Faculty of Engineering & Technology

Electrical & Computer Engineering Department

ENCS3320, Computer Networks

Second Semester (2025/2026)

Project 1 – Socket Programming

Team Members

Faris Sawalmeh	**1220013**	**Section: 2**
Mohammad Ewais	**1220053**	**Section: 5**
Yahia Sarhan	**1221858**	**Section: 2**

Instructor: Dr. Alhareth Zyoud & Ibrahim Nemer

Submission Date: May, 8, 2025

Abstract

This report presents the design and implementation of a multi-part project aimed at deepening practical knowledge in computer networks and socket programming.

The first task focuses on using essential networking tools such as ipconfig, ping, tracert, nslookup, and telnet to analyze IP routing, DNS resolution, and HTTP-level connectivity, while also investigating the autonomous system (AS) information for a public domain.

The second task involves building a custom TCP web server using socket programming, capable of handling basic HTML requests, redirection, and multilingual content display.

The third task extends socket programming to create a UDP-based multiplayer game, where the server handles multiple clients, sends trivia questions, and updates scores in real-time.

Overall, the project combines theoretical concepts with hands-on practice, offering experience in diagnostics, server-side development, communication protocols, and real-time data processing.

Table of Contents

Abstract 2

Table of Figures 4

List of Tables 6

Theory & Procedure 7

- ❖ **Socket Programming (TCP and UDP)** 7
  - • **Socket Programming with UDP** 7
  - • **Socket Programming with TCP** 8
- ❖ **Network Commands** 11
- ❖ **Wireshark** 12
- ❖ **Web Server** 13

Results and Discussion 15

1) **Task 1 – Network Commands and Wireshark** 15

Network Commands 15

Using Wireshark to capture some DNS messages 23

2) **Task 2 – Web Server** 25

- ❖ **Webpages** 26
- ❖ **HTML Code** 35
- ❖ **CSS Code** 38

..... 39

- ❖ **Web Server Implementation & Code** 45
- ❖ **Requests and Response Testing** 57

3) **Task 3 – Web Server** 64

- ❖ **Introduction for this task** 64
- ❖ **Server Implementation & Code** 65
- ❖ **Client Implementation & Code** 73
- ❖ **Results & Testing** 79

Alternative Solutions, Issues, and Limitations 89

Teamwork 90

References 91

Table of Figures

Figure 1: socket programming diagram. 7
Figure 2: User Datagram Protocol. 7
Figure 3: UDP socket programming. 8
Figure 4: Transmission Control Protocol. 8
Figure 5: TCP socket programming. 9
Figure 6: Request between webserver and browser 13
Figure 7: ipconfig/all command 15
Figure 8: Ping to local gateway (192.168.1.1) 16
Figure 9: Ping to another local device (192.168.1.13) 17
Figure 10: Ping to gaia.cs.umass.edu 18
Figure 11: Traceroute to gaia.cs.umass.edu 19
Figure 12: nslookup query for gaia.cs.umass.edu 20
Figure 13: Telnet connection to gaia.cs.umass.edu on port 80 21
Figure 14: Autonomous System (AS) details of University of Massachusetts 22
Figure 15: DNS query for gaia.cs.umass.edu 23
Figure 16: DNS response for gaia.cs.umass.edu showing IP address 128.119.245.12 24
Figure 17: Main English Webpage 26
Figure 18: Main English Webpage (CONT.) 26
Figure 19: Main English Webpage (CONT.) 27
Figure 20: Main English Webpage (CONT.) 27
Figure 21: Main English Webpage (CONT.) 28
Figure 22: Main English Webpage (CONT.) 28
Figure 23: Main English Webpage (CONT.) 29
Figure 24: Searching Material English Webpage 30
Figure 25: Main Arabic Webpage 30
Figure 26: Main Arabic Webpage (CONT.) 31
Figure 27: Main Arabic Webpage (CONT.) 31
Figure 28: Main Arabic Webpage (CONT.) 32
Figure 29: Main Arabic Webpage (CONT.) 32
Figure 30: Main Arabic Webpage (CONT.) 33
Figure 31: Main Arabic Webpage (CONT.) 33
Figure 32: Search Material Arabic Webpage 34
Figure 33: Illustration of client-server communication using a socket connection. 46
Figure 34: Localhost Media Search Interface 57
Figure 35 : Downloaded File Displayed in Safari's Downloads Panel 57

