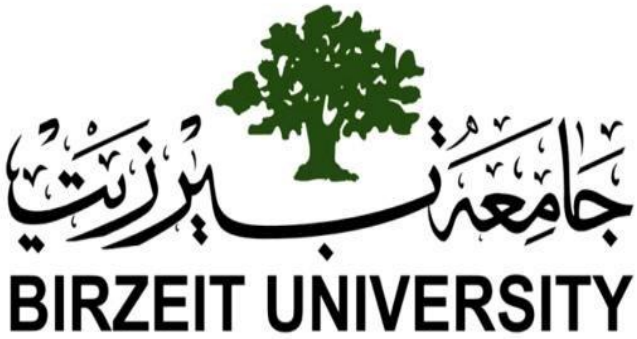

Faculty of Engineering & Technology
Electrical & Computer Engineering Department

COMPUTER NETWORKS– (ENCS3320)

Project Report

Prepared by:

Aws Hammad Id : 1221697

Amir Al-Rashayda Id : 1222596

Ibraheem Sleet Id : 1220200

Instructor: Dr. Ibrahem Nemer & Dr. Alhareth Zyoud

Section: 2 , 4

Date: 10. May. 2025

Abstract

This report details the execution of the Project , which was to implement learning of computer networks via socket programming and network analysis in action. The project involved three students and had three main tasks. Task 1 was to learn some simple network commands (ipconfig, ping, tracert, telnet, nslookup) and analyze network traffic using Wireshark, specifically focusing on DNS queries. Task 2 was to implement a simple web server from scratch using standard socket libraries, handling static HTML/CSS content, image rendering, handling different URL paths like default and error pages, and HTTP redirects for non-existent image/video requests. Finally, Task 3 involved designing a hybrid client-server multiplayer guessing game with TCP for secure connection setup, control messages, and end results announcements, and UDP for high-speed real-time player guesses and feedback. This paper demonstrates the theoretical foundation, implementation approaches, experimental results with screenshots and discussion, challenges encountered, and team work contributions for each task, demonstrating practical application of socket programming principles and network protocols.

Table of contents

Abstract ..... **I**

Table of contents ..... **II**

Table of Figures ..... **IV**

Theory **1**

Procedure & Data Analysis **7**

1. Task 1 (Network Commands and Wireshark): 7

1.1 Task 1 (A): 7

1.2 Task 1 (B): 7

1.2.1 ipconfig /all: 8

1.2.2 Local Network Ping: 9

1.2.3 External Network Ping: 10

1.2.4 Tracing the Route with tracert 11

1.2.5 DNS Lookup with nslookup 12

1.2.6 Testing TCP Connectivity with telnet on Port 80 13

1.2.7 Gathering BGP Details Online 14

1.3 Task 1 (C) - Wireshark Capture of DNS Traffic: 16

2. Task 2 (Web Server): 18

Main English Webpage: 19

Useful Links and Educational Content on Network Security: 20

Supporting Material English Webpage: 21

Main Arabic Webpage: 22

Useful Links and Educational Content on Network Security (Arabic): 23

Supporting Material Arabic Webpage: 24

HTML Code: 25

CSS Code: 28

Web Server Implementation and Code: 31

HTTP Response Builder Function:.....32
Error Handling Functionality: 33
Client Request Handling Function: 34
Requests and Response Testing:.....35
3. Task 3 (Web Server): 46
3.1 Server Implementation and Code: 46
3.2 Client Implementation and Code:.....53
3.3 Results & Testing: 56
Teamwork.....61
Conclusion.....62
References 63

Table of Figures

Figure 1: Network Fundamentals and Command Line Tools, task1-MindMap 1
Figure 2: Web Server Principles and Socket Programming, task2-MindMap 3
Figure 3: Hybrid Protocol Applications and Concurrent Programming, task3-MindMap 5
Figure 4: ipconfig /all command 8
Figure 5: Local Network Ping command 9
Figure 6: External Network Ping command 10
Figure 7: External Network Ping command 11
Figure 8: DNS Lookup with nslookup command 12
Figure 9: TCP Connectivity with telnet on Port 80 command 13
Figure 10: TCP Connectivity with telnet on Port 80 command (successful) 13
Figure 11: BGP lookup gaia.cs.umass.edu (Overview/Upstream) 14
Figure 12: BGP lookup gaia.cs.umass.edu (Prefixes) 14
Figure 13: AS 1968's details 15
Figure 14: starting Wireshark 16
Figure 15: Flushing the local DNS resolver cache 16
Figure 16: Opening gaia.cs.umass.edu in a web browser 16
Figure 17: capturing request and response for gaia.cs.umass.edu 17
Figure 18: Main English Webpage 19
Figure 19: Main English Webpage (Explaining Network security – zoomed out) 20
Figure 20: Supporting material English webpage 21
Figure 21: Main Arabic Webpage 22
Figure 22: Main Arabic Webpage (Explaining Network security – zoomed out) 23
Figure 23: Supporting Material Arabic Webpage 24
Figure 24: HTML Code for Main English Webpage 26
Figure 25: HTML Code for Supporting Material English Webpage 27
Figure 26: CSS Code 30
Figure 27: importing libraries in server code 31
Figure 28: build_response function code 32
Figure 29: Error handling code 33
Figure 30: handle_request function code 34
Figure 31: Browser Request for main English webpage 35
Figure 32: Requests for main English webpage 35
Figure 33: capturing request and response 1 36
Figure 34: Requests for main English webpage 2 36
Figure 35: Requests for main English webpage 3 37
Figure 36: Response for the main English webpage 37
Figure 37: Requests for main Arabic webpage 1 38
Figure 38: Requests for main Arabic webpage 2 38
Figure 39: Response for the main Arabic webpage 39
Figure 40: Request for invalid file 39
Figure 41: Response for invalid request 40
Figure 42: Request for Supporting Material Webpage 40

Figure 43: Search for image in supporting material 41
Figure 44: Request for image from Supporting Material 41
Figure 45: Response for the image 42
Figure 46: Request a video from Supporting Material 42
Figure 47: Request for a video 43
Figure 48: Response for the Video 43
Figure 49: Request for a webpage from another device 44
Figure 50: Server Response to another device request 44
Figure 51: Birzeit University Website 45
Figure 52: Textbook Website 45
Figure 53: initial segment of the server code 47
Figure 54: accept_joins function 48
Figure 55: Game Initialization. 49
Figure 56: Main Gameplay Loop. 50
Figure 57: Handling Player Messages and Exits 51
Figure 58: Processing Guesses. 52
Figure 59: Announcing Results. 52
Figure 60: Prompting for a New Game. 53
Figure 61: Initial segment of the client code. 54
Figure 62: Client functions. 55
Figure 63: Start a game with 2 players 56
Figure 64: Enter a guess and receive feedback 56
Figure 65: Ending a round and start a new one. 57
Figure 66: Starting a new round. 57
Figure 67: Out of the range case. 58
Figure 68: no players guesses the correct number 58
Figure 69: Disconnected player 59
Figure 70: Three players waiting 59
Figure 71: Starting the game for three players. 60
Figure 72: Teamwork Chart 61

Theory

1. Network Fundamentals and Command Line Tools (Relevant to Task1):

Task 1 entails using standard command-line network tools and analyzing network traffic according to basic networking principles.

IP Addressing and Configuration (ipconfig/ifconfig): Understanding how devices are addressed and configured on a network is fundamental. IP addresses (IPv4 or IPv6) provide single logical identification of hosts. Subnet mask defines what bit of an IP address represents the network and what represents the host, splitting one big network into a number of smaller subnets. A default gateway is the IP address of a router to which traffic must be routed when the destination is on another network. DNS server addresses inform the routers where to send traffic to servers that are responsible for translating user-friendly domain names into IP addresses. ipconfig (in Windows) and ifconfig or ip (Unix/Linux) are list and manipulate this configuration information on a local interface commands.

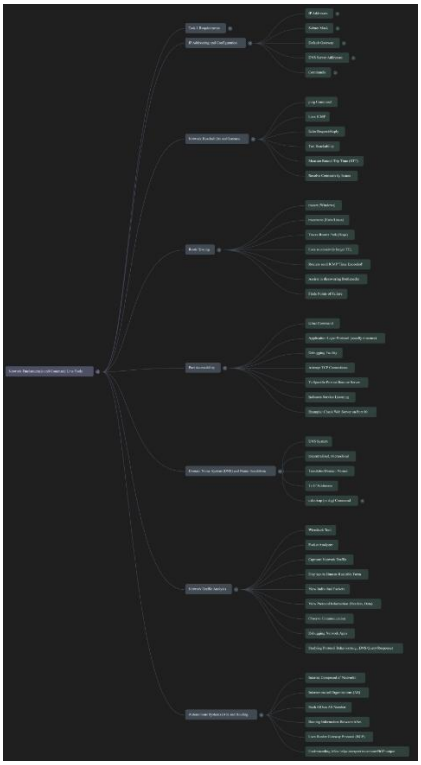

Figure 1: Network Fundamentals and Command Line Tools, task1-MindMap

Network Reachability and Latency (ping): The ping command uses Internet Control Message Protocol (ICMP) Echo Request and Echo Reply packets to test the reachability of a host on an Internet Protocol (IP) network and measure the round-trip time (RTT) for messages from the sending host to a receiving computer and back. It's an easy tool to utilize for resolving network connectivity issues and measuring latency.

Route Tracing (tracert/traceroute): tracert (Win) and traceroute (Unix/Linux) command traces the order of routers (hops) along the path that IP packets take from source to destination. The command sends packets with successively larger Time To Live (TTL) values, which causes the transit routers to respond with ICMP "Time Exceeded" messages. By following the source IPs of these ICMP messages back to the source, the command outlines the order of routers (hops) in the path and assists in the discovery of network bottlenecks or points of failure.

Port accessibility (telnet): While primarily an application-layer protocol for interactive text dialogue (which typically executes insecurely), telnet also acts as a straightforward debugging facility to attempt Transmission Control Protocol (TCP) connections to a specific port on a remote server. If it does connect, this indicates that a service is listening on the port. Attempting telnet to port 80, for example, helps check if a web server is active and accessible.

Domain Name System (DNS) and Name Resolution (nslookup): Domain Name System (DNS) is a decentralized, hierarchical computer, service, or any other Internet- or private network-addressable resource name system. DNS translates human-readable domain names (like gaia.cs.umass.edu) to machine-readable IP addresses. nslookup (or dig) is a command-line program to be used for DNS query of domain name or IP address mapping, and other DNS records.

Network Traffic Analysis (Wireshark): Wireshark is a widely used packet analyzer. It captures network traffic in real time and displays it in human-readable form so that users can view individual packets, view protocol information (headers, data), and observe how communication occurs between devices. It's invaluable when debugging network apps and studying protocol behavior, e.g., the typical DNS query and response process.

Autonomous Systems (AS) and Routing: The Internet is composed of networks interconnected by different organizations, referred to as Autonomous Systems (AS). Each AS has an AS number. Routing information between ASes is conveyed primarily by using the Border Gateway Protocol. Understanding ASes puts traceroute output and output obtained using BGP tools into perspective, showing which major networks are utilized to route traffic to a destination like gaia.cs.umass.edu.

2. Web Server Principles and Socket Programming (Relevant to Task2):

Task 2 is to implement a simple web server, client-server communication, and the Hypertext Transfer Protocol using TCP sockets.

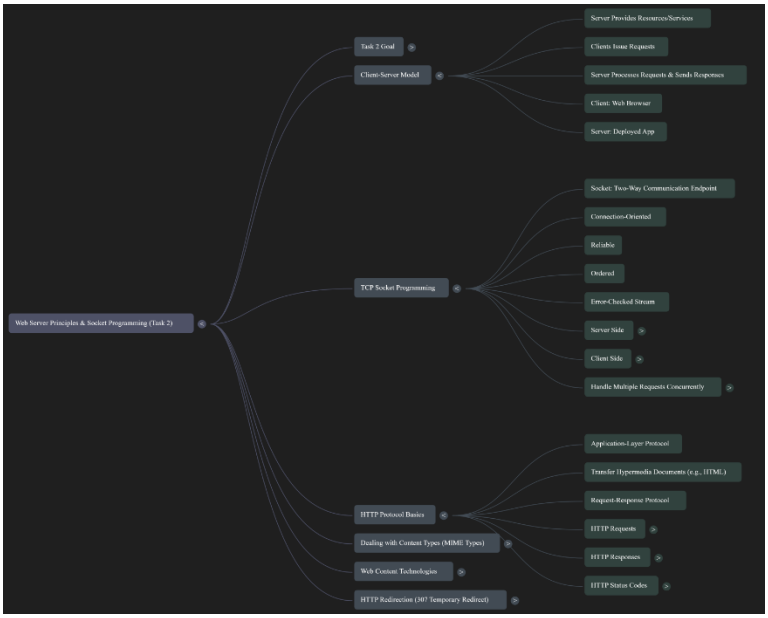

Figure 2: Web Server Principles and Socket Programming, task2-MindMap

- **Client-Server Model**: This is a simple network architecture where a server provides resources or services to one or more clients. Clients establish contact by issuing requests to the server, and the server processes the requests and sends back responses. In this task, a web browser acts as the client, and our deployed app is the server.
- **TCP Socket Programming**: A socket is one end-point of a two-way communication between programs running on the same network. TCP sockets provide a connection-oriented, reliable, ordered, and error-checked stream of bytes.
- **Server Side**: A TCP server typically creates a socket (socket()), binds it to a specific IP address and port (bind()), listens for incoming connections (listen()), and accepts incoming client connections (accept()). Each accept() call returns a new socket for communicating with that client. The server then accepts requests (recv()) and sends responses (sendall() or send()) on this client socket.
- **Client Side**: A TCP client creates a socket (socket()) and connects to the IP address and port of the server (connect()). Once connected, it sends requests (sendall() or send()) and receives responses (recv()).
- For this task, the server must handle multiple incoming HTTP requests potentially from different clients at the same time. This normally demands techniques like multithreading or

multiprocessing to handle each client connection independently and in parallel, without the server blocking while waiting for data from a specific client. (Even though threading is explicitly allowed for Task 3, concurrent handling concepts are inferred for building a responsive web server).

- **HTTP Protocol Basics**: Hypertext Transfer Protocol (HTTP) is the application-layer protocol for the transfer of hypermedia documents, such as HTML. It is a request-response protocol.
    - **HTTP Requests**: Clients send requests to the server (e.g., GET /path/to/resource HTTP/1.1\rHost:.\r). It consists of an HTTP method (e.g., GET), the requested resource's path (URL), and HTTP headers that define additional information (e.g., Host, User-Agent).
    - **HTTP Responses**: Servers respond with a status line (HTTP/1.1 200 OK\r), HTTP headers (Content-Type, Content-Length, etc.), and an optional message body (the data requested, e.g., HTML, CSS, or an image).
    - **HTTP Status Codes**: Three-digit numbers indicating the result of the request (e.g., 200 OK for success, 404 Not Found if there is no such resource, 307 Temporary Redirect to indicate the resource is temporarily somewhere else).
- **Dealing with Different Types of Content (MIME Types)**: Servers must inform the client (browser) about the kind of content that is being sent in the response body. This is done via the Content-Type HTTP header, using MIME types (e.g., text/html for HTML pages, text/css for CSS files, image/png for PNG images, video/mp4 for MP4 videos). The browser then uses this to correctly interpret and render the content.
- **Web Content Technologies (HTML, CSS):** HTML (HyperText Markup Language) is utilized to define the structure and content of web pages using elements and tags. CSS (Cascading Style Sheets) describes how the HTML elements should be displayed on the screen or other media, such as layout, colors, fonts, etc. These static files need to be read and served by the server.
- **HTTP Redirection (307 Temporary Redirect):** The HTTP 307 status code indicates that the target resource has been temporarily moved to a different URL, as indicated in the Location response header. The client (browser) should automatically redirect to the new URL without altering the initial request method. This is used in Task 2 for redirecting clients to search engine results when a requested local file does not exist.

3. Hybrid Protocol Applications and Concurrent Programming (Relevant to Task3):

Task 3 involves designing a real-time interactive game that strategically uses both TCP and UDP protocols and requires handling multiple players simultaneously.

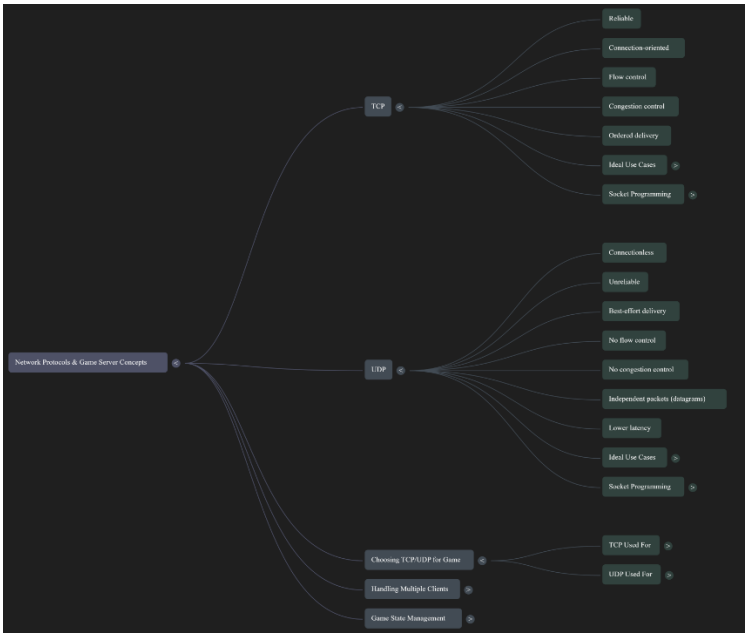

Figure 3: Hybrid Protocol Applications and Concurrent Programming, task3-MindMap

- **TCP vs. UDP**: Features and Applications:
  - **TCP (Transmission Control Protocol):** Ordered, connection-oriented, flow control, congestion control. Appropriate for applications where data order and consistency are critical, such as the initial handshakes, game rules transfers, registration commands, and final game outcomes where delivery reliability is a priority.
  - **UDP (User Datagram Protocol):** No connection, unreliable, best effort delivery, no flow control, no congestion control. Data is sent in independent packets (datagrams). Lower latency since there is less overhead. Real-time applications like games where speed is more important than guaranteed delivery of each and every packet, and lost packets are often acceptable or can be suppressed (e.g., one guess update may get lost, but the next one will arrive).

- **TCP and UDP chosen for the Game:** TCP is used in Task 3 for initial configuration (JOIN, rules), state changes (start of the game, declaring a winner), and assured commands. The rapid, round-to-round guessing time is chosen with UDP since speedy feedback ("Higher", "Lower", "Correct") is needed by players without the overhead and attendant delay of TCP retransmission on packet loss. A.rst slight delay or lost feedback packet during the guessing phase is less detrimental to the game.ow than a lost registration command or game result.
- **Socket Programming for TCP and UDP:**
  - **send()/sendall() is employed by TCP** sockets to write a stream of data and recv() to read a stream. The connection is established and maintained.
  - **UDP sockets use sendto(data, address)** to send a datagram to a destination address (IP, Port) and recvfrom(buffer_size) to receive a datagram, which also provides the sender's address. UDP is not connection-based; the server receives packets from numerous clients on its single listening UDP socket, and responses must be sent with the received address using recvfrom.
- **Handling Multiple Clients and Concurrency:** A game server has to handle lots of players concurrently. One of the common ways of doing that is by multithreading. Each thread may perform a separate task (for example, waiting for incoming TCP connections, waiting for UDP packets, or dealing with the state and communications of one player). This allows the server to remain responsive to all the players and external events without being bogged down by, for example, waiting for a single client. Python's threading module makes it simple to make and handle these concurrent threads.
- **Game State Management:** The server must track the game state (e.g., active players, their TCP connections and UDP addresses, secret number, timer, round status). The state must be readily accessible and atomically updatable by multiple threads handling client input.

Procedure & Data Analysis

1. Task 1 (Network Commands and Wireshark):

This task is about familiarizing us with the fundamental network diagnostic and information-gathering commands that are typically employed to debug connectivity, identify network settings, and trace routes. It also employs a packet sniffing utility, Wireshark, to observe network traffic at a lower level (i.e., the Domain Name System (DNS) protocol). Familiarity with these tools and concepts is necessary to diagnose network issues and comprehend how applications communicate over the network.

1.1 Task 1 (A):

We researched and defined the following essential network commands based on their primary functions:

I) ***ipconfig***: This command provides you with information about your network adapters and how they are set up. It tells you about your IP addresses, subnet masks, default gateways, and DNS servers.

II) ***Ping***: We use the ping command to check if we can reach another device on the network or the internet. It sends test messages and tells us if the device responded and how quickly.

III) ***tracert (Windows) / traceroute (macOS/Linux)***: The tracert or traceroute command shows us the path the data takes to get to a specific server or website. It tells us every router our data passes through and how long it took to reach each of them.

IV) ***Telnet***: We can use the telnet command to verify if a specific service is operating on a specific server's port. It tries to open a direct connection and shows if it was successful or rejected.

V) ***Nslookup:*** The nslookup command enables us to find out the IP address that corresponds to the name of a website by asking our DNS server. It is used for looking up domain name information.

1.2 Task 1 (B):

We performed various basic network commands on a Windows system to gather information about the local network setup and connectivity to external hosts. Below are the results and our discussion for each step.

