.comptia.org/content/articles/what-is-wireshark-and-how-to-use-it
[3] https://www.geeksforgeeks.org/socket-programming-cc/
[4] https://www.scaler.in/socket-programming-in-computer-network/
[5] https://www.spiceworks.com/tech/networking/articles/tcp-vs-udp/
[6] https://realpython.com/python-sockets/
[7] https://www.cs.dartmouth.edu/~campbell/cs60/socketprogramming.html
[8] https://developer.mozilla.org/en-US/docs/Learn_web_development/Howto/Web_mechanics/What_is_a_web_server

6. Teamwork:

We all did the same work at the same time, and we have a discord call and work together:

- Alaa Faraj 1220808:
  Task1: Used Wireshark to find the dns queries and response.
  Task2: Main English Webpage (main_en.html).
  Arabic Version (main_ar.html).
  Task3: TCP connection, UDP connection, Gamesetup
- Adnan Odah 1220175:
  Task1: write commands in the command line such as ipconfig, ping, tracert/traceroute.
  Task2: Local Webpage (mySite_STDID_en.html)
  Server Functionality.
  Task3: TCP connection, UDP connection, Gamesetup
- Abood Faialh 1220216:
  Task1: found the details about the autonomous system (AS) number, number of IP addresses,
  prefixes, peers, and the name of Tier 1 ISP(s) using the telnet command.
  Task 2: Default and Error Responses
  Server Functionality
  Task3: TCP connection, UDP connection, Gamesetup