Figure 36: Server output showing players joining and the winner announcement. 79
Figure 37: Client-side game interaction showing player guesses and final winner announcement. 79
Figure 38: Game session showing cooldown enforcement and final game result with the winner. 80
Figure 39: Player session showing error handling for cooldown violation and invalid input range. 81
Figure 40: Player left; game continued solo to win. 84
Figure 41: Early guesses, then player quit. 84
Figure 42: First scenario when Faris select to continue 85
Figure 43: Second scenario when Faris select to stop the game 86
Figure 44: Game ended due to time out. 87
Figure 45: Time ran out with no winner. 87
Figure 46: Game started but ended with no winners. 88
Figure 47: Teamwork Chart 90

List of Tables

Table 1: Differences between TCP & UDP 10

Theory & Procedure

❖ Socket Programming (TCP and UDP)

A typical **network application** is made up of two distinct **programs**: a **client program** and a **server program**, each running on separate **end systems**. When these programs are launched, they create a **client process** and a **server process**, which interact by **reading from** and **writing to sockets**. The primary responsibility of the **developer** when building such an application is to implement the **code** for both the **client-side** and **server-side** components.

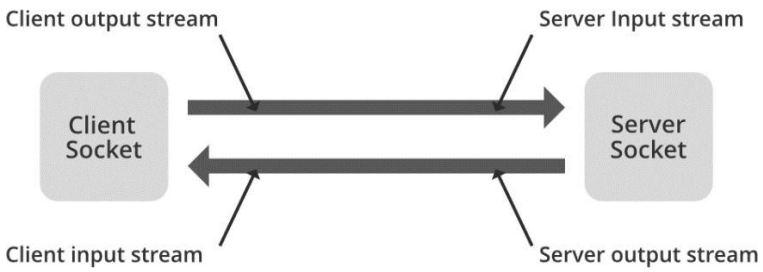


Figure 1: socket programming diagram.

• Socket Programming with UDP

The **User Datagram Protocol (UDP)** operates on top of the **Internet Protocol (IP)** and is designed for **applications** that do not require **reliability**, **acknowledgments**, or **flow control** at the **transport layer**. As a **lightweight protocol**, UDP offers **transport layer addressing** through **UDP ports** and includes an optional **checksum** feature for error detection.

UDP works by sending **datagrams**—individual **messages**—directly to other hosts on an **IP network**, without the need to establish dedicated **connections** or **transmission paths** in advance. To communicate, a **UDP socket** simply needs to be **opened**, allowing it to **listen** for incoming data and **send** messages when requested.

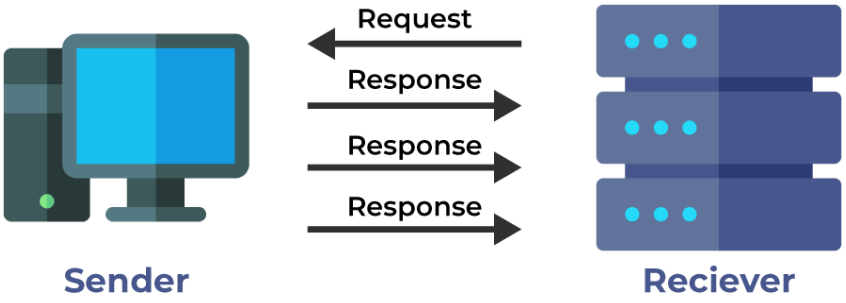


Figure 2: User Datagram Protocol.