1.2.1 ipconfig /all:

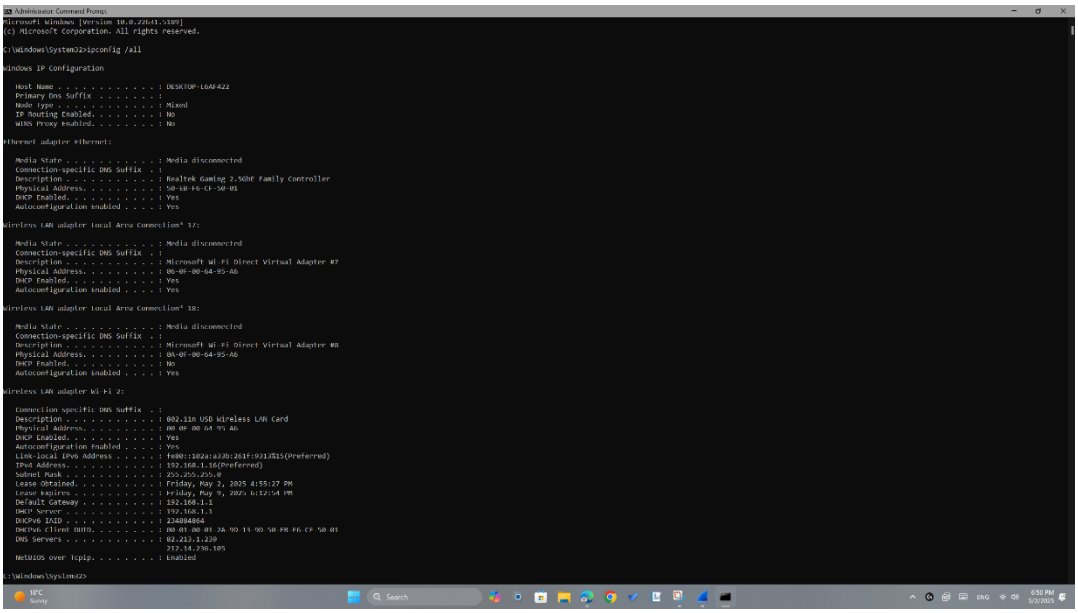

Figure 4: ipconfig /all command

We employed the ipconfig /all command in order to find out the configuration of the network interfaces on our system. The output provides certain information relevant for understanding how our system is plugged into the network.

Figure 4 shows the output of the ipconfig /all command. From the output of this Wireless LAN adapter Wi-Fi 2 (which was identified as the active network interface), we extracted the following configuration details:

- **IPv4 Address**: 192.168.1.16 (This is the unique IP address of my computer on the local network).
- **Subnet Mask**: 255.255.255.0 (This is the portion of the IP address that specifies the network our computer is on. In this case, it is a standard /24 network block.
- **Default Gateway**: 192.168.1.1 (This is the router's IP address, which serves as the gateway from our local network to other networks, including the internet).
- **DNS Servers**: 192.168.1.1, 82.213.1.239, 212.14.236.105 (These are the IP addresses of the servers that our computer uses in order to translate domain names into IP addresses. The first one is likely the router, which is possibly forwarding requests to external DNS servers, which are the second and third IPs).

This output confirms our computer's assigned identity on the local network (IP address) and its configuration for communicating both locally (Subnet Mask, potentially Gateway for local routing) and externally (Default Gateway, DNS Servers).

1.2.2 Local Network Ping:

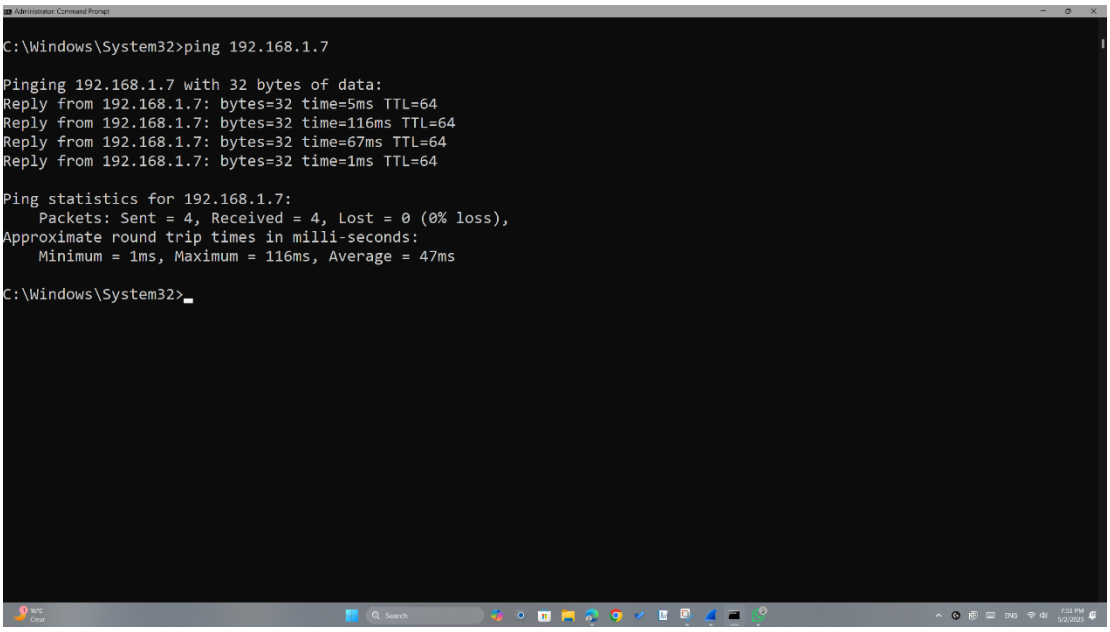

Figure 5: Local Network Ping command

Following our identification of our local network configuration, we then sent a ping to one device on the same local network segment (which is my smartphone). The aim here was to establish simple connectivity and note the round-trip time (latency) from our computer to the smartphone on the local network (in this case, a smartphone with an IP address of 192.168.1.7).

We executed the command ping 192.168.1.7 in the command prompt. Figure 2 displays the output obtained. The results showed that 4 packets were sent and 4 packets were received, resulting in 0% packet loss. This confirms successful network reachability between our PC and the smartphone on the local network. The round-trip times (RTT) were recorded as follows:

- Minimum = 1ms
- Maximum = 116ms
- Average = 47ms

The 0% packet loss does indicate good connectivity on the local network during the test; the measured RTT values are expected for communication between devices that typically are on the same Wi-Fi or wired network segment. Local pings should usually be in the 1-10 millisecond range; the main objective of verifying successful local connectivity via ping was achieved.

1.2.3 External Network Ping:

To test connectivity to a server outside our local network and measure typical internet latency, we used the ping command targeting the hostname gaia.cs.umass.edu. We executed the command ping gaia.cs.umass.edu. Figure 3 shows the output from this execution.

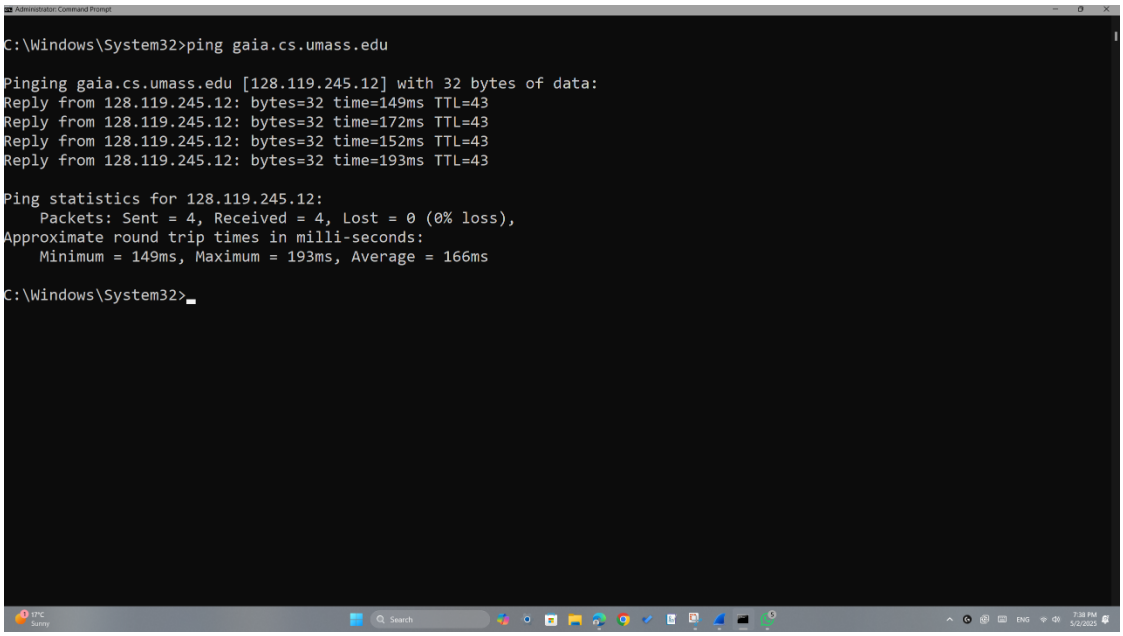

Figure 6: External Network Ping command

The ping command resolved gaia.cs.umass.edu to the IPv4 address 128.119.245.12. It then sent four ICMP Echo Request packets to this IP address. The results show 4 packets sent, 4 received, and 0% packet loss, confirming successful reachability over the internet. The round-trip times (RTT) were recorded as follows:

- Minimum = 149ms
- Maximum = 193ms
- Average = 166ms

In comparison to the local network ping at task B.2, the RTT mean (166ms) is significantly higher than the local ping time (47ms). This increased latency is normal for internet communication, as the packets must travel through numerous routers and segments of the network to arrive at the far-off server at the University of Massachusetts. The 0% packet loss shows an uninterrupted connection path throughout the test.

1.2.4 Tracing the Route with tracert

To observe the route packets, take from our computer to gaia.cs.umass.edu, we employed the tracert command. It helps in identifying the list of routers (hops) taken and the time at each hop in the route over the internet. We executed the command tracert gaia.cs.umass.edu in the command prompt.

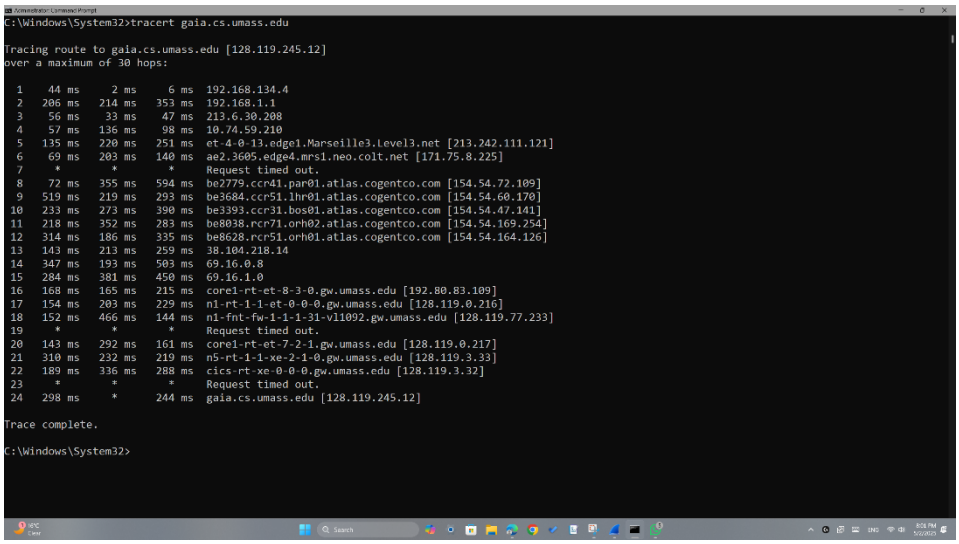

Figure 7: External Network Ping command

To observe the route packets, take from our computer to gaia.cs.umass.edu, we employed the tracert command. It helps in identifying the list of routers (hops) taken and the time at each hop in the route over the internet. We executed the command tracert gaia.cs.umass.edu in the command prompt. Figure 4 shows the complete trace output. Key observations from the route trace include:

- The trace started from our local network gateway and moved quickly onto the Internet Service Provider's network.
- Packets passed through several hops, such as infrastructure belonging to providers like Cogent (in line with our BGP lookup results).
- Round-trip times mostly went up from single-digit milliseconds at first to greater values (more than 200ms) on more remote hops.
- A few hops exhibited timeouts (*), which typically happens when routers fail to transmit ICMP replies and does not imply the path is down or relates to security matters.
- The final hops obviously show routers within the umass.edu network domain, affirming successful delivery to the destination network infrastructure.

1.2.5 DNS Lookup with nslookup

We used the nslookup command to search the Domain Name System (DNS) for the hostname gaia.cs.umass.edu and determine its IP address. This confirms how the hostname resolves using our DNS servers. We entered the command nslookup gaia.cs.umass.edu on the command prompt. Figure 5 illustrates the result.

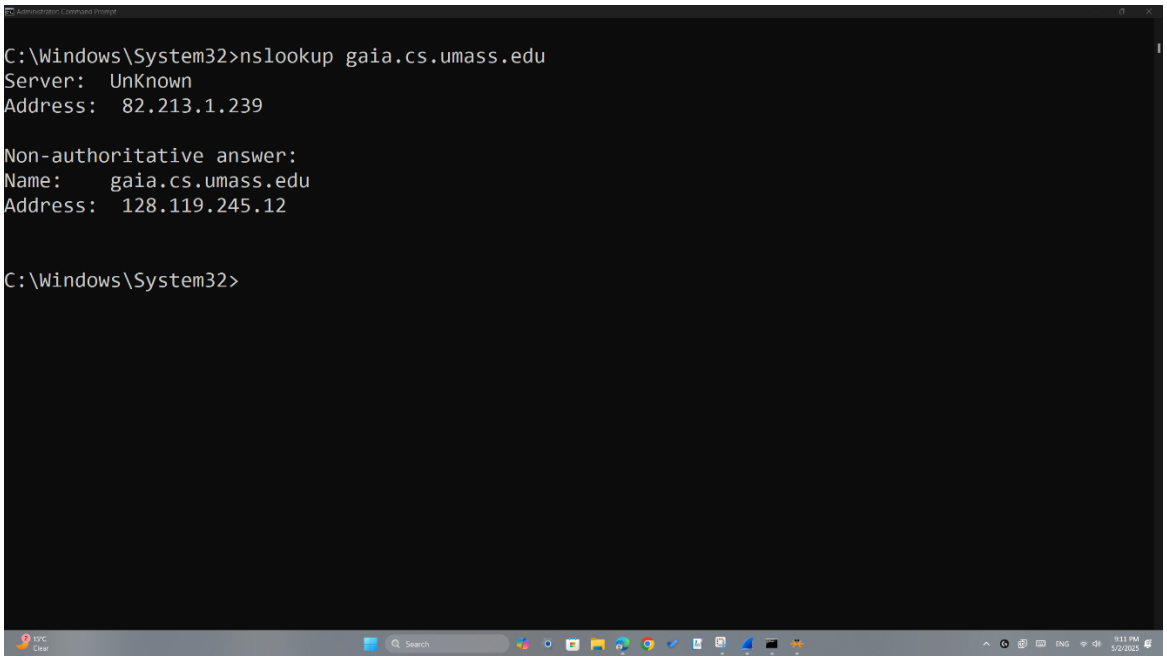

Figure 8: DNS Lookup with nslookup command

The output indicates that the DNS query was processed by our default DNS server(s) (82.213.1.239). The key result from the non-authoritative answer section is:

- Name: gaia.cs.umass.edu
- Address: 128.119.245.12

This command confirms the name resolution of the domain name gaia.cs.umass.edu to its assigned IPv4 address 128.119.245.12. It is the same IP address that is also found with the ping and tracert commands, and it continues to demonstrate consistency with which the hostname resolves to an IP by employing different tools. nslookup is a valuable utility for monitoring DNS records as well as name resolution issues.

1.2.6 Testing TCP Connectivity with telnet on Port 80

This step was designed to check if a service was actively listening for TCP connections on the default HTTP port (port 80) on the host gaia.cs.umass.edu. The telnet command is useful for basic port connectivity testing. We issued the command telnet gaia.cs.umass.edu 80 (Figure 6). Now Figure 7 shows the outcome of this connection request (connection mentioned in the header "telnet gaia.cs.umass.edu").

Figure 9: TCP Connectivity with telnet on Port 80 command

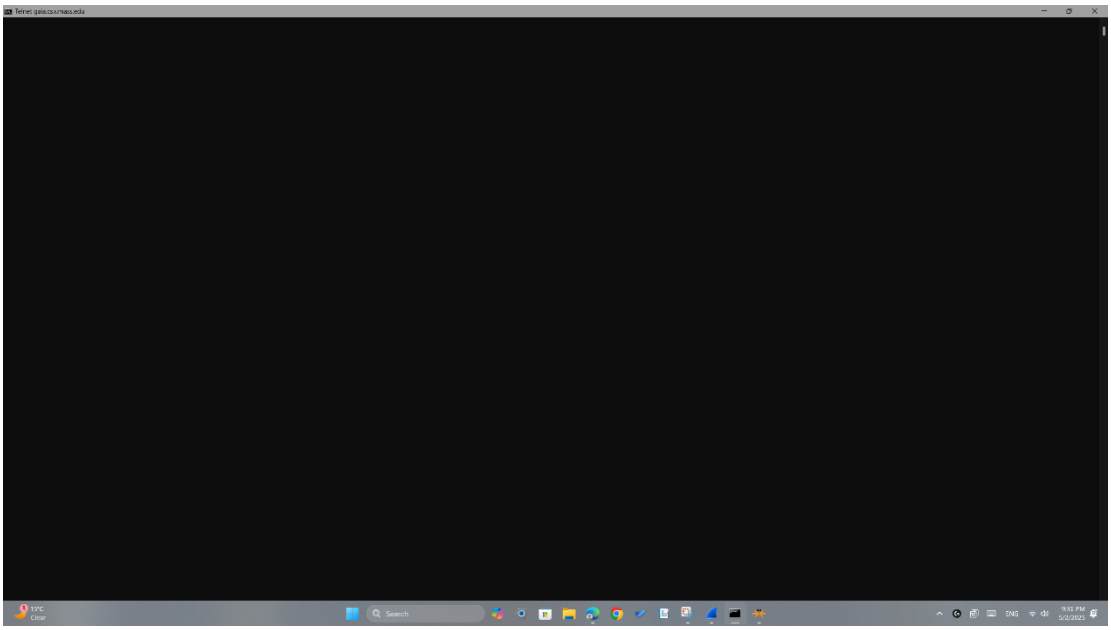

Figure 10: TCP Connectivity with telnet on Port 80 command (successful)

Upon executing the command, the telnet client successfully established a connection to gaia.cs.umass.edu on port 80. This was indicated by the screen clearing and presenting a blank interface with a blinking cursor (as seen in the screenshot title bar confirming the connection target).

Successful establishment of the connection to port 80 confirms the existence of a service, i.e., typically a web server responding to typical HTTP requests, operational and ready to accept the connection on said port on destination IP address (128.119.245.12). Server software (e.g., Apache, Nginx, etc.) is alive and network reachable on a standard web surfing port. While telnet itself does not display the web page without manually sending further HTTP commands, successfully connecting to the port confirms the base layer of service availability.

1.2.7 Gathering BGP Details Online

To accomplish this process, we utilized an online BGP lookup utility (BGP.Tools) to investigate the Autonomous System (AS) of gaia.cs.umass.edu and its connection to other networks, such as identifying Tier 1 ISPs. Having these BGP details provides information on the network topology and peering relationships for the target host's network.

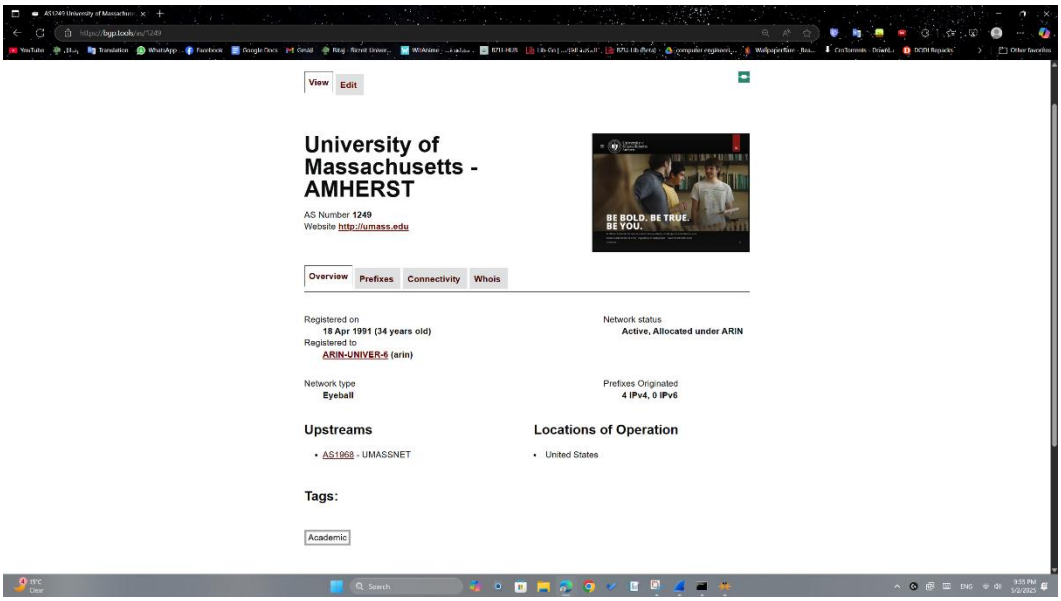

Figure 11: BGP lookup gaia.cs.umass.edu (Overview/Upstream)

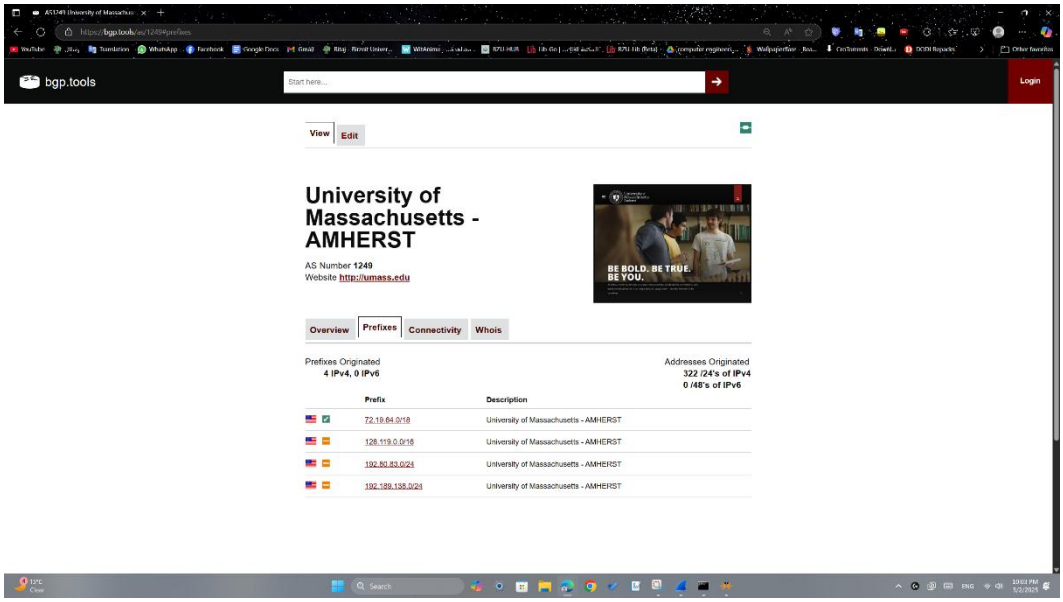

Figure 12: BGP lookup gaia.cs.umass.edu (Prefixes)

We have searched for gaia.cs.umass.edu or its IP address (128.119.245.12). Figure 8 gives the information of its overview and upstream network. Figure 9 gives the summary and

prefix information of the pertinent AS, and from the lookup results (shown in Figures 8 and 9), we gathered the following BGP details:

- Autonomous System (AS) number: Network block gaia.cs.umass.edu originates in AS 1249, that is, University of Massachusetts - AMHERST.
- Number of IP addresses: AS 1249 originates in network prefixes to correspond to 322 /24's of IPv4 addresses.
- Prefixes: The main IPv4 network prefixes originated by AS 1249 are:
  1) 72.19.64.0/18
  2) 128.119.0.0/16
  3) 192.80.83.0/24
  4) 192.189.138.0/24
- Peers: AS 1249's peer/upstream provider to access the internet in general is AS 1968 (UMASSNET). AS 1968's details (Figure 10) indicate that it peered directly to AS 174 (Cogent Communications) and AS 3257 (GTT Communications Inc.).

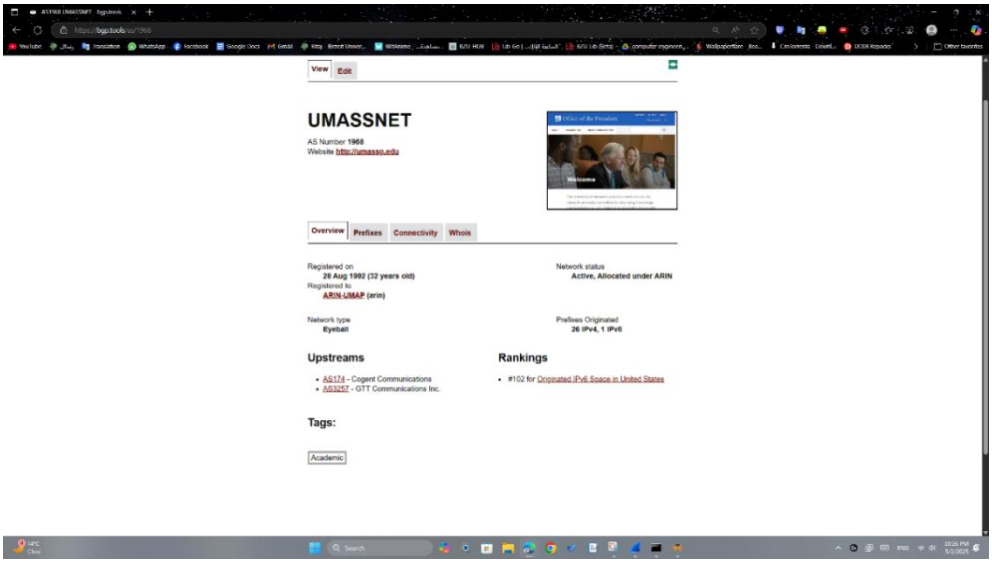

Figure 13: AS 1968's details

Tier 1 ISPs names: Based on AS 1968's immediate upstream connections, the respective Tier 1 ISPs to which UMASS's network is directly connected are Cogent Communications (AS 174) and GTT Communications Inc. (AS 3257), both of which are world-famous global Tier 1 network providers.

The BGP information shows that the University of Massachusetts network (AS 1249) isn't a direct member of the top-level internet backbone but peers with UMASSNET (AS 1968), which further connects to Tier 1 global providers like Cogent and GTT. This is a standard hierarchical structure for academic and regional networks to gain internet connectivity by purchasing transit from more aggregated backbone providers or through peering agreements. The collected prefixes decide the specific IP address ranges that belong to AS 1249.

1.3 Task 1 (C) - Wireshark Capture of DNS Traffic:

We used the multi-purpose network protocol analyzer known as Wireshark to capture the packets sent and received upon resolution by our computer, with assistance from the Domain Name System (DNS), of a text-readable domain name like gaia.cs.umass.edu to its corresponding machine-readable IP address of 128.119.245.12, Our procedure was:

1) Starting a capture on the network interface that is connected to the internet (Wi-Fi for my device.

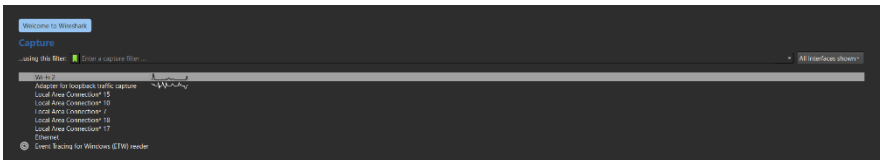

Figure 14: starting Wireshark

2) Flushing the local DNS resolver cache (ipconfig /flushdns) so that a new DNS lookup would occur.

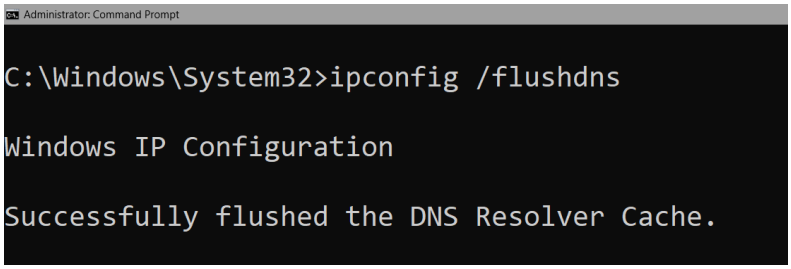

Figure 15: Flushing the local DNS resolver cache

3) Opening gaia.cs.umass.edu in a web browser triggers our system to send a DNS query.

Figure 16: Opening gaia.cs.umass.edu in a web browser

4) Halting the Wireshark capture shortly after the request. Using a display filter (dns) in Wireshark to show only DNS protocol packets.

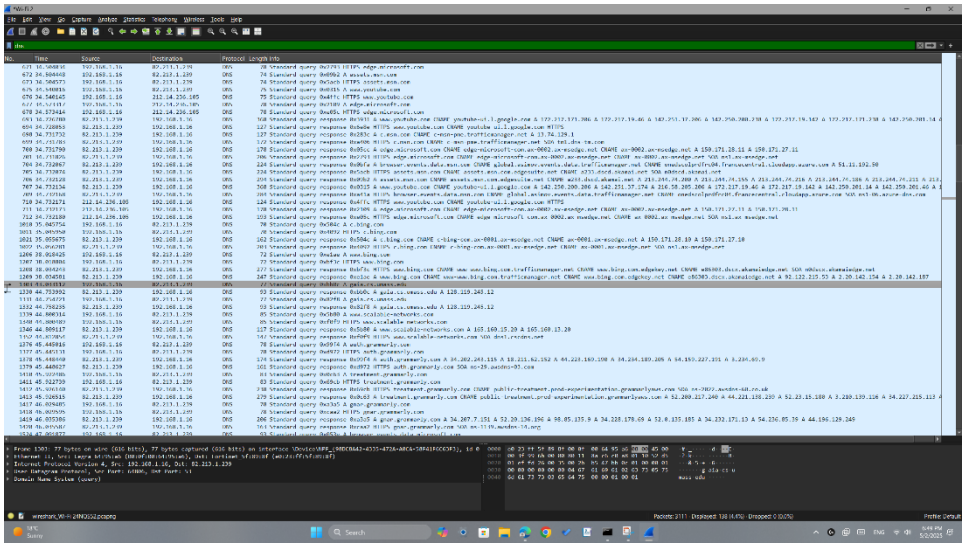

Figure 17: capturing request and response for gaia.cs.umass.edu

Figure 17 illustrates the output of Wireshark restricted to just the DNS traffic, the query and answer for gaia.cs.umass.edu in this instance.

- Packet No. 1303: DNS Query
- Source IP: 192.168.1.16 (Our computer)
- Destination IP: 82.213.1.239 (Our configured DNS server)
- Protocol: DNS
- Info: Standard query 0xbb0c A gaia.cs.umass.edu
- This packet shows our computer sending a standard query to the DNS server, asking for the 'A' record (IPv4 address) for the name gaia.cs.umass.edu. The transaction ID is 0xbb0c.

- Packet No. 1304: DNS Response
- Source IP: 82.213.1.239 (The DNS server)
- Destination IP: 192.168.1.16 (Our computer)
- Protocol: DNS
- Info: Standard query response 0xbb0c A gaia.cs.umass.edu 128.119.245.12
- This packet is the DNS server's response to our query. It shares the same transaction ID (0xbb0c), which confirms that it is the response to Packet 1303. Most importantly, the "Answers" section (shortened in the Info column) has the resolved IP address 128.119.245.12.

This Wireshark capture presents a real-life image of the DNS lookup process at the packet level. By clearing the cache, we compelled the operating system to initiate a live query. The query packet demonstrates the structure of the request made by the client (requesting type 'A' record for the specified hostname), and the response packet demonstrates the server granting the request by sending back the corresponding IP address. This resolved IP address (128.119.245.12) is then used by the operating system and browser to establish direct connections (like the TCP connection that was tested using telnet and the ICMP exchange that was measured using ping). Wireshark confirms that DNS queries typically use UDP on port 53, which can be seen in the packet details (seen by unrolling the layers, though not completely in the figures). This phase stresses the core purpose of DNS to translate hostnames into IPs before higher-level communications.

2. Task 2 (Web Server):

In this task, we were required to implement a custom web server using Python socket programming. The server's purpose is to handle HTTP requests from web clients, such as browsers, and respond based on the requested content. Our server listens on port 9956, which is based on one of our team member's student ID (1222596 → 9956).

To demonstrate the server's functionality, we designed several web pages using HTML and CSS, including a main English page, a main Arabic page, and supporting material pages in both languages. These pages include styled content such as team member profiles, educational material about network security, and an interactive form that allows users to request media files like images or videos related to networking topics.

The server is implemented in Python using the standard socket library and operates over TCP. It processes HTTP GET requests and responds with different status codes based on the request. If the requested file exists, the server returns a 200 OK response with the appropriate content. If the file does not exist but matches an image or video extension, the server issues a 307 Temporary Redirect to a Google search for the requested item. If the requested file does not exist and does not qualify for redirection, the server responds with a custom 404 Not Found page that also displays the client's IP address and port. This implementation demonstrates a complete HTTP request-handling cycle, including listening for connections, processing requests, serving content, managing redirects, and handling error responses. More technical details, code explanations, and testing results are provided in the following sections of this report.

Main English Webpage:

This webpage represents the main page in the English version of our web server. It includes the project title, our team members' details, and links to external resources and the local supporting material webpage.

As shown in Figure 18, this is the Main English Webpage, and the web browser tab correctly displays the title "ENCS3320-Webserver". The page title appears as "Welcome to ENCS3320 - Computer Networks Webserver" with a clean, structured layout and styled background.

Additionally, the webpage presents detailed information for each team member, including their name, student ID, profile picture, listed skills, completed projects, and personal hobbies, all organized in separate profile cards. This helps introduce the team behind the project in a professional and engaging manner.

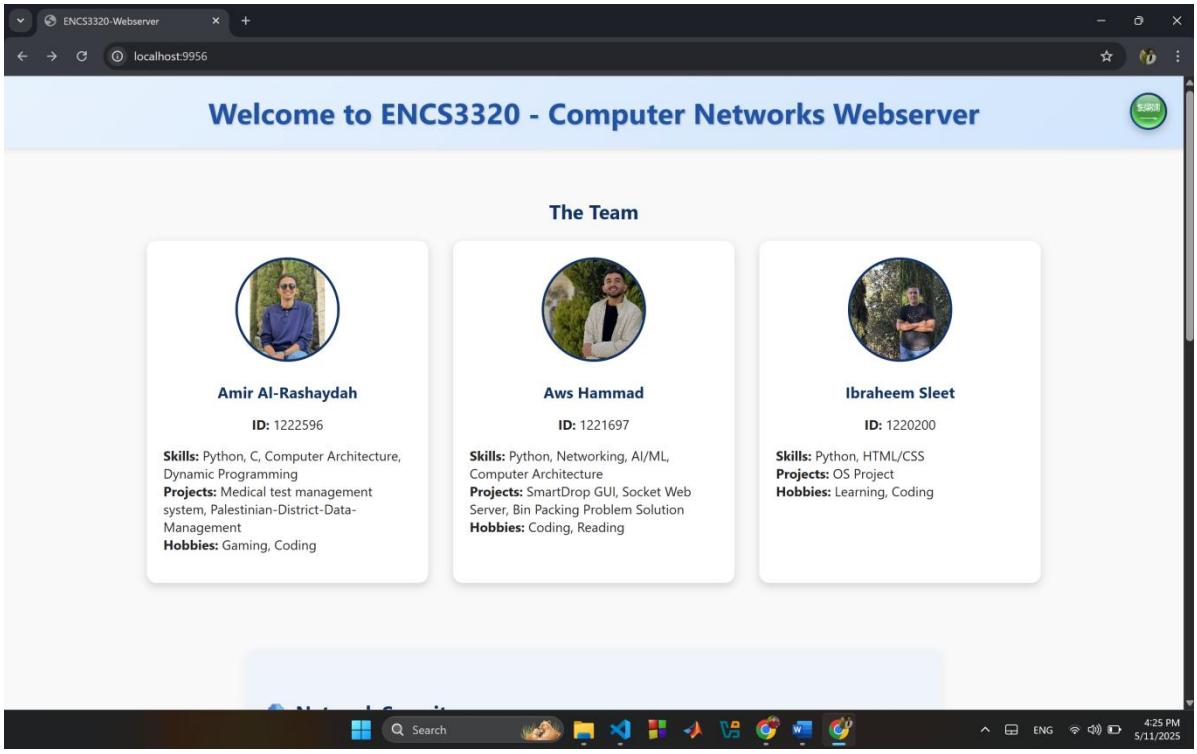

Figure 18: Main English Webpage

Useful Links and Educational Content on Network Security:

As shown in Figure 19, the footer section of the Main English Webpage includes three important links. The first link redirects the user to the Textbook Website for the "Computer Networking: A Top-Down Approach" course material. The second link takes the user to the Birzeit University official website, providing direct access to the university's resources. Finally, there is a link to the Local Site Page, which allows the user to navigate to the supporting material page where they can submit media requests. Additionally, the page includes a well-structured educational section explaining Network Security concepts, which are summarized from the course textbook. This section highlights topics such as common network threats, security design goals, and the importance of including security in all network layers.

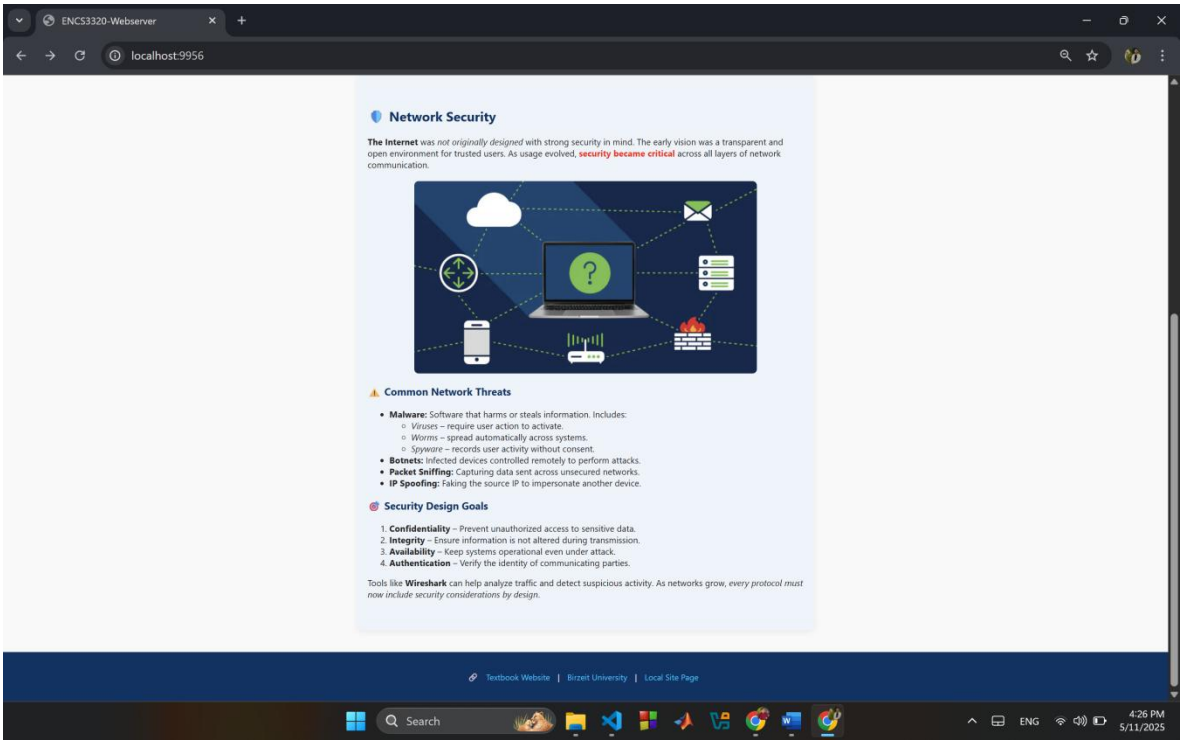

Figure 19: Main English Webpage (Explaining Network security – zoomed out)

Supporting Material English Webpage:

The Supporting Material English Webpage is designed to allow users to request specific images or videos related to networking topics. On this page, users are provided with a form interface where they can enter the file name (including the file extension) in the input field. After submitting the request by clicking on the "Submit Request" button, the server processes the input.

If the requested file exists on the server, the server responds with the file content, such as displaying the image or providing the video. However, if the file is not available, the server issues a 307 Temporary Redirect response. This redirect takes the user to Google Search with results filtered to either images or videos, depending on the type of file requested.

As shown in Figure 20, the form clearly instructs the user to enter the file name, for example, "router.png" or "video.mp4", making the process intuitive and user-friendly. Additional screenshots of the server's behavior when responding to these requests are provided later in the Server Implementation and Testing section.

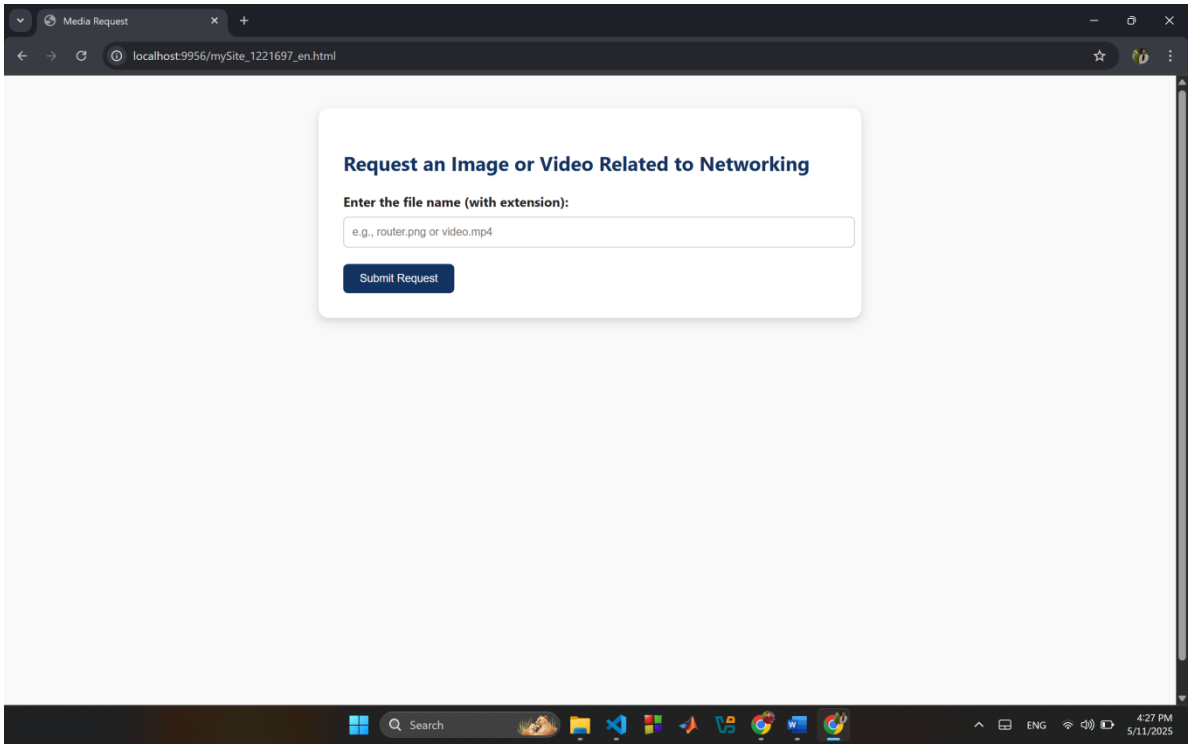

Figure 20: Supporting material English webpage

Main Arabic Webpage:

This webpage represents the main page in the Arabic version of our web server. It includes the project title, team members' details, and links to external resources as well as the local supporting material webpage in Arabic.