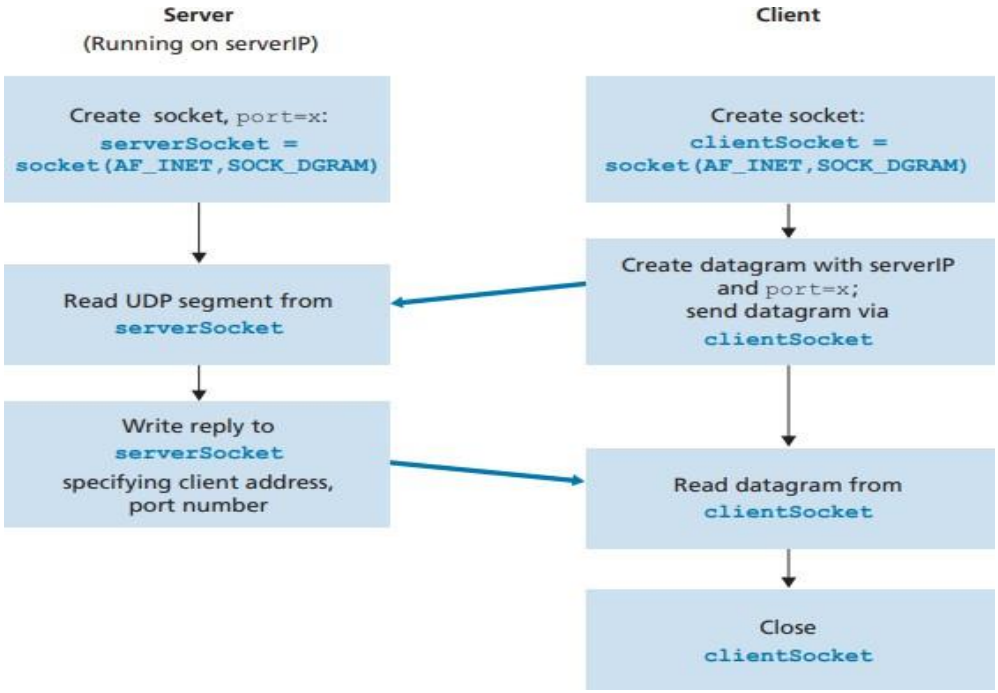


Figure 3: UDP socket programming.

- **Socket Programming with TCP**

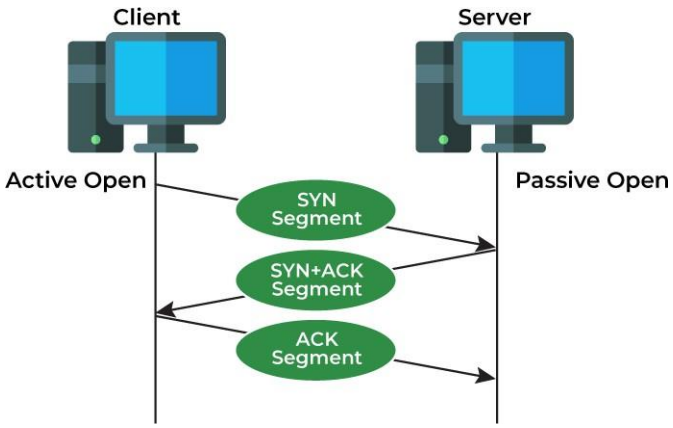


Figure 4: Transmission Control Protocol.

In contrast to **UDP**, **TCP** is a **connection-oriented protocol**. This means that before any **data exchange** can occur, the **client** and **server** must first perform a **handshake** to establish a **TCP connection**. One end of this connection is linked to the **client socket**, while the other is connected to the **server socket**. During the setup of this **TCP connection**, both the **client's** and **server's socket addresses**—consisting of **IP addresses** and **port numbers**—are associated with the connection.

Once the connection is in place, either side can simply **send data** by placing it into the **TCP stream** through its **socket**, without specifying the destination each time. This differs from **UDP**, where the **server** must explicitly assign a **destination address** to each **packet** before sending it through the **socket**.

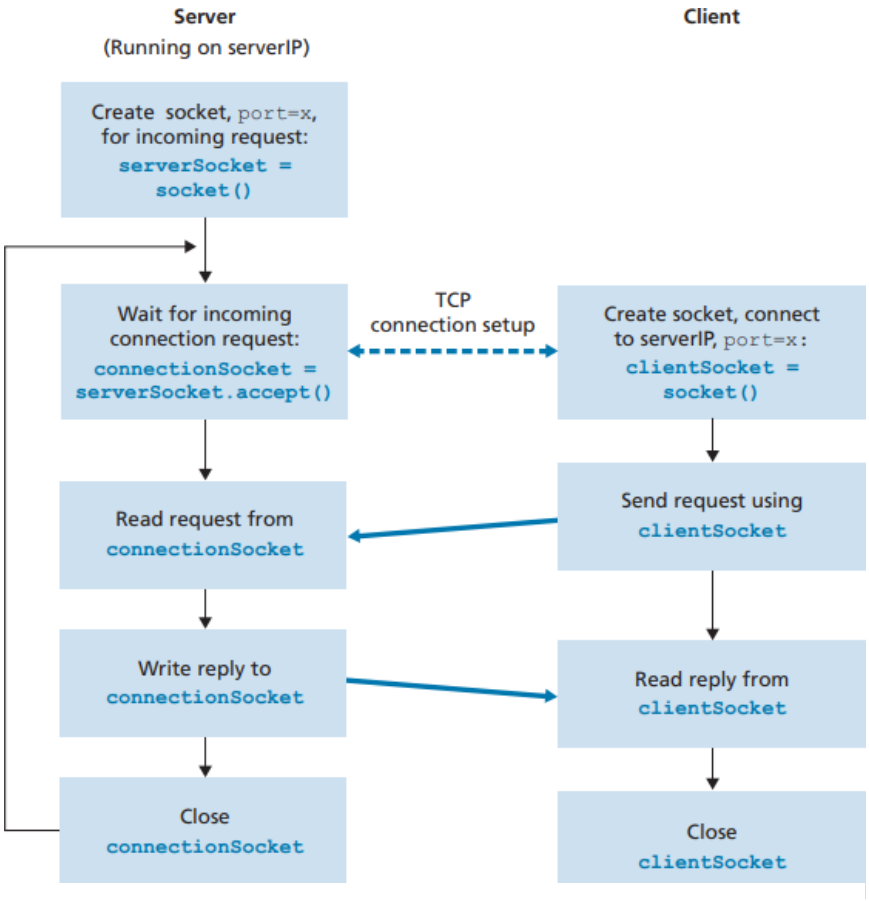


Figure 5: TCP socket programming.

In conclusion, **TCP (Transmission Control Protocol)** is a **connection-oriented** protocol that guarantees **reliable** and **ordered delivery** of data through mechanisms like **error checking**, **acknowledgments**, and **retransmissions**. Although this makes it **slower**, it is ideal for **dependable applications** such as **web browsing** and **email**. On the other hand, **UDP (User Datagram Protocol)** is **connectionless** and emphasizes **speed** with **minimal overhead**, making it better suited for **real-time applications** like **video streaming** and **online gaming**. Unlike **TCP**, **UDP** does **not guarantee** delivery, packet **order**, or perform **congestion control**.

Table 1: Differences between TCP & UDP

TCP	UDP
Keeps track of lost packets. Makes sure that lost packets are re-sent	Doesn't keep track of lost packets
Adds sequence numbers to packets and reorders any packets that arrive in the wrong order	Doesn't care about packet arrival order
Slower, because of all added additional functionality	Faster, because it lacks any extra features
Requires more computer resources, because the OS needs to keep track of ongoing communication sessions and manage them on a much deeper level	Requires less computer resources
Examples of programs and services that use TCP: - HTTP - HTTPS - FTP - Many computer games	Examples of programs and services that use UDP: - DNS - IP telephony - DHCP - Many computer games

❖ Network Commands

Network commands are essential **tools** in computing that enable users to **manage** and **interact** with various **network configurations** and **connections**. They are commonly used to **diagnose issues**, **test connectivity**, retrieve **network device information**, and **analyze data traffic**. By running these **commands**, users can perform tasks such as viewing **IP settings**, tracing **data routes**, resolving **domain names**, and controlling **network protocols**. These tools are particularly useful for **network administrators** and anyone involved in **troubleshooting** or improving **network performance**.
Here are the specific **commands** we will be using in this **project**:

1. ipconfig

This **networking command** is used to display **IP configuration details**. It provides essential information such as the **IPv4 address**, **Subnet Mask**, and **Default Gateway**.