A language switch button located at the top-right corner, represented by a Saudi Arabia flag. When the user clicks on the Saudi flag, the Arabic page is already displayed and the button turns into the British flag. If the user clicks on the British flag, the page switches to the English version.

As shown in Figure 21, this is the Main Arabic Webpage, and the web browser tab correctly displays the title "خادم ENCS3320". The page title appears as "مرحبًا بكم في ENCS3320 - شبكات الحاسوب", with a clean, structured layout that follows right-to-left (RTL) formatting to suit Arabic readers.

Additionally, the webpage presents detailed information for each team member, including their name, student ID, profile picture, listed skills, completed projects, and personal hobbies, all organized in separate profile cards. This provides a professional and accessible introduction to the team, fully translated for Arabic-speaking users.

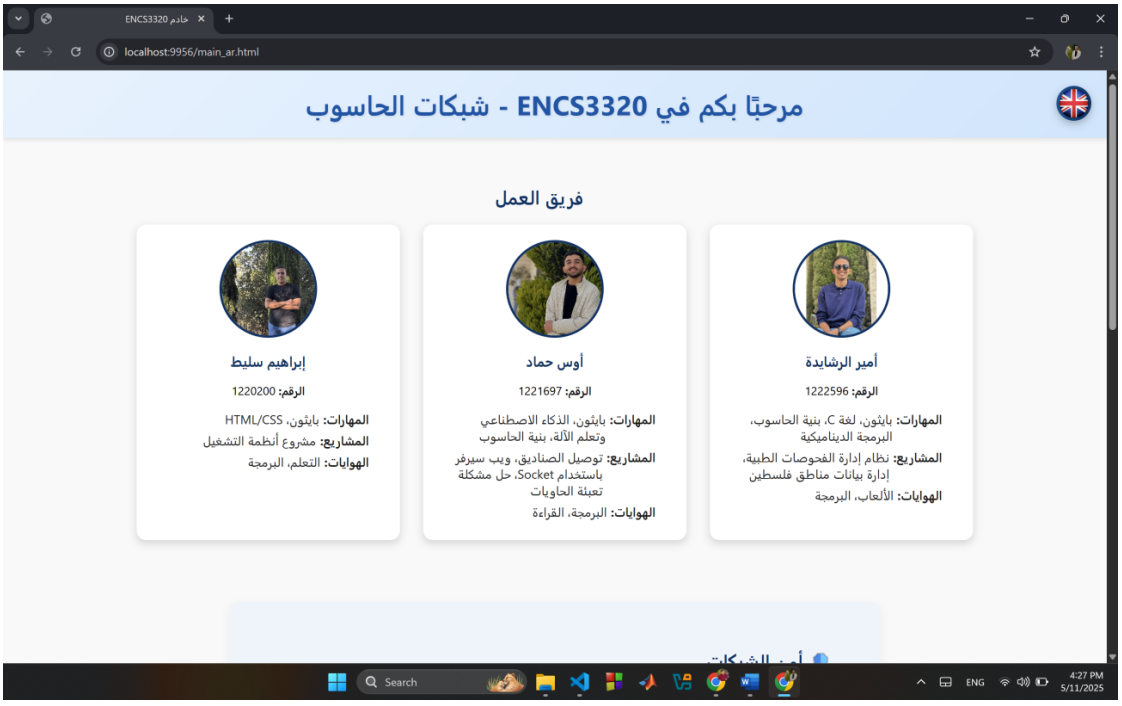

Figure 21: Main Arabic Webpage

Useful Links and Educational Content on Network Security (Arabic):

As shown in Figure 22, the footer section of the Main Arabic Webpage includes three important links. The first link redirects the user to the Textbook Website for the "Computer Networking: A Top-Down Approach" course material. The second link takes the user to the official Birzeit University website, providing direct access to the university's resources. Finally, there is a link to the Local Site Page in Arabic, which allows the user to navigate to the Arabic version of the supporting material page, where they can submit media requests for images or videos related to networking topics.

Additionally, the page contains a well-structured educational section that explains Network Security concepts, presented fully in Arabic and summarized from the course textbook. This section covers common network threats such as malware, botnets, packet sniffing, and IP spoofing. It also highlights security design goals, including confidentiality, integrity, availability, and authentication, emphasizing the importance of integrating security considerations at all layers of the network.

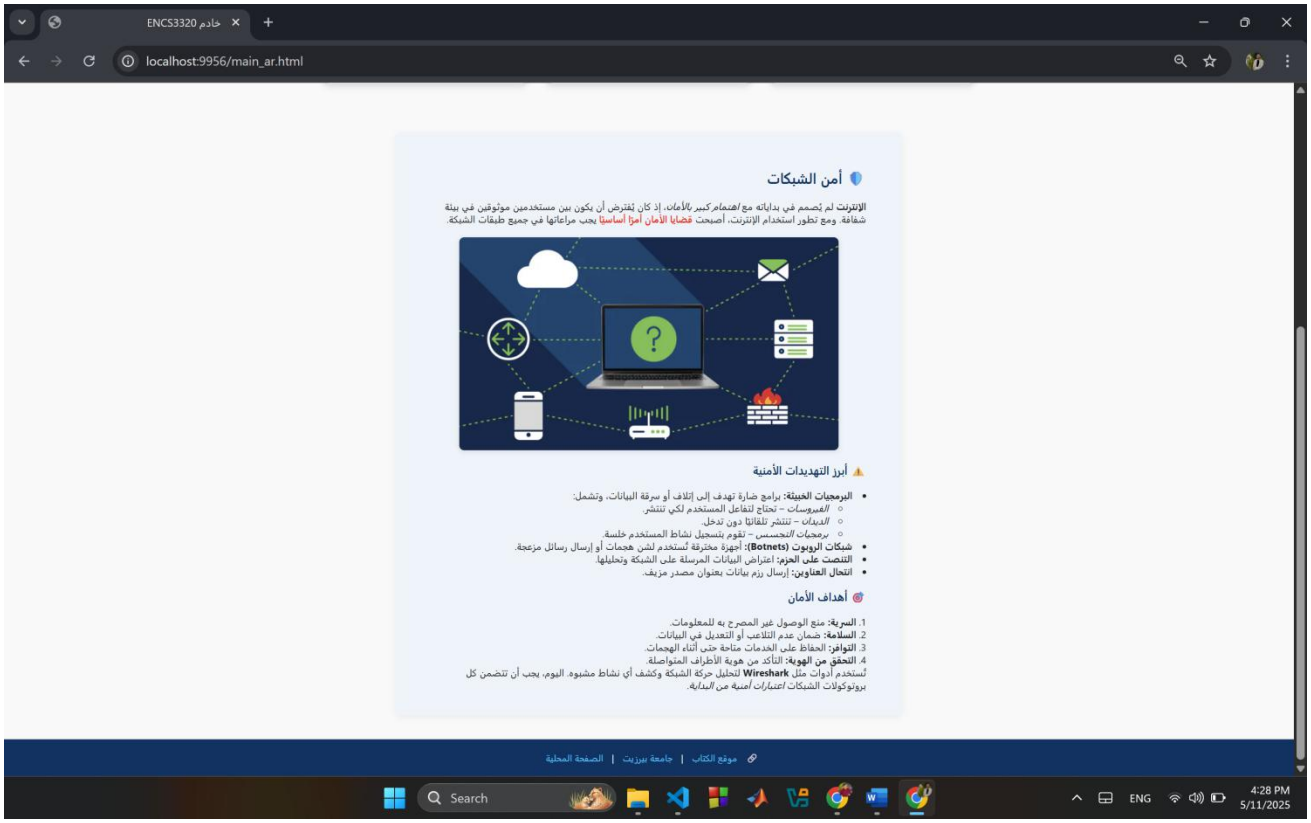

Figure 22: Main Arabic Webpage (Explaining Network security – zoomed out)

Supporting Material Arabic Webpage:

The Supporting Material Arabic Webpage provides users with an interface to request specific images or videos related to networking topics, fully presented in Arabic and designed with right-to-left (RTL) formatting. On this page, users are given a form interface where they can enter the file name (including the file extension) in Arabic or English, such as "router.png" or "video.mp4". After clicking the "Submit Request" button, the server processes the request. If the requested file exists on the server, the server responds by displaying the image or providing the video content. However, if the file is not available, the server responds with a 307 Temporary Redirect. This redirect takes the user to Google Search, showing results filtered to either images or videos, depending on the file type entered.

As shown in Figure 23, the form is simple and user-friendly, clearly instructing the user on how to enter the file name and submit their request. Additional screenshots demonstrating the server's behavior for these cases are included later in the Server Implementation and Testing section.

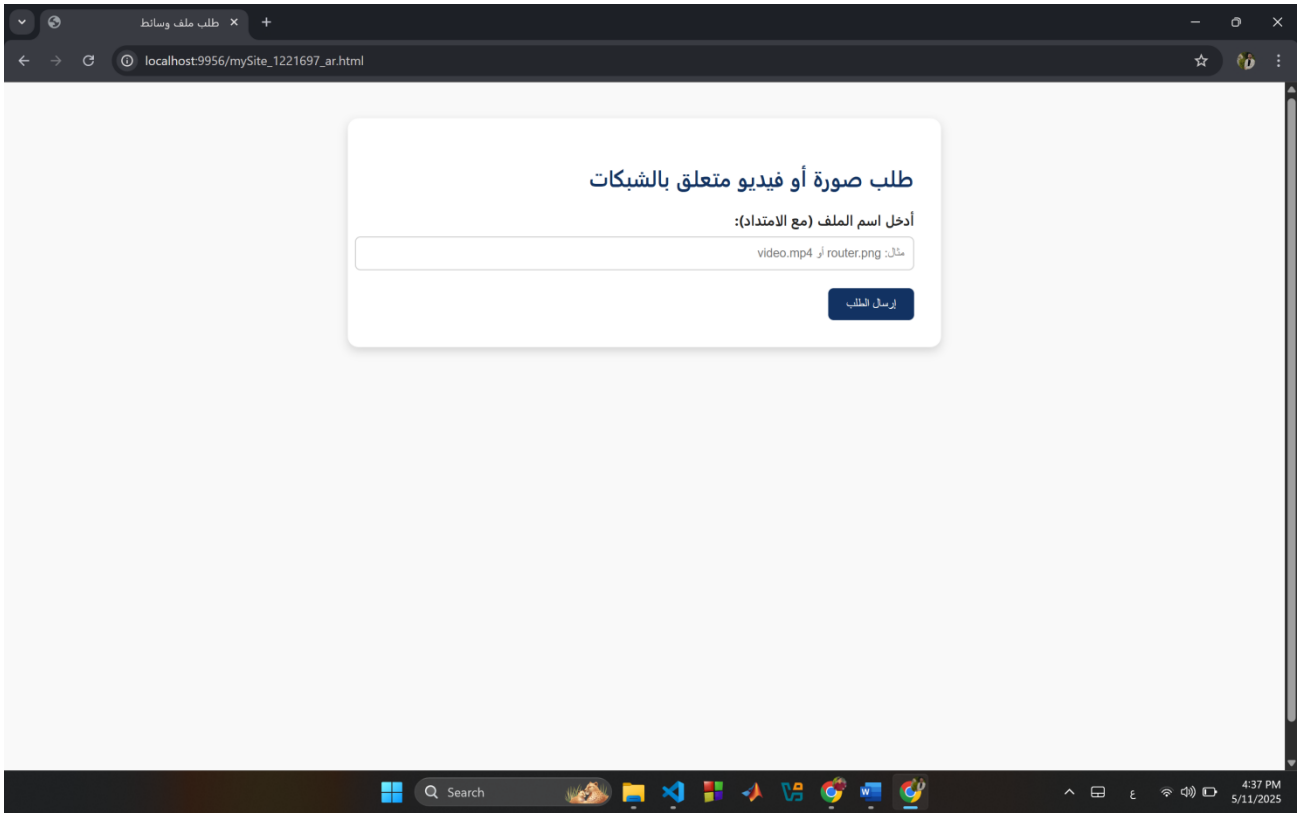

Figure 23: Supporting Material Arabic Webpage

HTML Code:

HTML Code for the Main English Webpage, (the Arabic version almost the same):

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>ENCS3320-Webserver</title>
    <!-- Link to external stylesheet -->
    <link rel="stylesheet" href="../css/style.css">
</head>
<body>
    <!-- Language Switcher Button -->
    <div class="lang-switch-container">
        <a href="main_ar.html" class="lang-btn" title="Switch to Arabic">
            <img src="../imgs/sa.jpeg" alt="Arabic" class="flag-icon">
        </a>
    </div>

    <!-- Page Header -->
    <header>
        <h1>Welcome to ENCS3320 - Computer Networks Webserver</h1>
    </header>

    <!-- Team Members Section -->
    <section class="team">
        <h2>The Team</h2>
        <div class="team-grid">
            <!-- Member 1: Amir -->
            <div class="member-card">
                <img src="../imgs/Amir.jpeg" alt="Amir Al-Rashaydah">
                <h3>Amir Al-Rashaydah</h3>
                <p><strong>ID:</strong> 1222596</p>
                <ul>
                    <li><strong>Skills:</strong> Python, C, Computer Architecture, Dynamic Programming</li>
                    <li><strong>Projects:</strong> Medical test management system, Palestinian-District-Data-Management</li>
                    <li><strong>Hobbies:</strong> Gaming, Coding</li>
                </ul>
            </div>

            <!-- Member 2: Aws -->
            <div class="member-card">
                <img src="../imgs/Aws.jpeg" alt="Aws Hammad">
                <h3>Aws Hammad</h3>
                <p><strong>ID:</strong> 1221697</p>
                <ul>
                    <li><strong>Skills:</strong> Python, Networking, AI/ML, Computer Architecture</li>
                    <li><strong>Projects:</strong> SmartDrop GUI, Socket Web Server, Bin Packing Problem Solution</li>
                    <li><strong>Hobbies:</strong> Coding, Reading</li>
                </ul>
            </div>

            <!-- Member 3: Ibraheem -->
            <div class="member-card">
                <img src="../imgs/Ibraheem.jpeg" alt="Ibraheem Sleet">
                <h3>Ibraheem Sleet</h3>
                <p><strong>ID:</strong> 1220200</p>
                <ul>
                    <li><strong>Skills:</strong> Python, HTML/CSS</li>
                    <li><strong>Projects:</strong> OS Project</li>
                    <li><strong>Hobbies:</strong> Learning, Coding</li>
                </ul>
            </div>
        </div>
    </section>

    <!-- Network Security Section -->
    <section class="topic">
        <h2 style="color:#003366;">🛡️ Network Security</h2>

        <p>
```

```
        <p>
            <strong>The Internet</strong> was <em>not originally designed</em> with strong security in mind.
            The early vision was a transparent and open environment for trusted users.
            As usage evolved, <span style="color:red;"><strong>security became critical</strong></span> across all layers of network communication
        </p>

        <!-- Network Security Image -->
        <img src="../imgs/network.jpeg" alt="Network Security Concept" class="topic-image">

        <!-- Network Threats Section -->
        <h3 style="color:#003366;">⚠️ Common Network Threats</h3>
        <ul>
            <li><strong>Malware:</strong> Software that harms or steals information. Includes:
                <ul>
                    <li><em>Viruses</em> – require user action to activate.</li>
                    <li><em>Worms</em> – spread automatically across systems.</li>
                    <li><em>Spyware</em> – records user activity without consent.</li>
                </ul>
            </li>
            <li><strong>Botnets:</strong> Infected devices controlled remotely to perform attacks.</li>
            <li><strong>Packet Sniffing:</strong> Capturing data sent across unsecured networks.</li>
            <li><strong>IP Spoofing:</strong> Faking the source IP to impersonate another device.</li>
        </ul>

        <!-- Security Design Goals -->
        <h3 style="color:#003366;">🎯 Security Design Goals</h3>
        <ol>
            <li><strong>Confidentiality</strong> – Prevent unauthorized access to sensitive data.</li>
            <li><strong>Integrity</strong> – Ensure information is not altered during transmission.</li>
            <li><strong>Availability</strong> – Keep systems operational even under attack.</li>
            <li><strong>Authentication</strong> – Verify the identity of communicating parties.</li>
        </ol>


        <p>
            Tools like <strong>Wireshark</strong> can help analyze traffic and detect suspicious activity.
            As networks grow, <em>every protocol must now include security considerations by design</em>.
        </p>
    </section>

    <!-- Footer Links -->
    <footer>
        <p>
            🔗 <a href="https://gaia.cs.umass.edu/kurose_ross/index.php" target="_blank">Textbook Website</a> |
            <a href="https://www.birzeit.edu/" target="_blank">Birzeit University</a> |
            <a href="mySite_1221697_en.html">Local Site Page</a>
        </p>
    </footer>

    <!-- JavaScript for Language Switcher Visibility on Scroll -->
    <script>
        const langSwitcher = document.querySelector('.lang-switch-container');

        // Hide language switcher when scrolling down
        window.addEventListener('scroll', () => {
            if (window.scrollY > 50) {
                langSwitcher.style.opacity = '0';
                langSwitcher.style.pointerEvents = 'none'; // Disable click interaction
            } else {
                langSwitcher.style.opacity = '1';
                langSwitcher.style.pointerEvents = 'auto'; // Re-enable click interaction
            }
        });
    </script>
</body>
</html>
```

Figure 24: HTML Code for Main English Webpage

HTML Code for the Supporting Material English Webpage, (the Arabic version almost the same):

```html
<!DOCTYPE html>
<html lang="en"> <!-- Set language to English -->
<head>
    <meta charset="UTF-8">
    <title>Media Request</title>
    <!-- Link to external stylesheet -->
    <link rel="stylesheet" href="../css/style.css">
    <!-- Inline styling for the form appearance -->
    <style>
        /* Centered form container with padding and styling */
        .form-container {
            max-width: 600px;
            margin: 40px auto;
            background-color: #fff;
            padding: 30px;
            border-radius: 12px;
            box-shadow: 0 4px 10px rgba(0, 0, 0, 0.15);
            text-align: left; /* Left-align for English */
        }
        /* Form title styling */
        .form-container h2 {
            color: #003366;
            margin-bottom: 20px;
        }
        /* Label styling */
        label {
            font-weight: bold;
        }
        /* Input field styling */
        input[type="text"] {
            width: 100%;
            padding: 10px;
            margin-top: 8px;
            margin-bottom: 20px;
            border: 1px solid #ccc;
            border-radius: 6px;
        }
        /* Submit button styling */
        input[type="submit"] {
            background-color: #003366;
            color: white;
            padding: 10px 20px;
            border: none;
            border-radius: 6px;
            cursor: pointer;
        }
        /* Hover effect for submit button */
        input[type="submit"]:hover {
            background-color: #005599;
        }
    </style>
</head>
<body>
    <!-- Form Section to Request Media -->
    <div class="form-container">
        <h2>Request an Image or Video Related to Networking</h2>
        <!-- Form submits a GET request to the server at /handle -->
        <form action="/handle" method="get">
            <!-- Label for the filename input field -->
            <label for="filename">Enter the file name (with extension):</label>
            <!-- Text input to enter the file name -->
            <input type="text" id="filename" name="filename" placeholder="e.g., router.png or video.mp4" required>
            <!-- Submit button to submit the request -->
            <input type="submit" value="Submit Request">
        </form>
    </div>

</body>
</html>
```

Figure 25: HTML Code for Supporting Material English Webpage

CSS Code:

CSS Code that style all the Webpages in a good way:

```css
/*Global Styles (LTR by default)*/
body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    background-color: #f9f9f9;
    color: #222;
    margin: 0;
    padding: 0;
    min-height: 100vh; /* Ensure gradient covers full viewport height */
}

/*Header Styling*/
header {
    background: linear-gradient(135deg, #e6f3ff 0%, #cce6ff 100%); /* Subtle blue gradient */
    padding: 20px;
    color: #003366; /* Darker blue text for contrast */
    text-align: center;
    box-shadow: 0 2px 6px rgba(0, 0, 0, 0.1);
}

/* Header Title Styling */
header h1 {
    color: #0055aa;
    margin: 0;
    font-size: 2.2em;
    text-shadow: 1px 1px 2px rgba(0, 0, 0, 0.2);
}

/*Team Section Layout*/
.team {
    padding: 40px 20px;
    text-align: center;
}

/* Team Section Title Styling */
.team h2 {
    color: #003366;
    margin-bottom: 20px;
}
```

```css
/* Grid Layout for Team Members */
.team-grid {
    display: flex;
    flex-wrap: wrap;
    justify-content: center;
    gap: 30px;
}