1. **Subnet Mask** – This represents the **boundary** of the **local network**, helping to determine which part of an IP address refers to the **network** and which part refers to the **host**.
2. **Default Gateway** – This is the **IP address** of the **router** that your computer contacts **first** when trying to reach a device **outside** the local network.

2. ipconfig/all

This **command** can be seen as an **enhanced version** of the **ipconfig** command. It provides the **physical (MAC) address** of your **device** along with detailed **network information**. It displays values like **IPv4**, **IPv6**, **Default Gateway**, **Subnet Mask**, and also shows the **devices** your system is currently **connected to**, including their **configuration details**.
Additionally, it gives information about **DHCP (Dynamic Host Configuration Protocol)**—a protocol mainly used to **automatically assign IP addresses** to connected **devices**, simplifying **network management**.

3. ping

The **ping command** is used to determine whether a specific **website** or **host** is **reachable**. It works by sending **data packets** to the target **IP address** and waits to see if a **response** is received within a certain **time frame**. If the response arrives, it confirms that the **destination** is accessible. To use it, simply type **ping** followed by the **IP address** or **domain name** of the site you want to test.

4. **Tracert**

This **command**, known as **traceroute** (or **tracert** on some systems), is used to track the **path** that data takes from your **computer** to a specific **destination**. It reveals the sequence of **servers** or **network devices**—also called **hops**—that your data passes through to reach the **target location**. This helps in identifying where **delays** or **failures** occur along the **route**.

5. **Nslookup**

This **command** is used to **translate** a given **domain name** or **search term** into its corresponding **IP address**. It helps in resolving **hostnames** to numerical **network addresses**, which are necessary for locating and connecting to **web servers** on the **Internet**.

6. **Telnet**

This command enables users to **connect remotely** to other **devices** using the **Telnet protocol**. It is commonly used for **remote command-line access** and the **administration** of **servers** and **network equipment**.

❖ Wireshark

Wireshark is a powerful, **open-source network protocol analyzer** that is widely used for **network troubleshooting**, **monitoring**, and **security analysis**. It allows users to **capture**, **inspect**, and **analyze network traffic** either **in real-time** or from **saved capture files**, making it an essential tool for **IT professionals** and **cybersecurity specialists**.
With its **intuitive interface**, **Wireshark** supports a vast range of **network protocols** and provides features like **advanced filtering**, **packet marking**, and **statistical tools** to help identify problems such as **latency**, **packet loss**, or **bandwidth abuse**.

Common **use cases** include spotting **network slowdowns**, evaluating **VoIP call quality**, examining **protocol behavior**, and detecting **security threats** like **malware communications**. However, using Wireshark effectively requires a strong grasp of **network protocols**, as it can only display what it captures and **cannot decrypt encrypted traffic** unless the correct **decryption keys** are supplied.
It's also important to consider **legal** and **ethical** factors—**unauthorized packet capture** may **violate privacy laws** and **regulations**.

❖ Web Server

A **web server** is a **software application** that handles **network requests** from users and delivers the necessary **files** to generate **web pages**. This interaction occurs using the **Hypertext Transfer Protocol (HTTP)**.
In essence, a **web server** is a **computer** that stores **HTTP-based content** (like HTML, images, and media files) that make up a **website**. When a **client** (like your laptop) requests a website—say, by typing a URL like Facebook into a browser—the computer sends an **HTTP request** to the **web server** hosting Facebook. The **web server**, in turn, responds by sending the **website files** back to the client so the webpage can be displayed.

Multiple **websites** can be hosted on a single or multiple **web servers**, and this does not affect how the website appears to the **user**. A **web server** may be implemented as **software** or **hardware**, but most often it is **software** running on a **computer**. These servers are capable of handling **multiple simultaneous users**, which is crucial—otherwise, we'd need a separate **server** for each user, which is unrealistic on a global scale.

Importantly, a **web server** must stay **connected to the internet** at all times. If it becomes **disconnected**, it cannot receive or respond to incoming **requests**, rendering it **inaccessible** to users.

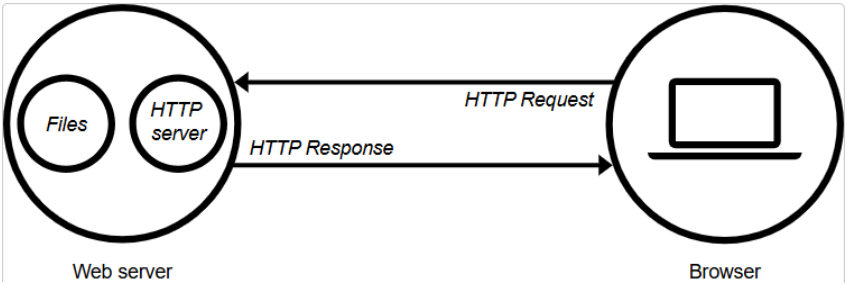


Figure 6: Request between webserver and browser

How Web Servers Work:

1. **Client Request**: The process starts when a **client** (typically a **web browser**) sends an **HTTP request** to a **web server**, asking for a specific **resource** (like **/index.html**).
2. **Server Processing**: The **server** then processes the request by:
    - Finding the **requested file** in its **storage**.
    - Running any necessary **server-side code** if the request is for **dynamic content** (such as **PHP**, **Python**, or other **scripts**).
3. **Server Response**: The **web server** responds by sending an **HTTP response** to the **client**, which includes:
    - A **Status Code** that tells whether the request was successful (e.g., **200 OK**, **404 Not Found**).
    - **Headers** that provide **metadata** (such as **Content-Type: text/html**).
    - The **Body**, which contains the actual **web content** (like **HTML**, **CSS**, or **JSON**).
4. **Client Rendering**: The **client** (browser) then processes the **response** and **displays** the **web page** to the **user**.

Results and Discussion

1) Task 1 – Network Commands and Wireshark

Network Commands

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

```
Wireless LAN adapter WiFi:

   Connection-specific DNS Suffix  . :
   Description . . . . . . . . . . . : Intel(R) Wi-Fi 6 AX201 160MHz
   Physical Address. . . . . . . . . : 60-E3-2B-D5-F8-95
   DHCP Enabled. . . . . . . . . . . : Yes
   Autoconfiguration Enabled . . . . : Yes
   Link-local IPv6 Address . . . . . : fe80::3b01:30c5:9950:a002%8(Preferred)
   IPv4 Address. . . . . . . . . . . : 192.168.1.10(Preferred)
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Lease Obtained. . . . . . . . . . : Friday, May 2, 2025 2:41:47 PM
   Lease Expires . . . . . . . . . . : Saturday, May 3, 2025 2:41:47 PM
   Default Gateway . . . . . . . . . : 192.168.1.1
   DHCP Server . . . . . . . . . . . : 192.168.1.1
   DHCPv6 IAID . . . . . . . . . . . : 140567339
   DHCPv6 Client DUID. . . . . . . . : 00-01-00-01-2A-A7-D7-EA-50-EB-F6-E6-60-58
   DNS Servers . . . . . . . . . . . : 192.168.1.1
   NetBIOS over Tcpip. . . . . . . . : Enabled
```

Figure 7: ipconfig/all command

This command displays the network configuration of the device.

- **IPv4 Address**: 192.168.1.10 (This is the local IP address assigned to the device by the router).
- **Subnet Mask**: 255.255.255.0 (Defines the network range for local communication).
- **Default Gateway**: 192.168.1.1 (The router's IP address, used to connect to external networks like the Internet).
- **DNS Server**: 192.168.1.1 (The router is configured to act as a DNS server, resolving domain names to IP addresses).