/*Team Member Card Styling*/
.member-card {
    background-color: #ffffff;
    border-radius: 12px;
    padding: 20px;
    width: 300px;
    box-shadow: 0 4px 10px rgba(0, 0, 0, 0.15);
    text-align: center;
}

/* Member Profile Image Styling */
.member-card img {
    width: 120px;
    height: 120px;
    object-fit: cover;
    border-radius: 50%;
    margin-bottom: 10px;
    border: 3px solid #003366;
}

/* Member Name Styling */
.member-card h3 {
    margin: 10px 0 5px;
    color: #003366;
}

/* List of Skills, Projects, Hobbies */
.member-card ul {
    list-style-type: none;
    padding: 0;
```

```
    text-align: left; /* Left align for LTR */
}

/*Topic Section Styling*/
.topic {
    background-color: #eef4fb;
    padding: 40px 20px;
    text-align: left;
    max-width: 800px;
    margin: 40px auto;
    border-radius: 12px;
    box-shadow: 0 4px 10px rgba(0, 0, 0, 0.05);
}

/* Topic Title Styling */
.topic h2 {
    color: #003366;
}

/* Topic Image Styling */
.topic-image {
    max-width: 100%;
    display: block;
    margin: 20px auto;
    border-radius: 8px;
}

/*Footer Styling*/
footer {
    text-align: center;
    padding: 20px;
    background-color: #003366;
    color: white;
    font-size: 15px;
}

/* Footer Links Styling */
footer a {
    color: #00c6ff;
    text-decoration: none;
    margin: 0 10px;
}

/* Footer Links Hover Effect */
footer a:hover {
    text-decoration: underline;
}

/*Language Switch Button Styling*/
.lang-switch-container {
    position: fixed;
    top: 20px;
    right: 20px;
    z-index: 999;
    transition: opacity 0.1s ease;
}

/* Rounded Button with Gradient Background */
.lang-btn {
    display: inline-block;
    background: linear-gradient(to left, #003366, #005599);
    border-radius: 50%;
    width: 45px;
    height: 45px;
    box-shadow: 0 4px 10px rgba(0, 0, 0, 0.25);
    transition: transform 0.3s ease, background 0.3s ease;
    padding: 0;
    overflow: hidden;
    text-align: center;
    line-height: 45px;
}

/* Button Hover Effect */
.lang-btn:hover {
    transform: scale(1.1);
```

```
    background: linear-gradient(to left, #005599, #0077cc);
}

/* Make Flag Icon Cover Button Fully */
.flag-icon {
    width: 100%;
    height: 100%;
    object-fit: cover;
    display: block;
}

/*RTL Support (Overrides When dir="rtl" is Set)*/

/* Right-to-left direction for body */
body[dir="rtl"] {
    direction: rtl;
    text-align: right;
}

/* Right-align headings and paragraphs in RTL */
body[dir="rtl"] h1,
body[dir="rtl"] h2,
body[dir="rtl"] h3,
body[dir="rtl"] h4,
body[dir="rtl"] p {
    direction: rtl;
    text-align: right;
    unicode-bidi: plaintext;
}

/* Lists in RTL: start bullets/numbers inside */
body[dir="rtl"] ul,
body[dir="rtl"] ol {
    list-style-position: inside;
    padding-right: 0;
    margin-right: 0;
}

 /* List items aligned to the right in RTL */
 body[dir="rtl"] li {
     direction: rtl;
     text-align: right;
     unicode-bidi: plaintext;
 }

```

Figure 26: CSS Code

Web Server Implementation and Code:

We implemented our web server using socket programming in Python. The server is configured to listen for client connections on port 9956, which is derived from one of our team member's student ID. The server receives HTTP requests from web browsers (clients) using the TCP connection protocol, since HTTP/1.1 operates over TCP to ensure reliable communication between clients and the server.

The server's implementation begins by importing the socket library, as shown below:

```
1   import socket
2   import os
3   from datetime import datetime
4
5   # Server Configuration
6   HOST = '0.0.0.0'    # Listen on all network interfaces
7   PORT = 9956         # Amir's Id = 1222596 => XY = 56
8
122 # Start the server and wait for incoming connections
123 def start_server():
124     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
125         server.bind((HOST, PORT))
126         server.listen(5)
127         print(f"🌐 Server running at http://localhost:{PORT}/")
128         print(f"Server port: {PORT}")
129
130         # Handle incoming client connections indefinitely
131         while True:
132             conn, addr = server.accept()
133             handle_request(conn, addr)
134
135 # Run server when script is executed directly
136 if __name__ == "__main__":
137     start_server()
```

Figure 27: importing libraries in server code

As shown in the segment of the code above, firstly, we import the socket library and the OS library (used to list the files that available in the server dynamically). Then we initialize the server to listen to the HTTP request on port 9956, we defined the start_server() function to initialize and start the serverand initialize TCP socket using the function socket (AF_INF, SOCK_STREM). After that we connect the current IP address of the server and the port number with the socket using bind function. Then, let the server listens to the requests (Up to 5 clients) on the specific port (9956). Finally the server is ready to receive the requests.

HTTP Response Builder Function:

The build_response function in our implementation is responsible for constructing the HTTP response that is sent back to the client. This response includes the HTTP status code, the status message, and the content type based on the requested file or action.

As shown in the code segment below, the function takes the following parameters:

status_code: The HTTP status code to send (e.g., 200 OK, 307 Temporary Redirect, or 404 Not Found).

content: The body content to send back to the client (optional, defaults to empty).

content_type: The MIME type of the content (e.g., text/html, image/png).

redirect_location: An optional URL for redirection if a 307 Temporary Redirect is needed.

The function builds the status line and appends standard HTTP headers such as Content-Type, Content-Length, and Connection.

If the status code is 307, the function adds a Location header with the redirect URL and returns it immediately, as no content body is needed in that case.

For 200 OK and 404 Not Found responses, the function combines the headers with the provided content and returns the complete HTTP response.

```
23  # Build HTTP Response based on status code and content
24  def build_response(status_code, content=b"", content_type="text/html", redirect_location=None):
25      status_messages = {
26          200: "OK",
27          404: "Not Found",
28          307: "Temporary Redirect",
29      }
30
31      # Prepare status line
32      header = f"HTTP/1.1 {status_code} {status_messages[status_code]}\r\n"
33
34      # Handle redirection if needed
35      if status_code == 307 and redirect_location:
36          header += f"Location: {redirect_location}\r\n\r\n"
37          return header.encode()
38
39      # Standard headers for successful responses
40      header += f"Content-Type: {content_type}\r\n"
41      header += f"Content-Length: {len(content)}\r\n"
42      header += "Connection: close\r\n\r\n"
43
44      return header.encode() + content
```

Figure 28: build_response function code

Error Handling Functionality:

The error handling in our server is designed to respond when a requested file does not exist on the server. As shown in the code segment below, if the requested file cannot be found, the server dynamically generates an HTML error page and sends it to the client as a 404 Not Found response. This error page displays the message "The file is not found" in red text on the webpage body, and shows the client's IP address and port number, along with the server port. In addition, the browser tab title is set to "Error 404", making it clear to the user that the file they requested does not exist. The HTML content for this error page is written directly in the Python code, meaning no separate .html file is needed. This allows the server to immediately generate and return the error page whenever required.

The relevant portion of the server's Python code is shown below:

```
84          # Send file content if exists
85          if os.path.isfile(file_path):
86              with open(file_path, "rb") as f:
87                  content = f.read()
88              response = build_response(200, content, get_content_type(file_path))
89              print(f"Response: 200 OK - Serving {file_path}")
90          else:
91              # Send 404 Not Found response with client details
92              html = f"""
93              <html>
94              <head>
95                  <title>Error 404</title>
96                  <style>
97                      body {{ font-family: Arial, sans-serif; margin: 40px; }}
98                      .error {{ color: red; font-size: 24px; }}
99                      .details {{ margin-top: 20px; }}
100                 </style>
101             </head>
102             <body>
103                 <h1 class="error">The file is not found</h1>
104                 <div class="details">
105                     <p>Client IP: {addr[0]}</p>
106                     <p>Client Port: {addr[1]}</p>
107                     <p>Server Port: {PORT}</p>
108                 </div>
109             </body>
110             </html>
111             """
112             response = build_response(404, html.encode())
113             print(f"Response: 404 Not Found - Requested file not found")
114
```

Figure 29: Error handling code

Client Request Handling Function:

The handle_request function is designed to process incoming client requests and respond appropriately. It starts by receiving the HTTP request from the client and printing detailed information to the terminal, including the timestamp, client's IP address, client port, and server port. The server then attempts to extract the requested path from the HTTP request. If the path is missing or invalid, it defaults to the root path "/". Based on the extracted path, the server checks if the request is for the main English page ("/", "/en", "/index.html", or "/main_en.html") or the Arabic page ("/ar" or "/main_ar.html") and responds with the corresponding HTML file. If the request starts with "/handle?filename=", the server extracts the filename from the query and determines whether it is a video (".mp4") or another file type. For videos, the server issues a 307 Temporary Redirect to a Google video search, while other file types trigger a redirect to a Google image search. If the request does not match any of these predefined paths, the server attempts to locate the requested file on the server's file system. If the file is not found, the server prepares to send a 404 Not Found response. This process ensures the server is capable of handling valid requests, redirecting to external search results when needed, and properly managing errors for unavailable content.

```
46  # Handle client request and send appropriate response
47  def handle_request(conn, addr):
48      try:
49          # Receive and decode client request
50          request = conn.recv(1024).decode()
51          print(f"[{datetime.now()}] Request from {addr[0]}:{addr[1]} to server port {PORT}")
52          print(f"Request details:\n{request}\n{'-'*50}")
53
54          # Parse requested path or fallback to root
55          try:
56              path = request.split(" ")[1]
57          except:
58              path = "/"
59
60          # Handle language-specific or root paths
61          if path in ["/", "/en", "/index.html", "/main_en.html"]:
62              file_path = "html/main_en.html"
63          elif path in ["/ar", "/main_ar.html"]:
64              file_path = "html/main_ar.html"
65          # Handle dynamic redirection for /handle?filename=
66          elif path.startswith("/handle?filename="):
67              filename = path.split("filename=")[-1]
68              ext = os.path.splitext(filename)[-1].lower()
69              # Redirect to Google Video or Image search based on file type
70              if ext == ".mp4":
71                  redirect_url = f"https://www.google.com/search?q={filename}&tbm=vid"
72              else:
73                  redirect_url = f"https://www.google.com/search?q={filename}&tbm=isch"
74              response = build_response(307, redirect_location=redirect_url)
75              conn.sendall(response)
76              conn.close()
77              return
78          else:
79              # Clean the path and attempt to locate the file
80              file_path = path.strip("/")
81              if not os.path.isfile(file_path):
82                  file_path = "html" + path
```

Figure 30: handle_request function code

Requests and Response Testing:

When the client (browser) request ‘/’, ‘/index.html’, ‘/main_en.html’ or ‘/en’:

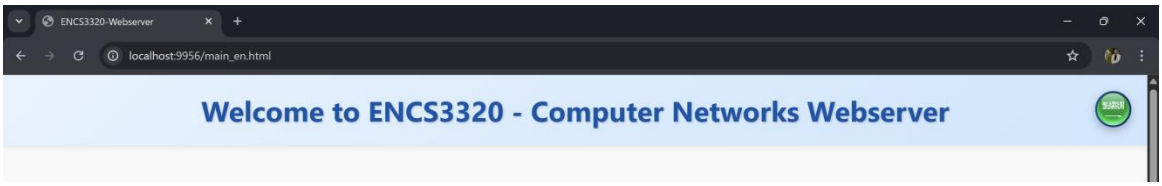

Figure 31: Browser Request for main English webpage

The server receives a request for the main English webpage, and then receives a request for each object in this page, such as photos, CSS file and etc:

```
Server running at http://localhost:9956/
Server port: 9956
[2025-05-11 10:48:31.835160] Request from 127.0.0.1:56589 to server port 9956
Request details:
GET / HTTP/1.1
Host: localhost:9956
Connection: keep-alive
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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/136.0.0.0 Safari/537.36
Sec-Purpose: prefetch;prerender
Purpose: prefetch
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: 200 OK - Serving html/main_en.html
[2025-05-11 10:48:32.085400] Request from 127.0.0.1:56593 to server port 9956
Request details:
GET /css/style.css HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
Sec-Purpose: prefetch;prerender
sec-ch-ua-mobile: ?0
Accept: text/css,*/*;q=0.1
Purpose: prefetch
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: style
Referer: http://localhost:9956/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```

Figure 32: Requests for main English webpage

```
--------------------------------------------------
Response: 200 OK - Serving css/style.css
[2025-05-11 10:48:32.152992] Request from 127.0.0.1:56595 to server port 9956
Request details:
GET /imgs/sa.jpeg HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
Sec-Purpose: prefetch;prerender
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Purpose: prefetch
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9956/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


--------------------------------------------------
Response: 200 OK - Serving imgs/sa.jpeg
[2025-05-11 10:48:32.155017] Request from 127.0.0.1:56596 to server port 9956
Request details:
GET /imgs/Amir.jpeg HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
Sec-Purpose: prefetch;prerender
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Purpose: prefetch
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9956/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```

Figure 33: capturing request and response 1

```
--------------------------------------------------
Response: 200 OK - Serving imgs/Amir.jpeg
[2025-05-11 10:48:32.157184] Request from 127.0.0.1:56597 to server port 9956
Request details:
GET /imgs/Aws.jpeg HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
Sec-Purpose: prefetch;prerender
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Purpose: prefetch
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9956/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


--------------------------------------------------
Response: 200 OK - Serving imgs/Aws.jpeg
[2025-05-11 10:48:32.396441] Request from 127.0.0.1:56600 to server port 9956
Request details:
GET /imgs/Ibraheem.jpeg HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
Sec-Purpose: prefetch;prerender
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Purpose: prefetch
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9956/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```

Figure 34: Requests for main English webpage 2

```
--------------------------------------------------
Response: 200 OK - Serving imgs/Ibraheem.jpeg
[2025-05-11 10:48:32.470507] Request from 127.0.0.1:56601 to server port 9956
Request details:
GET /imgs/network.jpeg HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
Sec-Purpose: prefetch;prerender
sec-ch-ua-mobile: ?0
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Purpose: prefetch
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:9956/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


--------------------------------------------------
Response: 200 OK - Serving imgs/network.jpeg
[2025-05-11 10:48:55.226628] Request from 127.0.0.1:56602 to server port 9956
Request details:
GET /main_ar.html HTTP/1.1
Host: localhost:9956
Connection: keep-alive
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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/136.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:9956/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```

Figure 35: Requests for main English webpage 3

The Response:

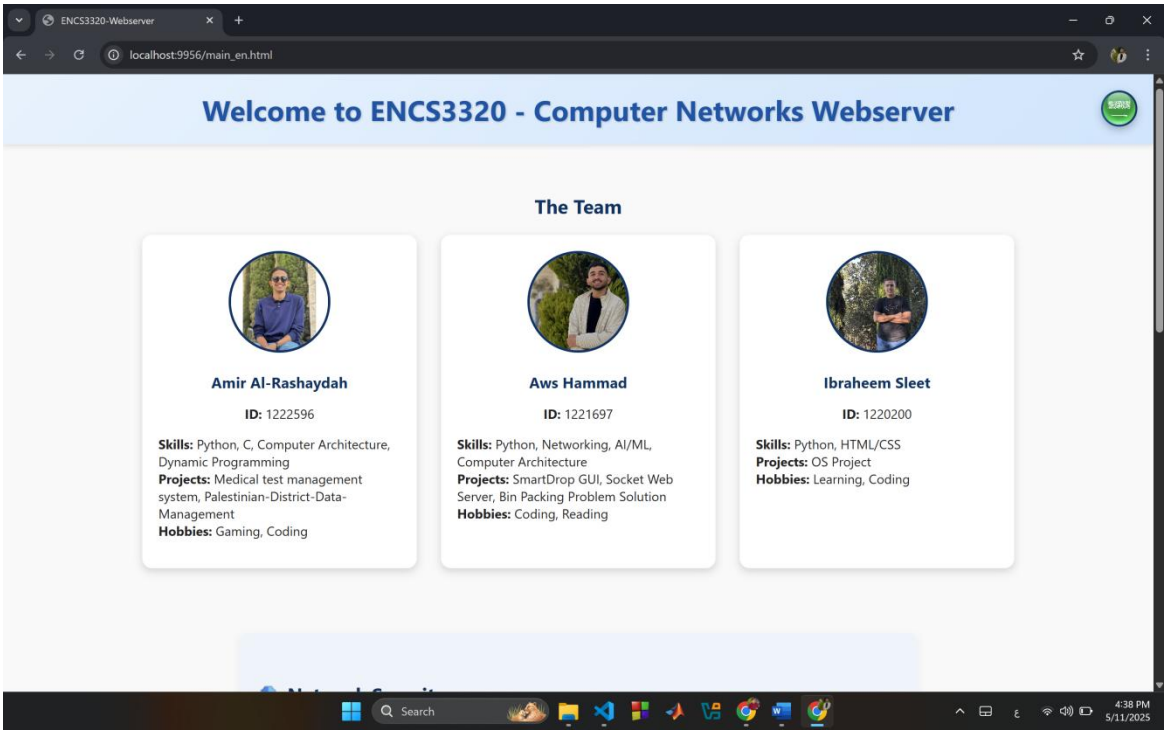

Figure 36: Response for the main English webpage

When the client (browser) request ‘/ar’, or ‘/main_ar.html’:

```
--------------------------------------------------
Response: 200 OK - Serving html/main_ar.html
[2025-05-11 10:48:55.540983] Request from 127.0.0.1:56619 to server port 9956
Request details:
GET /css/style.css HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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:9956/main_ar.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


--------------------------------------------------
Response: 200 OK - Serving css/style.css
[2025-05-11 10:48:55.542276] Request from 127.0.0.1:56620 to server port 9956
Request details:
GET /css/rtl.css HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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:9956/main_ar.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```

Figure 37: Requests for main Arabic webpage 1


```
--------------------------------------------------
Response: 200 OK - Serving html/main_ar.html
[2025-05-11 10:48:55.540983] Request from 127.0.0.1:56619 to server port 9956
Request details:
GET /css/style.css HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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:9956/main_ar.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


--------------------------------------------------
Response: 200 OK - Serving css/style.css
[2025-05-11 10:48:55.542276] Request from 127.0.0.1:56620 to server port 9956
Request details:
GET /css/rtl.css HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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:9956/main_ar.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```


Figure 38: Requests for main Arabic webpage 2

The Response:

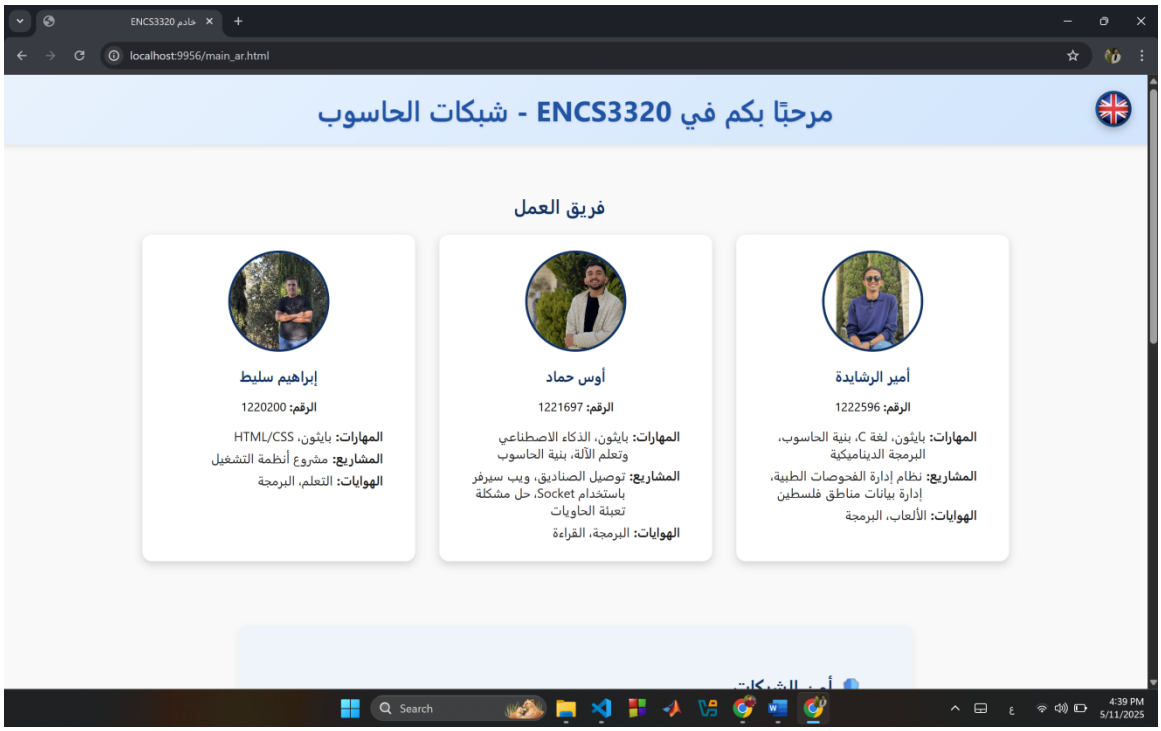

Figure 39: Response for the main Arabic webpage

When the client (browser) request a file that not found in the server (/Fr):

```
--------------------------------------------------
Response: 404 Not Found - Requested file not found
[2025-05-11 11:16:22.204469] Request from 127.0.0.1:57040 to server port 9956
Request details:
GET /Fr HTTP/1.1
Host: localhost:9956
Connection: keep-alive
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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/136.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```

Figure 40: Request for invalid file

The Response:

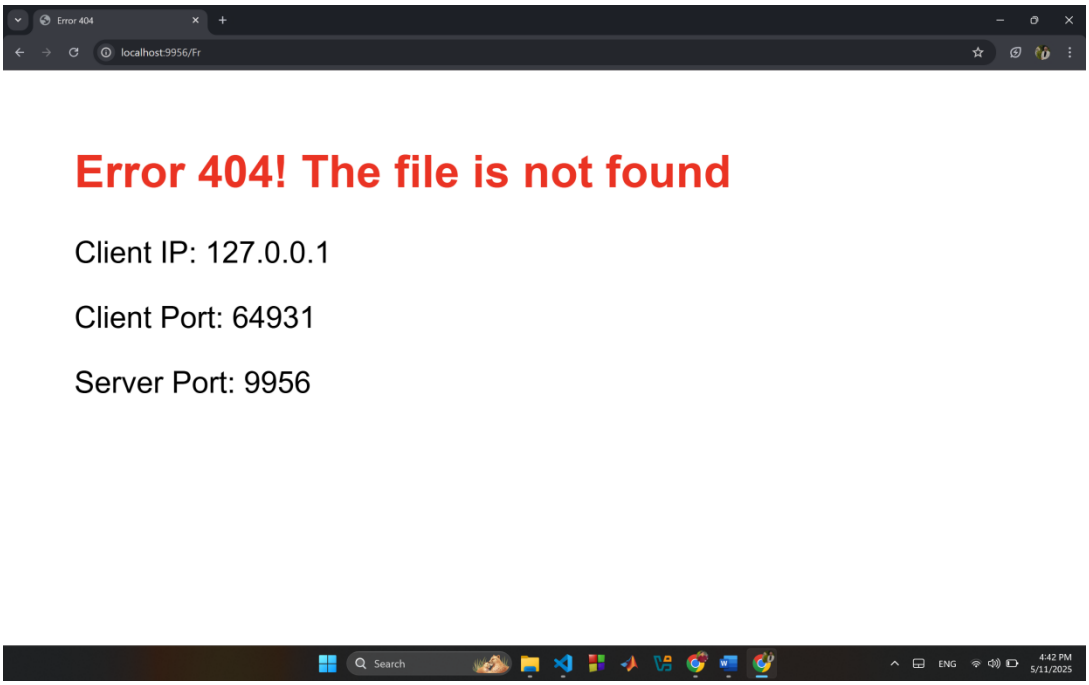

Figure 41: Response for invalid request

When the user search for an image in the supporting material webpage:

```
--------------------------------------------------
Response: 200 OK - Serving html/mySite_1221697_en.html
[2025-05-11 11:23:33.933014] Request from 127.0.0.1:57240 to server port 9956
Request details:
GET /css/style.css HTTP/1.1
Host: localhost:9956
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/136.0.0.0 Safari/537.36
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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:9956/mySite_1221697_en.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8


--------------------------------------------------
Response: 200 OK - Serving css/style.css
[2025-05-11 11:24:32.354982] Request from 127.0.0.1:57241 to server port 9956
Request details:
GET /mySite_1221697_en.html HTTP/1.1
Host: localhost:9956
Connection: keep-alive
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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/136.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```

Figure 42: Request for Supporting Material Webpage

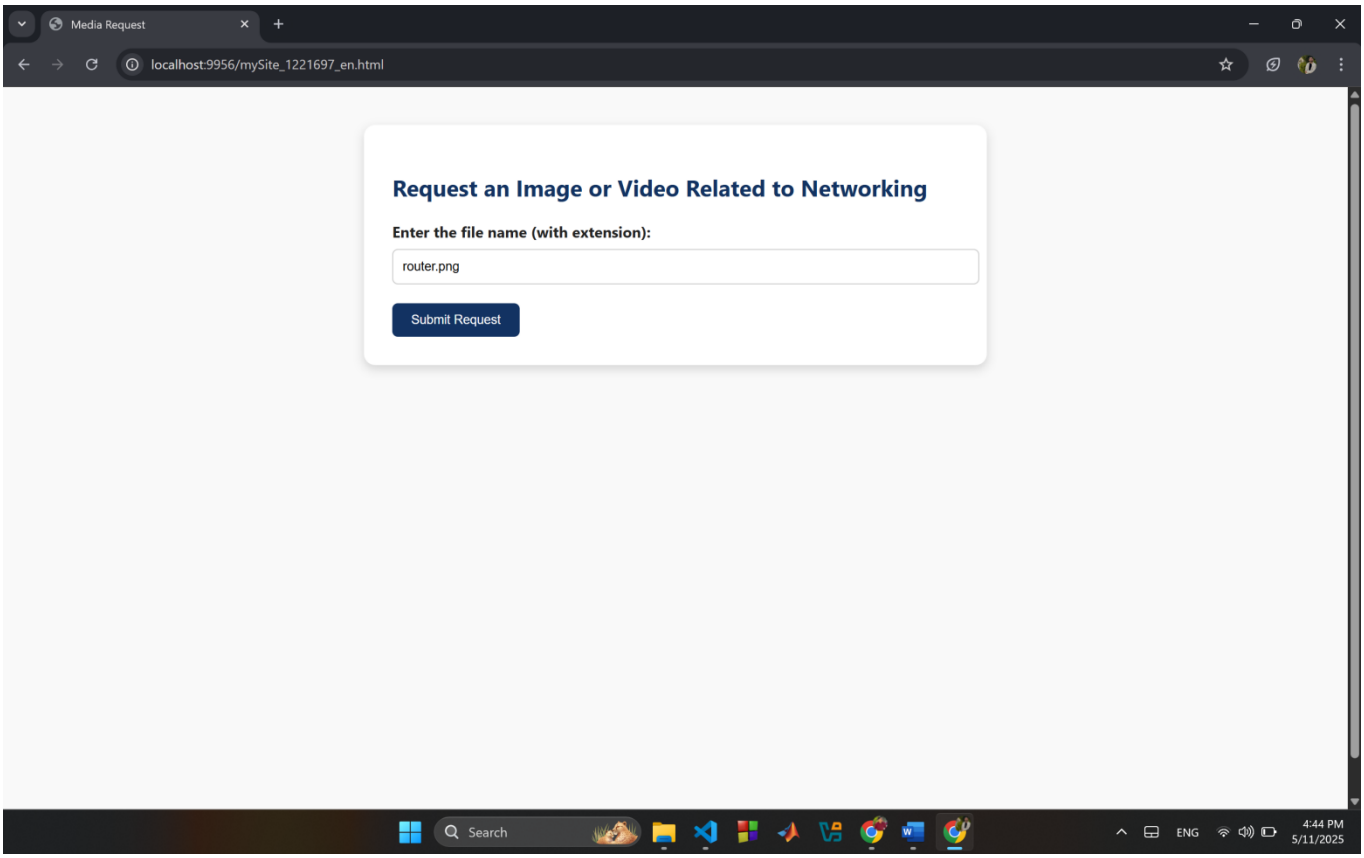

Figure 43: Search for image in supporting material

The request that the server received is:

```
------------------------------------------------
[2025-05-11 11:29:03.529372] Request from 127.0.0.1:57331 to server port 9956
Request details:
GET /handle?filename=router.png HTTP/1.1
Host: localhost:9956
Connection: keep-alive
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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/136.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:9956/mySite_1221697_en.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```

Figure 44: Request for image from Supporting Material

The Response:

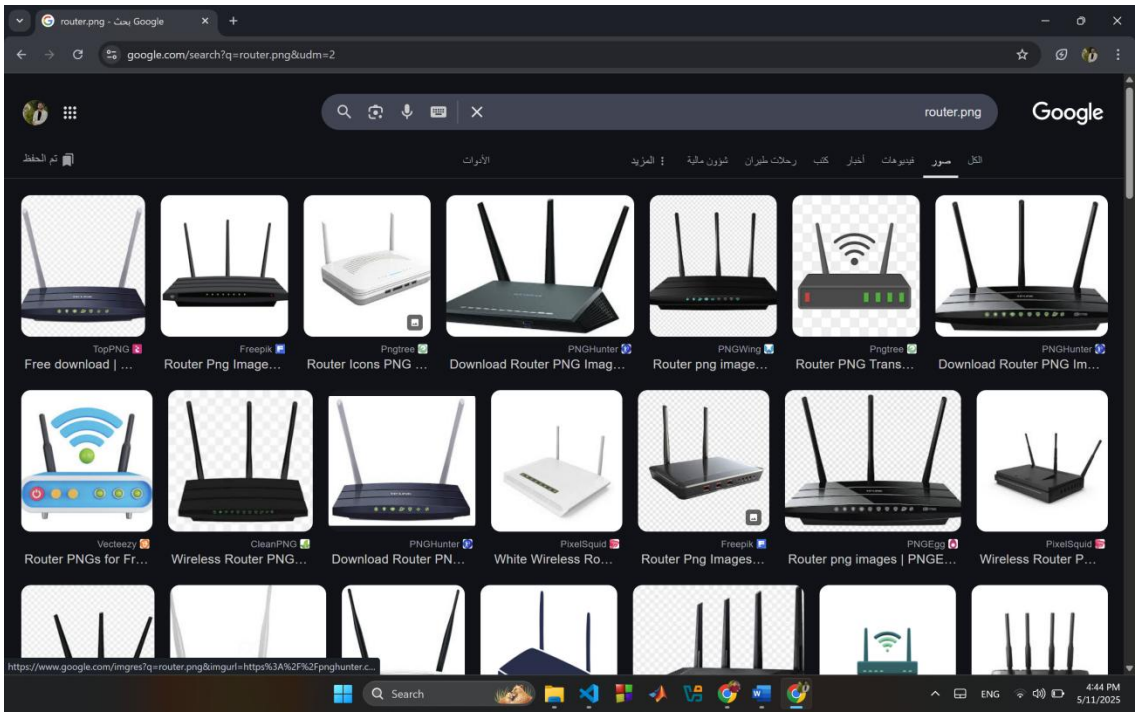

Figure 45: Response for the image

When the user search for a video in the supporting material webpage:

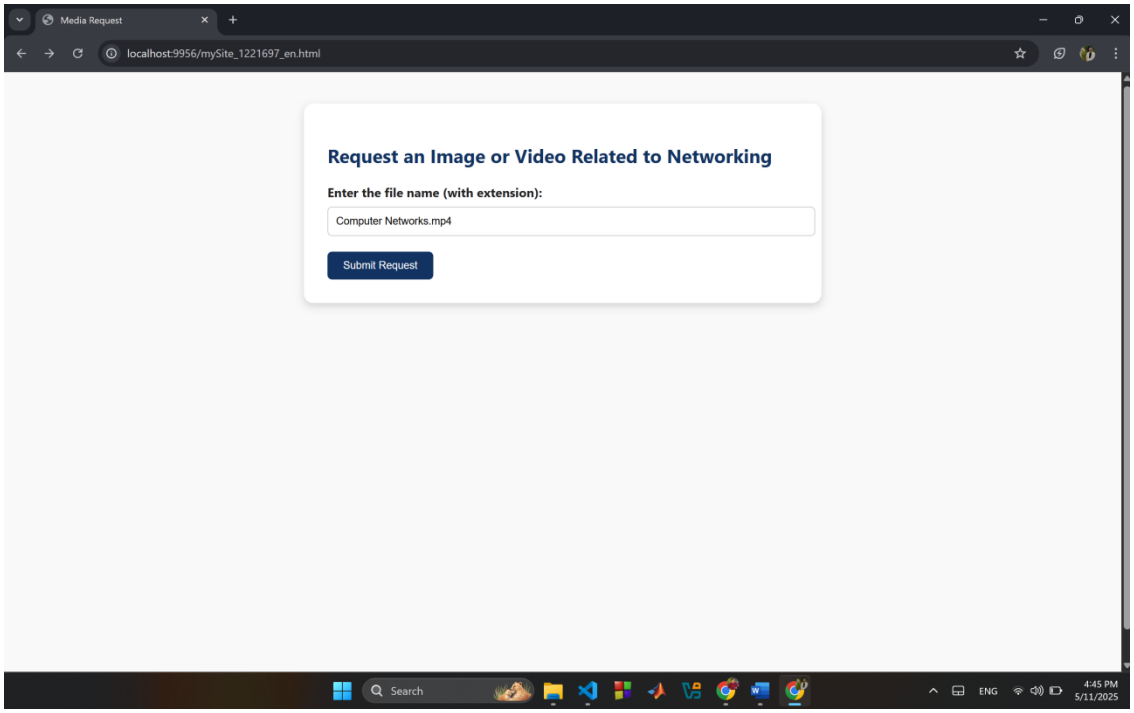

Figure 46: Request a video from Supporting Material

The request that the server received is:

```
------------------------------------------------
Response: 200 OK - Serving css/style.css
[2025-05-11 11:37:42.163615] Request from 127.0.0.1:57591 to server port 9956
Request details:
GET /handle?filename=Computer+Networks.mp4 HTTP/1.1
Host: localhost:9956
Connection: keep-alive
sec-ch-ua: "Chromium";v="136", "Google Chrome";v="136", "Not.A/Brand";v="99"
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/136.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:9956/mySite_1221697_en.html
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9,ar;q=0.8
```

Figure 47: Request for a video

The Response:

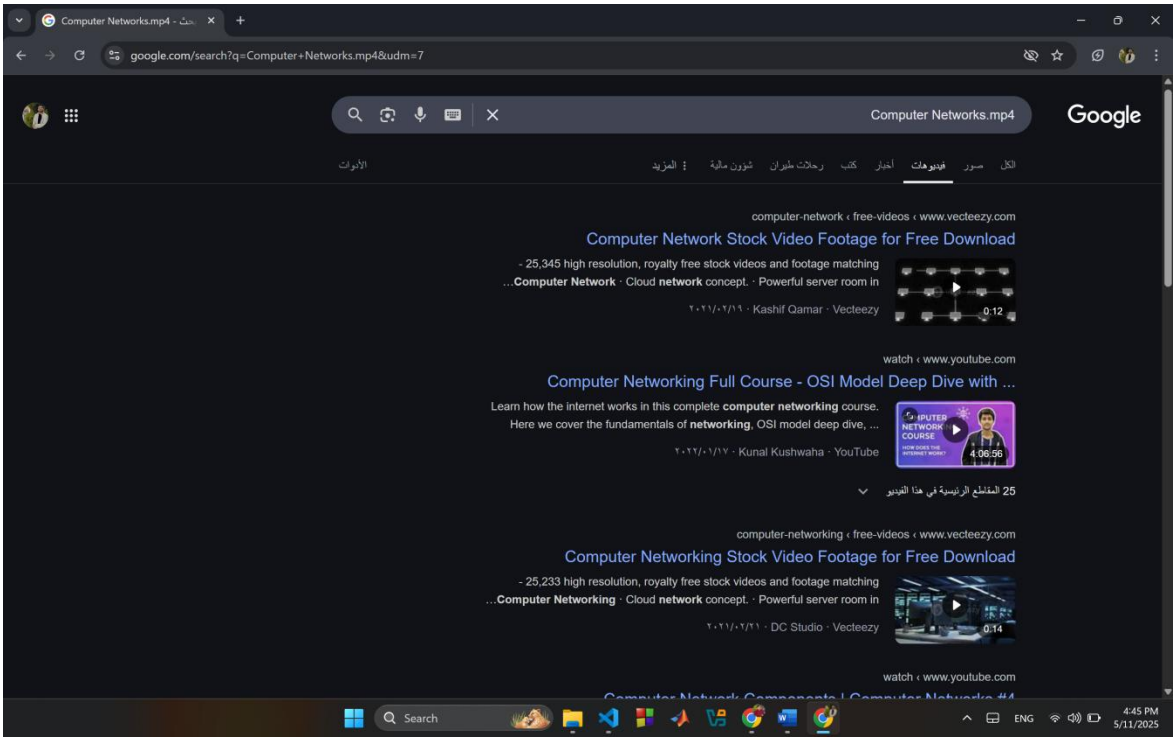

Figure 48: Response for the Video

When another device on the same local network request a webpage from the server (for example /main_en.html), The Requests that the server received, are:

```
[2025-05-11 11:54:28.642827] Request from 192.168.0.120:57738 to server port 9956
Request details:
GET /main_en.html HTTP/1.1
Host: 192.168.0.101:9956
Connection: keep-alive
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
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
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9


--------------------------------------------------
Response: 200 OK - Serving html/main_en.html
[2025-05-11 11:54:28.682626] Request from 192.168.0.120:57739 to server port 9956
Request details:
GET /css/style.css HTTP/1.1
Host: 192.168.0.101:9956
Connection: keep-alive
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
Accept: text/css,*/*;q=0.1
Referer: http://192.168.0.101:9956/main_en.html
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
```

Figure 49: Request for a webpage from another device

The Response:

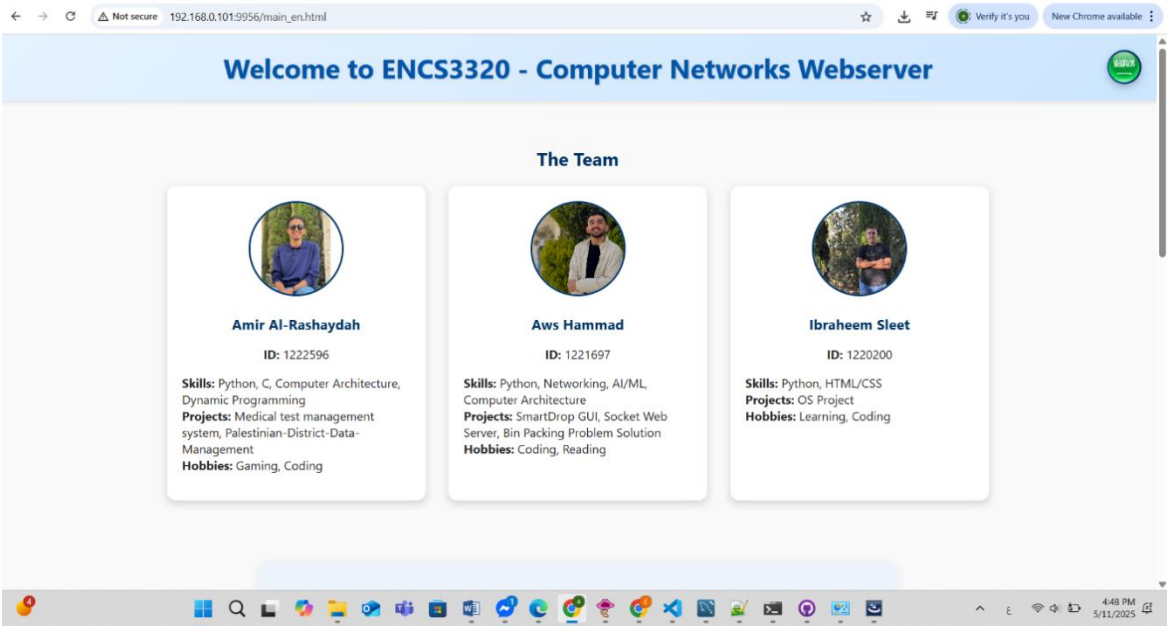

Figure 50: Server Response to another device request

When the user clicks on the Birzeit University website button:

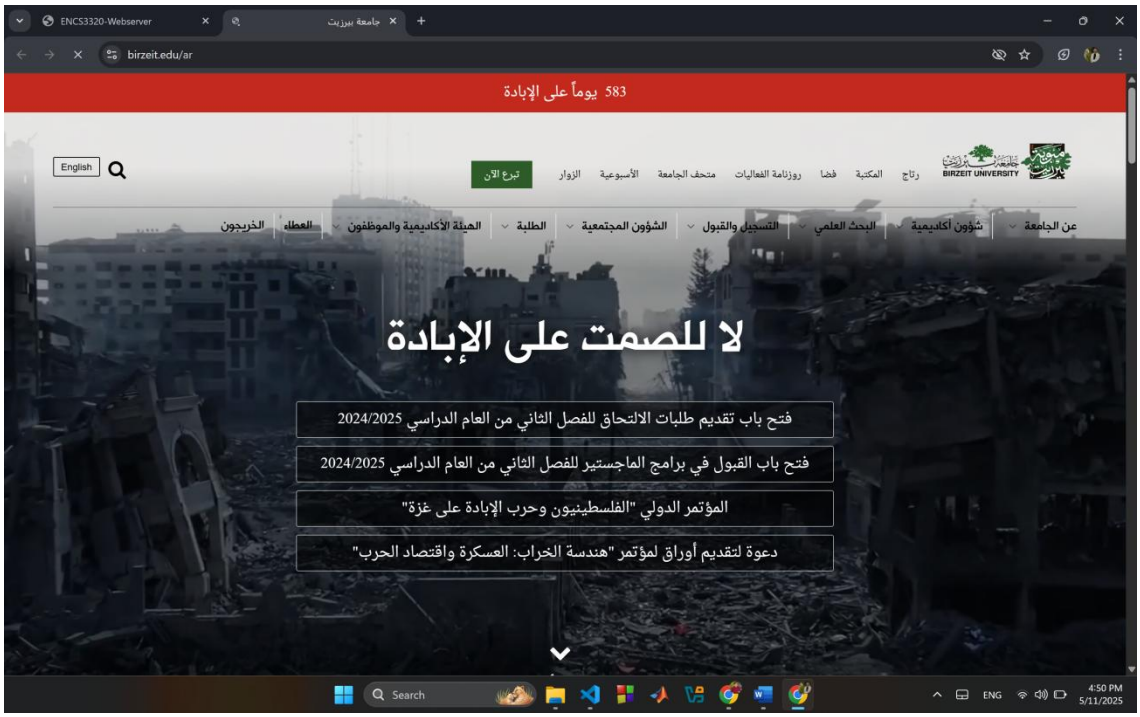

Figure 51: Birzeit University Website

When the user clicks on the textbook website button:

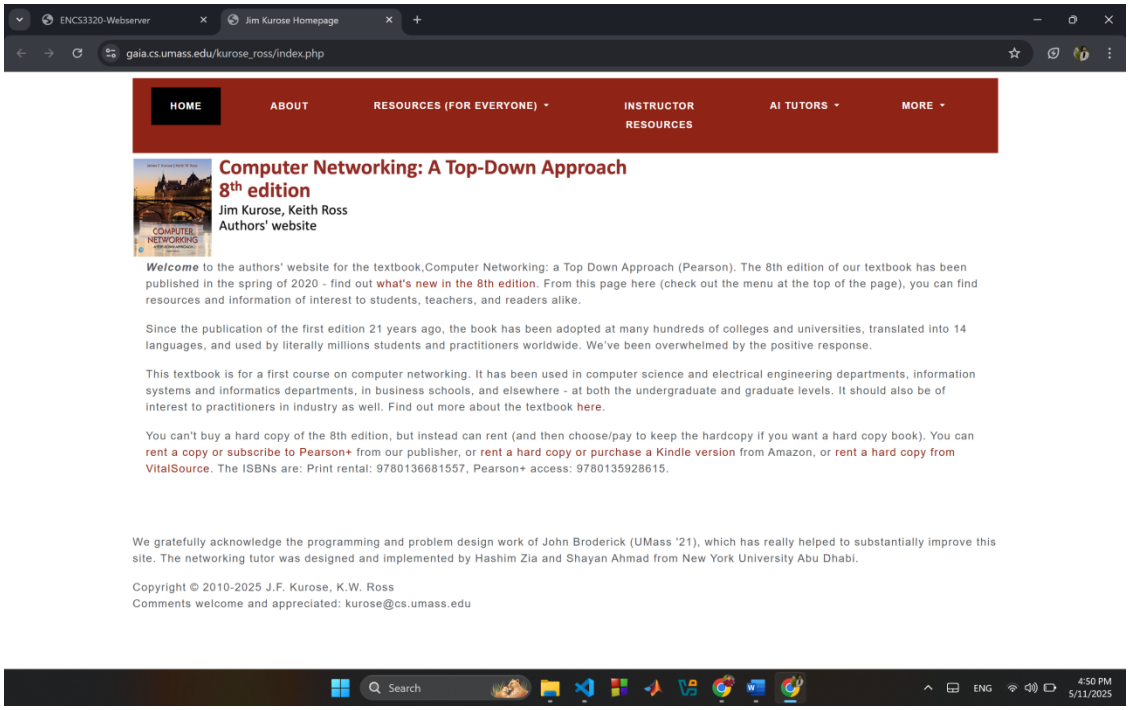

Figure 52: Textbook Website

3. Task 3 (Web Server):

This Task presents a hybrid client-server number guessing game using both TCP and UDP protocols, designed to support real-time multiplayer gameplay over a network. The server facilitates the game setup and control messages using TCP, including player registration, waiting room coordination, and game rules dissemination. Meanwhile, UDP is employed for fast-paced, low-latency gameplay during the guessing rounds, enabling real-time interaction between players and the server. The game supports 2 to 4 players, structured into multiple rounds where each player attempts to guess a randomly selected number within a limited time. The first player to guess correctly in each round wins that round, and the player with the most round victories is declared the overall winner. This hybrid approach leverages TCP's reliability for game state synchronization and UDP's speed for responsive gameplay, demonstrating effective use of network programming concepts in a real-time, interactive application.

3.1 Server Implementation and Code:

Initial segment of the server code:

In this project, we implement a hybrid client-server number guessing game using Python socket programming. In the initial segment of the server code, essential Python libraries such as socket, threading, random, and time are imported to facilitate network communication, concurrent execution, randomness in gameplay, and time-based control, respectively. The configuration section defines the core parameters of the game, including the server's IP (SERVER_HOST), the TCP and UDP ports used for communication (TCP_PORT, UDP_PORT), and game-specific constants like the minimum and maximum number of players, the number of sub-rounds (SUBROUNDS), the duration of each round (SUB_DURATION), and the valid range for guesses (RANGE_LOW to RANGE_HIGH). The global state section initializes key data structures such as clients (a dictionary mapping usernames to their TCP connections), udp_addrs (mapping usernames to their UDP address), and a lock to synchronize access to shared resources across threads. This setup ensures the server is well-prepared to handle multiple players in a structured, concurrent, and interactive environment.

The server's implementation begins by importing the socket library, configuration section and global state section as shown below:

```python
# server.py
import socket
import threading
import random
import time

# ── Configuration ──────────────────────────────────────────────
SERVER_HOST   = '0.0.0.0'
TCP_PORT      = 6000
UDP_PORT      = 6001

MIN_PLAYERS   = 2
MAX_PLAYERS   = 4
SUBROUNDS     = 6
SUB_DURATION  = 10     # seconds per round
RANGE_LOW     = 1
RANGE_HIGH    = 100

# ── Global State ───────────────────────────────────────────────
clients    = {}    # username -> tcp_conn
udp_addrs  = {}    # username -> (ip, udp_port)
lock       = threading.Lock()

```

Figure 53: initial segment of the server code

The accept_joins function:

The accept_joins function is responsible for managing incoming player connections over TCP until a specified number of players (target_count) has joined or an optional timeout is reached. It prompts each connecting client to provide a username using the format "JOIN <username>". If the username is unique, the server adds the player to the clients dictionary and proceeds with a simple UDP handshake by requesting the client's UDP port. This UDP address is stored in the udp_addrs dictionary for future gameplay communication. If a client provides a duplicate or invalid username, the server responds accordingly and closes the connection. Throughout the process, the function uses thread-safe operations with a lock to avoid race conditions and broadcasts each successful join event to all connected players using TCP.

```python
def accept_joins(tcp_sock, target_count, timeout=None):
    """
    Fill `clients` until we have target_count or timeout expires.
    If timeout is None, block indefinitely.
    """
    if timeout is not None:
        tcp_sock.settimeout(timeout)
    try:
        while len(clients) < target_count:
            conn, addr = tcp_sock.accept()
            conn.sendall(f"Enter your username: ({len(clients)}/{MAX_PLAYERS}):\n".encode())
            data = conn.recv(1024).decode().strip().split()
            if len(data)==2 and data[0].upper()=="JOIN":
                name = data[1]
                with lock:
                    if name in clients:
                        conn.sendall(b"Username taken\n")
                        conn.close()
                        continue
                    clients[name] = conn
                conn.sendall(f"Connected as {name}\n".encode())
                # UDP handshake
                conn.sendall(b"SEND_UDP_PORT <port>\n")
                parts = conn.recv(1024).decode().split()
                if len(parts)==2 and parts[0]=="SEND_UDP_PORT":
                    udp_addrs[name] = (addr[0], int(parts[1]))
                else:
                    udp_addrs[name] = (addr[0], UDP_PORT)
                broadcast_tcp(f"{name} joined ({len(clients)}/{MAX_PLAYERS})\n")
            else:
                conn.sendall(b"Invalid. Enter username:\n")
                conn.close()
    except socket.timeout:
        pass
    finally:
        tcp_sock.settimeout(None)
```

Figure 54: accept_joins function

Game Initialization:

The run_one_game() function starts by generating a random secret number between predefined bounds (RANGE_LOW and RANGE_HIGH) which players will attempt to guess. It broadcasts game setup information to all connected clients using TCP, including the number of players, the guessing range, and the number of subrounds with each round's duration. A UDP socket is then created and bound to the server to receive real-time guesses from the players.

```
69  def run_one_game():
70      """
71      Runs one secret-number game of up to SUBROUNDS × SUB_DURATION seconds.
72      Returns True if players want to start another game, False otherwise.
73      """
74      secret = random.randint(RANGE_LOW, RANGE_HIGH)
75      broadcast_tcp(
76          "\n=== NEW GAME ===\n"
77          f"{len(clients)} players, secret is 1–100.\n"
78          f"{SUBROUNDS} rounds × {SUB_DURATION}s each, one UDP guess per round.\n"
79      )
80      print(f"[Server] Secret number is {secret}")
81
82      udp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
83      udp_sock.bind((SERVER_HOST, UDP_PORT))
84
85      winner = None
86
```

Figure 55: Game Initialization.

Main Gameplay Loop (Rounds):

The game proceeds for a maximum number of subrounds (SUBROUNDS). Each round starts with a broadcast to inform clients about the current round number. The server records the round's start time and waits for player guesses via UDP within the specified duration (SUB_DURATION). It keeps track of which users have already guessed in the current round using a set to ensure each player gets only one guess per round.

```
    # play up to SUBROUNDS
    for rnd in range(1, SUBROUNDS+1):
        if not clients or winner:
            break

        broadcast_tcp(f"\n--- Round {rnd}/{SUBROUNDS} ---\n")
        round_start = time.time()
        guessed = set()

        while time.time() - round_start < SUB_DURATION and not winner:
            udp_sock.settimeout((round_start + SUB_DURATION) - time.time())
            try:
                data, addr = udp_sock.recvfrom(1024)
            except socket.timeout:
                break
            except ConnectionResetError:
                # client closed its UDP socket; ignore and keep waiting
                continue

            msg = data.decode().strip()
            user = next((u for u,a in udp_addrs.items() if a == addr), None)
            if not user or user in guessed:
                continue
            guessed.add(user)
```

Figure 56: Main Gameplay Loop.

Handling Player Messages and Exits:

Upon receiving a message over UDP, the server identifies the sender by matching their UDP address with registered users. If a user sends "exit", they are removed from the game, and their departure is broadcast to all others. If only one player remains, that player is given the choice to continue or end the game. If the user responds with "no" or disconnects, the game ends early. Otherwise, the loop continues.

```
            if msg.lower() == 'exit':
                udp_sock.sendto(b"You exited mid-game.\n", addr)
                with lock:
                    clients.pop(user, None)
                    udp_addrs.pop(user, None)
                broadcast_tcp(f"{user} left mid-game.\n")

                # If exactly one player remains, give them the choice:
                if len(clients) == 1:
                    lone = next(iter(clients))
                    lone_conn = clients[lone]
                    lone_conn.sendall(b"You're alone now. Continue? (yes/no)\n")
                    try:
                        ans = lone_conn.recv(1024).decode().strip().lower()
                    except:
                        ans = 'no'

                    if ans == 'yes':
                        broadcast_tcp(f"{lone} chose to continue alone.\n")
                        # break out of this round so we jump to the next for-loop iteration
                        break
                    else:
                        broadcast_tcp(f"{lone} chose to end the game.\n")
                        udp_sock.close()
                        return False

                # otherwise (0 or >1 remain), just continue this round
                continue
```

Figure 57: Handling Player Messages and Exits

Processing Guesses:

If a player submits a guess, the server attempts to parse it as an integer. If it's invalid or out of range, an error message is sent back, and the guess is ignored. If the guess is valid but incorrect, the server responds with hints like "Try HIGHER" or "Try LOWER." If the guess is correct, the server announces the winner and exits the gameplay loop early.

```
        # try to parse an integer guess
        try:
            val = int(msg)
        except ValueError:
            udp_sock.sendto(
                b"Invalid guess. Enter a number between 1 and 100. Your chance this round is over.\n",
                addr
            )
        if val < RANGE_LOW or val > RANGE_HIGH:
            udp_sock.sendto(
                b"Invalid guess. Enter a number between 1 and 100. Your chance this round is over.\n",
                addr
            )
            continue
        if val < secret:
            udp_sock.sendto(b"Try HIGHER\n", addr)
        elif val > secret:
            udp_sock.sendto(b"Try LOWER\n", addr)
        else:
            udp_sock.sendto(b"CORRECT!\n", addr)
            winner = user
            break
```

Figure 58: Processing Guesses.

Announcing Results:

After either a correct guess or all subrounds are exhausted, the server closes the UDP socket. It then informs all players of the game outcome via TCP, either announcing the winner or stating that no one guessed the number correctly.

```
    udp_sock.close()

    # announce result
    if winner:
        print(f"[Server] Game over. Winner: {winner} (secret was {secret})")
        broadcast_tcp(f"\n🎉 {winner} guessed it! Secret was {secret}.\n")
    else:
        print(f"[Server] Game over. No winner (secret was {secret})")
        broadcast_tcp(f"\nNo one guessed it. Secret was {secret}.\n")

    # prompt for next game
    broadcast_tcp("Start a new round? (yes/no)\n")
```

Figure 59: Announcing Results.

Prompting for a New Game:

Finally, the server asks all connected players whether they'd like to start another game. Each player's response is collected. If at least one player responds with "yes", a new round begins; otherwise, the game session ends with a farewell message.

```
173
174     # prompt for next game
175     broadcast_tcp("Start a new round? (yes/no)\n")
176
177     # collect yes/no from each remaining player
178     responses = {}
179     with lock:
180         survivors = list(clients.items())
181     for name, conn in survivors:
182         try:
183             ans = conn.recv(1024).decode().strip().lower()
184         except:
185             ans = 'no'
186         responses[name] = ans
187
188     # if any say "yes", restart
189     if any(v=='yes' for v in responses.values()):
190         broadcast_tcp("\n▶️ Starting next round...\n")
191         return True
192     else:
193         broadcast_tcp("\n❌ No more rounds. Goodbye!\n")
194         return False
195
```

Figure 60: Prompting for a New Game.

3.2 Client Implementation and Code:

Initial segment of the client code:

This part of the client.py file imports the socket and threading libraries. The socket library is used to connect the client to the server using TCP and UDP, while threading helps run parts of the program at the same time, like sending guesses while receiving messages. It also sets the TCP and UDP ports to 6000 and 6001, which are used to talk to the server.

```python
# client.py
import socket
import threading

TCP_PORT = 6000
UDP_PORT = 6001
```

Figure 61: Initial segment of the client code.

The recv_tcp(sock) function:

It is responsible for continuously receiving TCP messages from the server and displaying them to the user. These messages are typically control or informational messages such as game announcements, instructions, or results. The function runs in a loop, checking for incoming data and decoding it to be printed. If the connection closes or an error occurs, the loop breaks, ending the function.

The recv_udp(sock) function:

It handles incoming UDP messages, which are used for real-time gameplay updates, such as hints in response to guesses (e.g., "Try HIGHER" or "CORRECT!"). It operates similarly to the TCP receiver but uses the recvfrom method to get both the message and the sender's address. It decodes and prints messages until an error occurs or the socket is closed.

The stdin_loop(tcp_sock, udp_sock, server_ip) function:

It handles all user inputs from the terminal. It reads each input line, trims it, and sends it to the appropriate socket based on its content. If the input is "yes" or "no", it's sent via TCP to answer control prompts (like whether to continue playing). Any other input is considered a game guess and sent via UDP to the server. If the user types "exit", it sends that message via UDP and exits the loop, signaling the client is leaving the game.

```
173
174     # prompt for next game
175     broadcast_tcp("Start a new round? (yes/no)\n")
176
177     # collect yes/no from each remaining player
178     responses = {}
179     with lock:
180         survivors = list(clients.items())
181     for name, conn in survivors:
182         try:
183             ans = conn.recv(1024).decode().strip().lower()
184         except:
185             ans = 'no'
186         responses[name] = ans
187
188     # if any say "yes", restart
189     if any(v=='yes' for v in responses.values()):
190         broadcast_tcp("\n▶️ Starting next round...\n")
191         return True
192     else:
193         broadcast_tcp("\n❌ No more rounds. Goodbye!\n")
194         return False
195
```

Figure 62: Client functions.

3.3 Results & Testing:

Once the minimum player count is reached, the server waits for up to 30 seconds to allow a ddditional players to join before beginning the game.

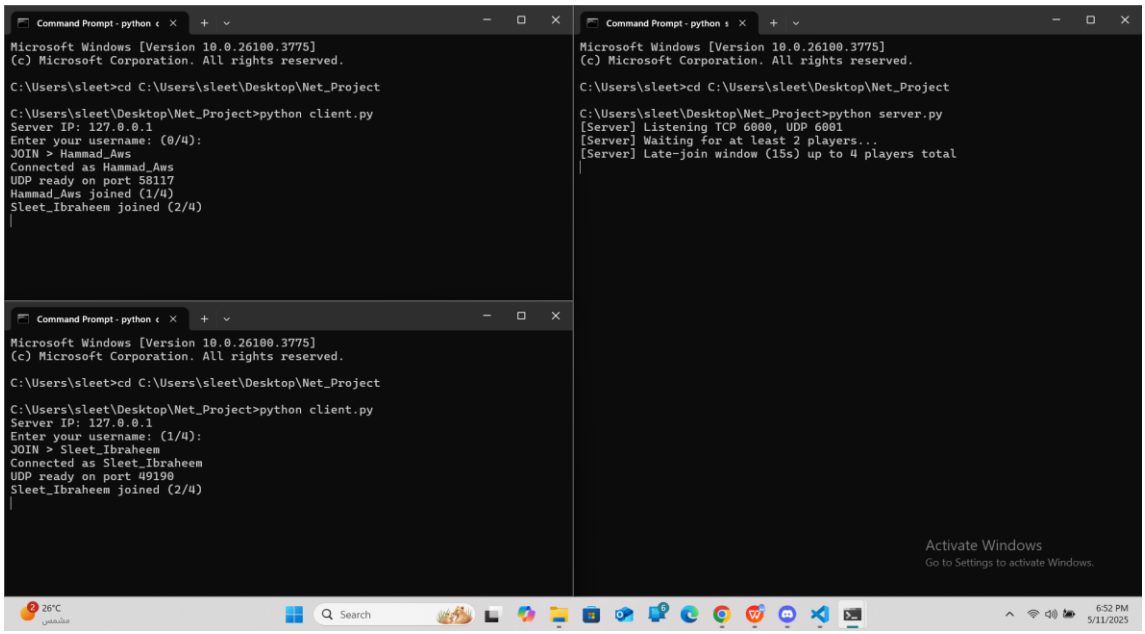

Figure 63: Start a game with 2 players

The game begins with the rules displayed to all players. Each client then submits a guess and receives immediate feedback indicating whether they should guess higher, lower, or if they guessed correctly.

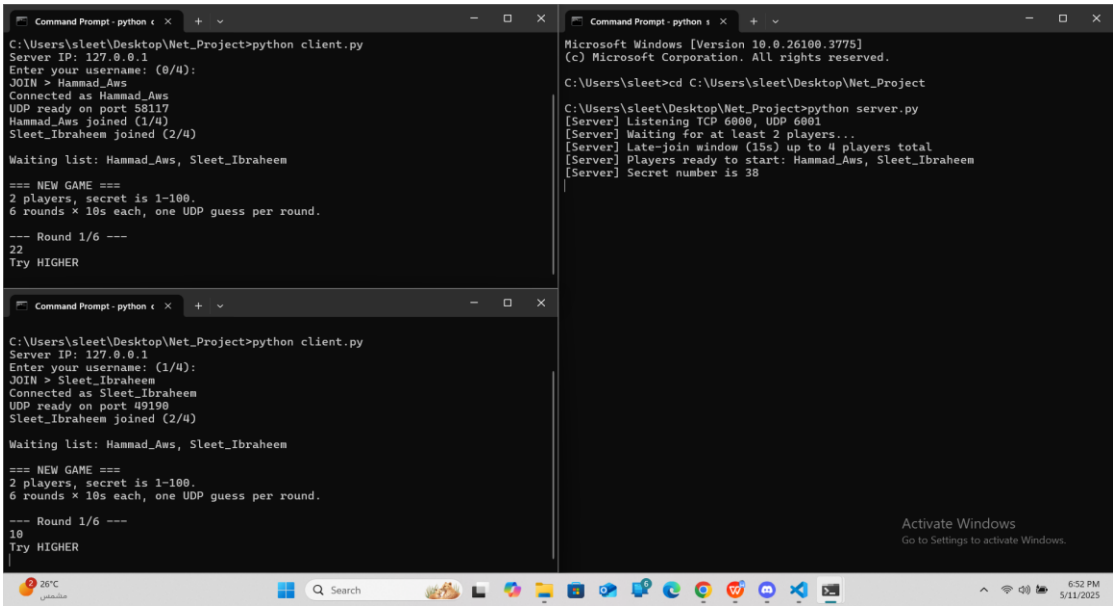

Figure 64: Enter a guess and receive feedback

When a player correctly guesses the secret number, the game ends immediately. The server notifies all connected players that the game is over, announces the winner, and then asks each player whether they want to start a new round.

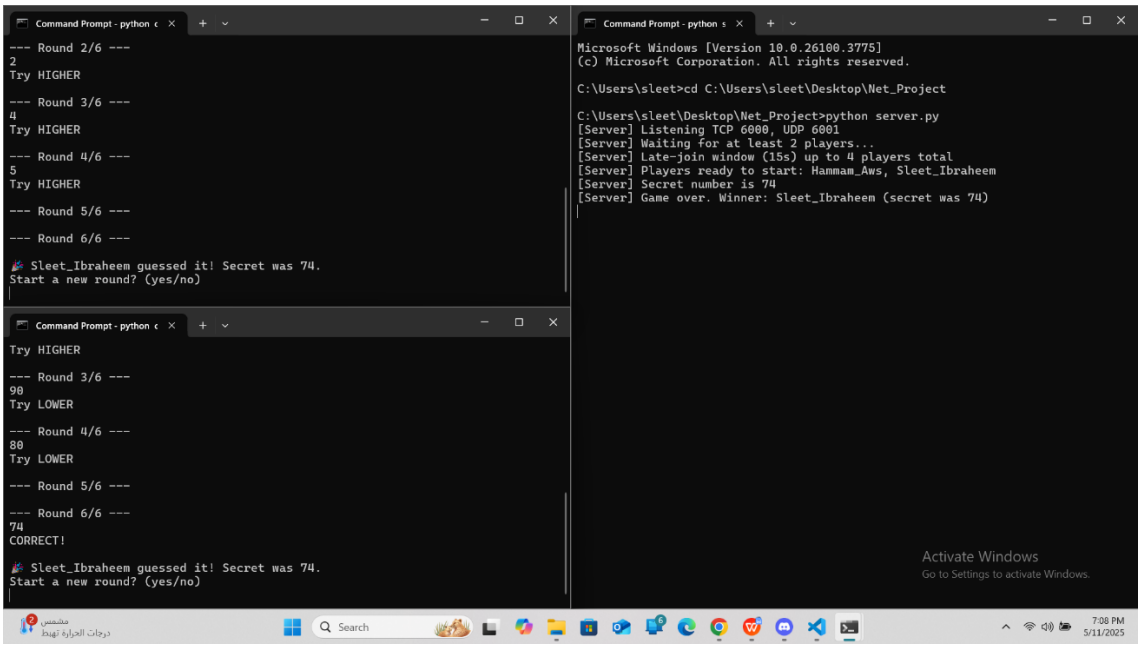

Figure 65: Ending a round and start a new one.

When a player enters "yes," they wait for other players to join the game. If at least two players respond with "yes," a 30-second waiting period begins to allow additional players to join. After the wait, a new round starts.

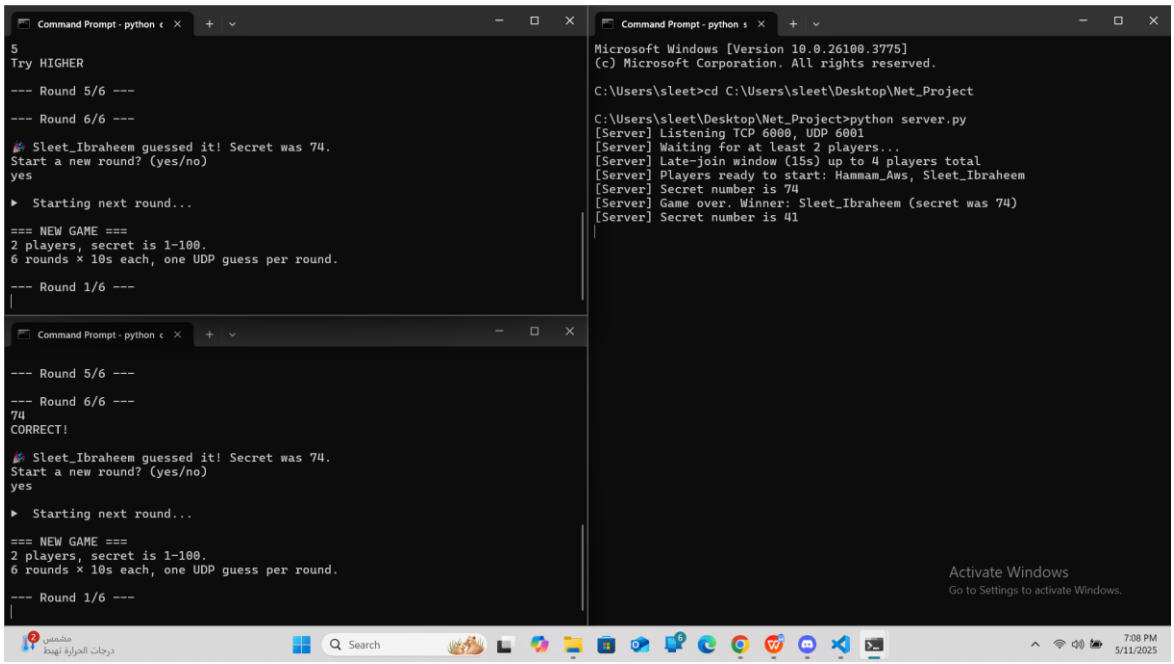

Figure 66: Starting a new round.

When a player enters a guess outside the valid range, an error message is displayed, and they lose their chance for that round.

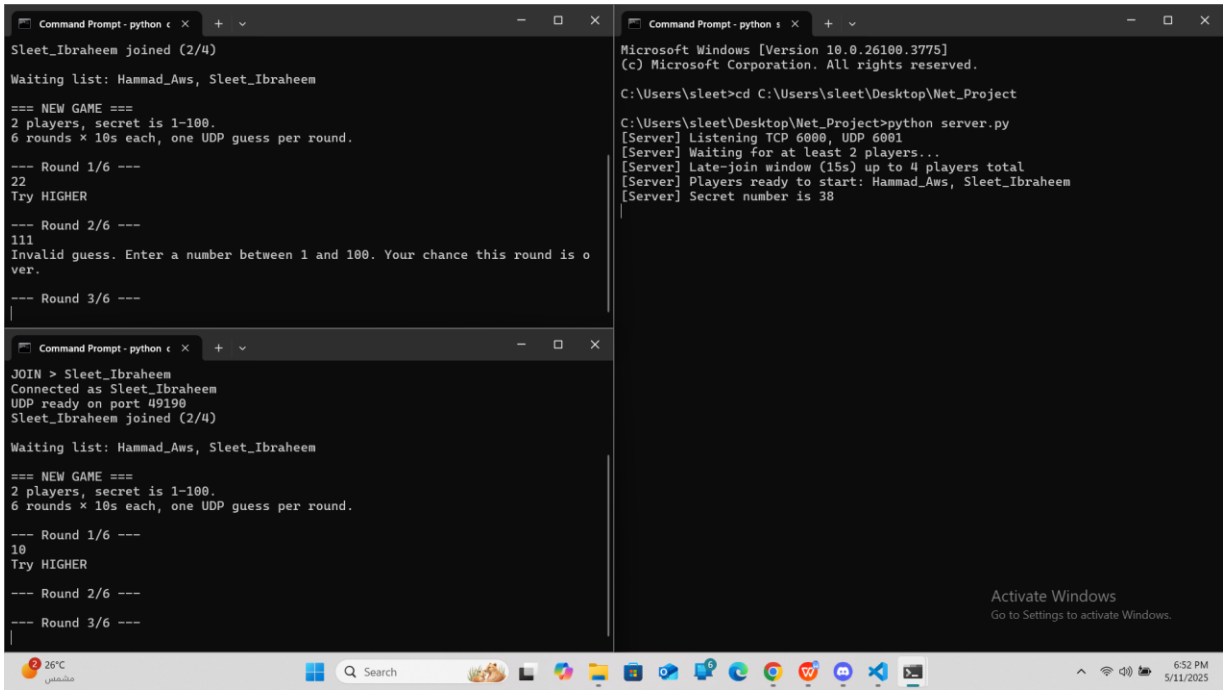

Figure 67: Out of the range case.

If no one guesses the number, there is no winner, and players are asked if they want to play again.

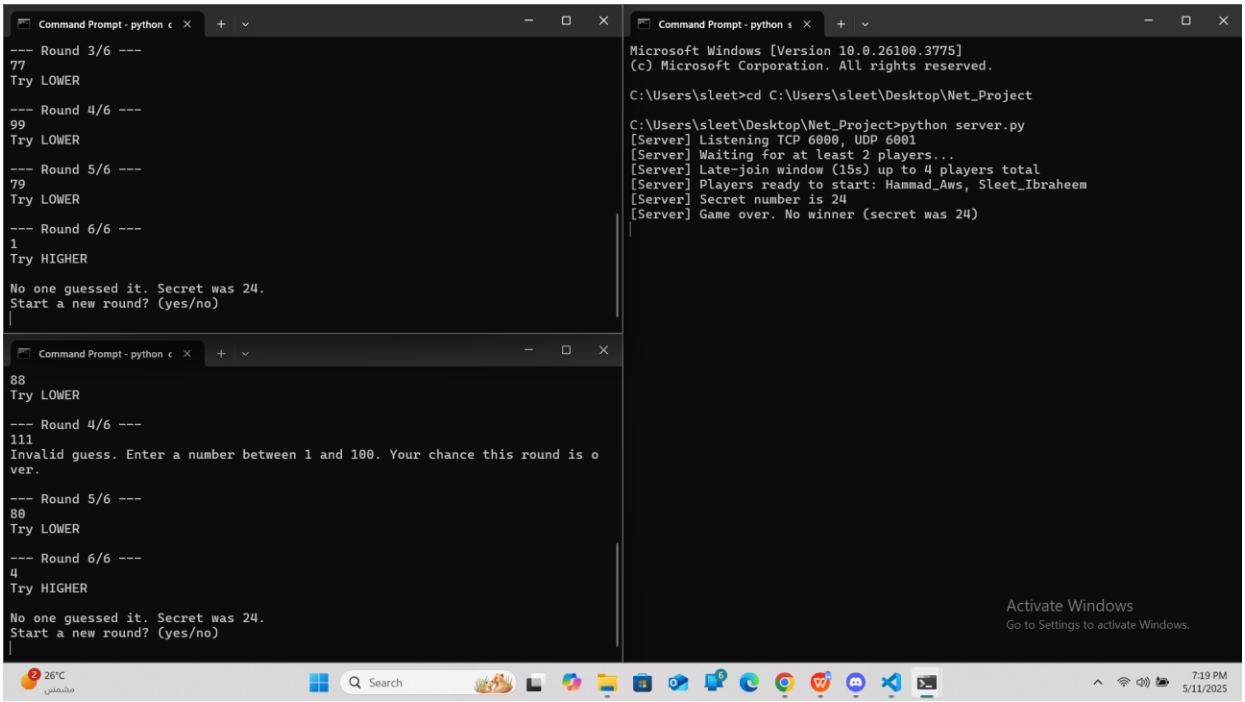

Figure 68: no players guesses the correct number

When one player disconnects by entering 'EXIT,' we ask the other player if they want to continue or restart the game.

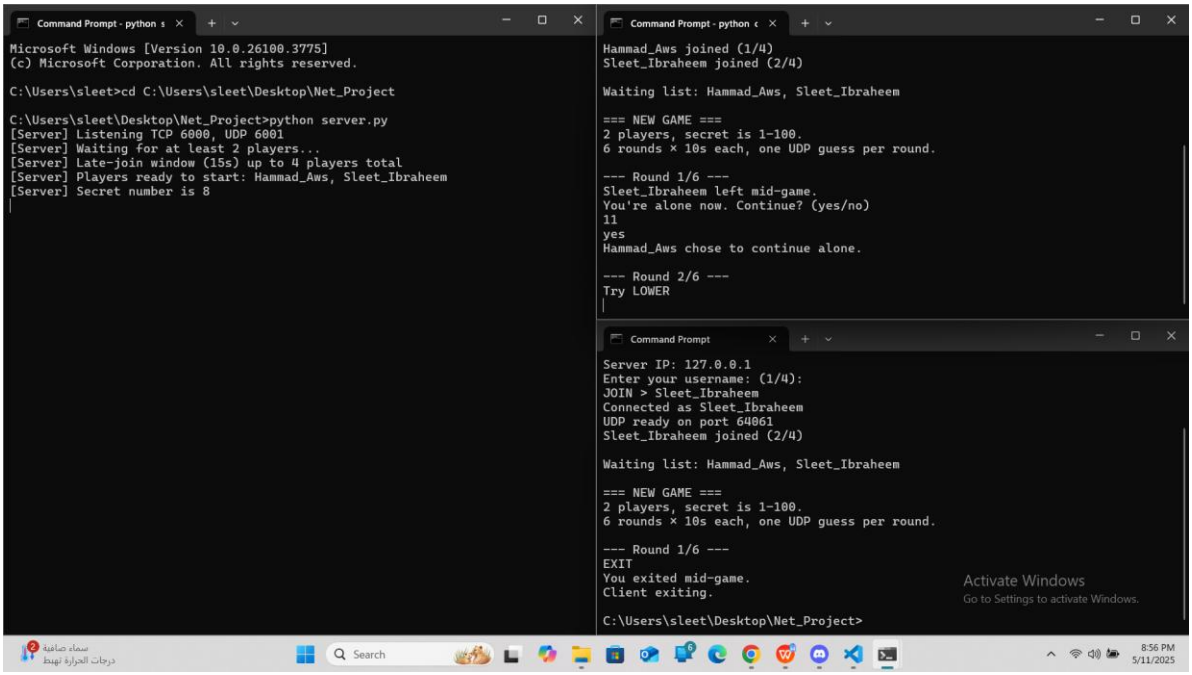

Figure 69: Disconnected player.

When two players connect, they wait for 30 seconds to start the game. If a third player joins during that time, the game waits another 30 seconds to allow a fourth player to join before starting.

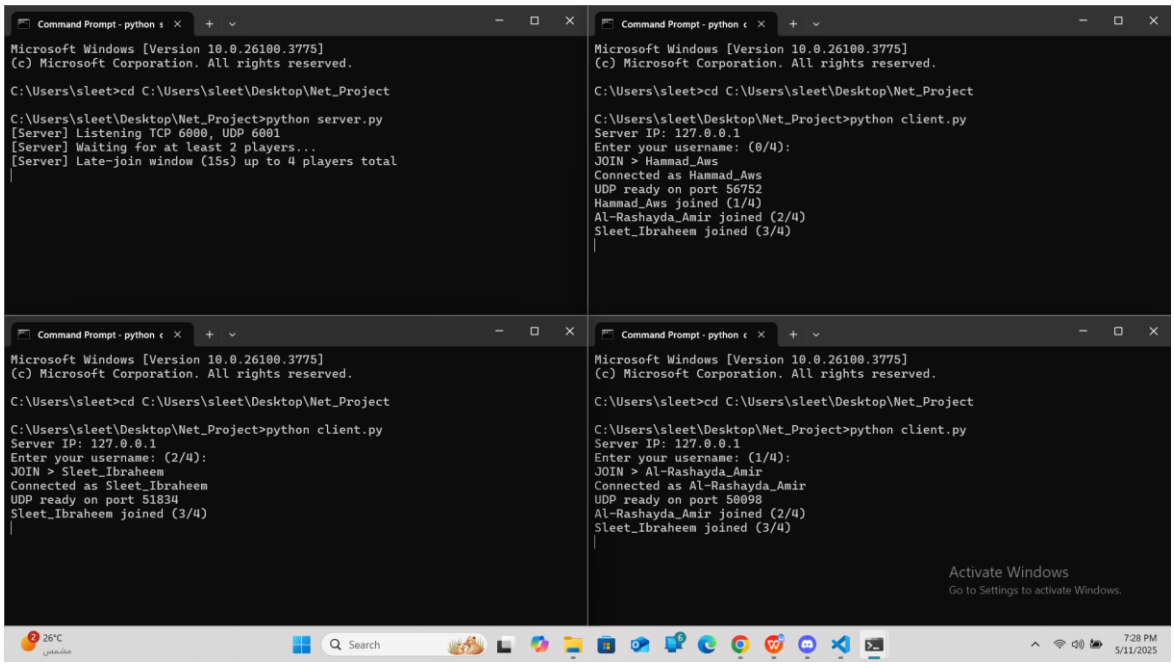

Figure 70: Three players waiting

Starting the game for three players.

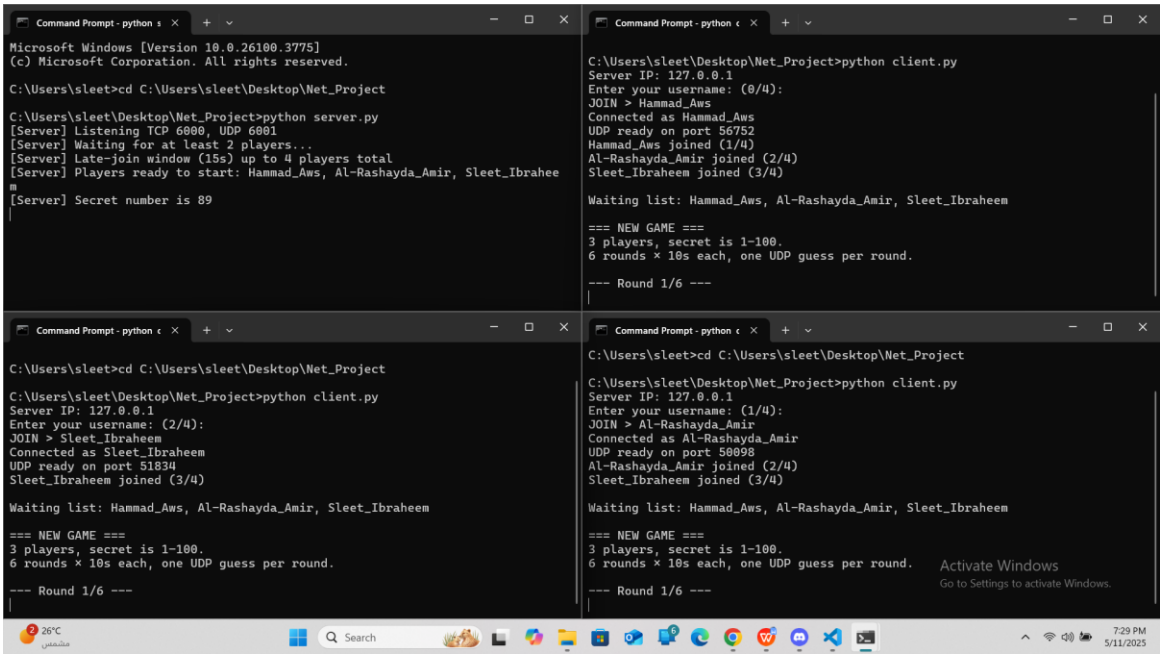

Figure 71: Starting the game for three players

Teamwork

Throughout this project, we made sure that all team members participated equally in all the tasks. We worked together on coding, testing, preparing the website content, building the server, and writing the reports. We didn't assign specific parts to specific people, but rather, we collaborated on everything as one team to make sure everyone was involved and learning from every step.

We regularly met at the university to discuss our progress, test our implementations, and solve any challenges we faced. When we couldn't meet physically, we stayed connected through online meetings and messages to ensure that we stayed on track and worked efficiently as a team. This approach helped us build a better understanding of the project and allowed each member to contribute equally to the success of our work.

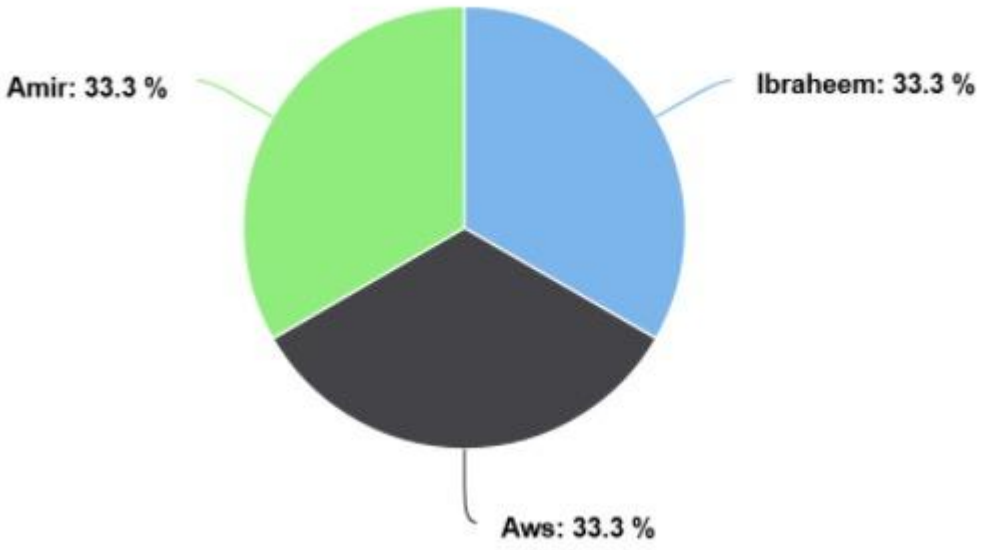

Figure 72: Teamwork Chart

Conclusion

In this project, we successfully explored multiple core concepts of computer networking through practical implementation and experimentation.

In Task 1, we applied diagnostic tools such as ipconfig, ping, tracert, nslookup, and telnet to analyze local and external network connectivity, verify DNS resolution, and understand routing paths. Additionally, we captured live network traffic using Wireshark to observe the DNS protocol in action, gaining practical insights into how applications translate domain names into IP addresses.

In Task 2, we developed a fully functional static web server using Python socket programming. This server was capable of handling HTTP GET requests and serving HTML, CSS, and image files to a browser, simulating how real web servers operate. We also created multilingual web pages, supporting both Arabic and English, and designed a professional user interface.

In Task 3, we designed and implemented a multiplayer number guessing game using both TCP and UDP socket communication. The server managed player sessions, game rounds, and timing constraints, while the client provided an interactive experience for players to submit their guesses and receive real-time feedback. This task demonstrated our ability to apply socket programming concepts to build a stateful networked application.

Overall, this project allowed us to translate theoretical networking knowledge into hands-on experience, reinforcing our understanding of client-server architecture, network protocols, and real-world debugging tools. Through teamwork and iterative development, we achieved all required objectives and delivered a comprehensive networking solution that combines diagnostics, web serving, and real-time multiplayer communication.

References

[1] https://www.wireshark.org/download.html **[Accessed 10-5-2025]**

[2] https://bgp.tools/as/1249 **[Accessed 10-5-2025]**

[3] https://bgp.tools/as/1968 **[Accessed 10-5-2025]**

[4] https://gaia.cs.umass.edu/ **[Accessed 10-5-2025]**

[5] https://notebooklm.google/ **[Accessed 11-5-2025]**

[6] https://www.fing.com/ **[Accessed 11-5-2025]**