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

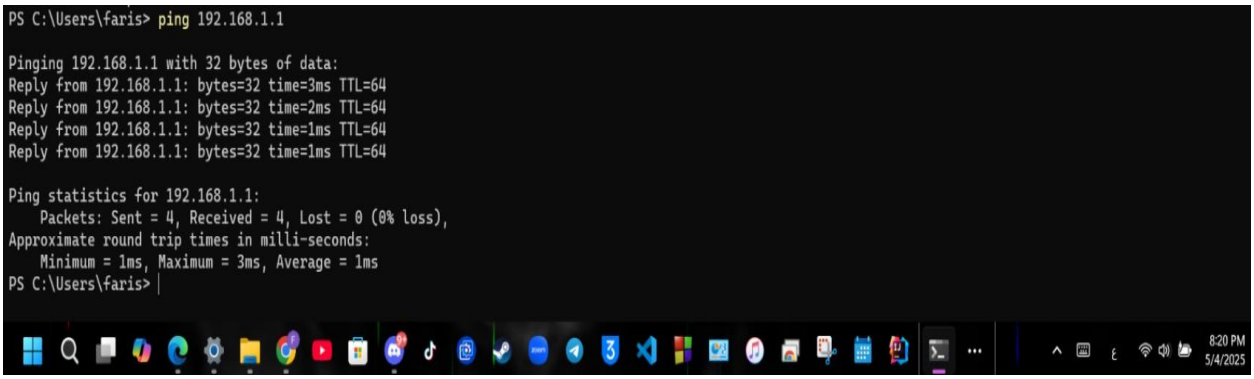


Figure 8: Ping to local gateway (192.168.1.1)

A ping command was executed to the local router IP address 192.168.1.1 to test network connectivity and response time.

Results:

- Packets Sent: 4
- Packets Received: 4
- Packet Loss: 0% (No loss)
- Round Trip Time:
  - Minimum: 1 ms
  - Maximum: 3 ms
  - Average: 1 ms
- TTL (Time To Live): 64

Interpretation:

The test confirms stable and fast communication with the local gateway.

All packets were delivered successfully with extremely low latency, indicating:

- Excellent local network performance
- No delay or congestion between the host device and the router
- The TTL value of 64 is typical for Linux/Unix-based routers and shows the packet didn't travel far (as expected in local network tests)

Ping Test Analysis – Local Device (192.168.1.13)

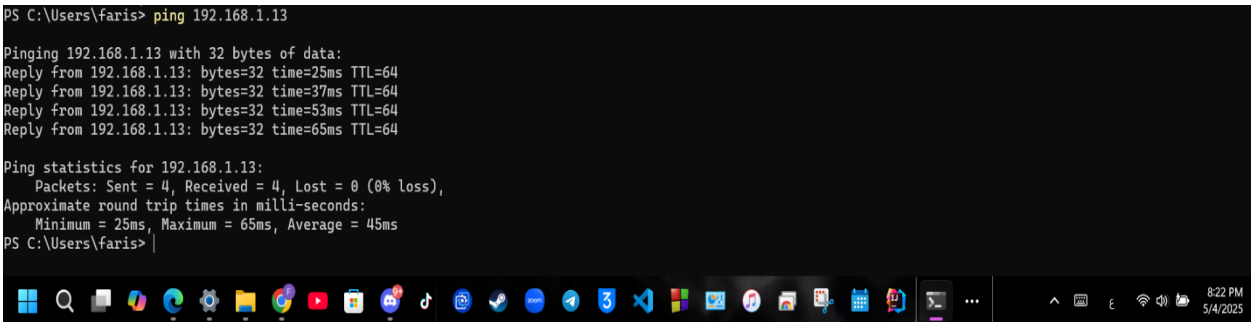


Figure 9: Ping to another local device (192.168.1.13)

A ping command was executed to a device on the same local network with IP address 192.168.1.13.

Results:

- Packets Sent: 4
- Packets Received: 4
- Packet Loss: 0% (No loss)
- Round Trip Time:
  - Minimum: 25 ms
  - Maximum: 65 ms
  - Average: 45 ms
- TTL (Time To Live): 64

Interpretation:

The device at 192.168.1.13 is reachable and active on the local network.

Although there is no packet loss, the round-trip time is **higher than typical for LAN communication**, which may indicate:

- Temporary background load on the target device
- Wi-Fi interference or lower signal quality
- Or minor congestion on the local network

Despite the slightly elevated latency, communication was successful and confirms functional connectivity between the two devices.

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

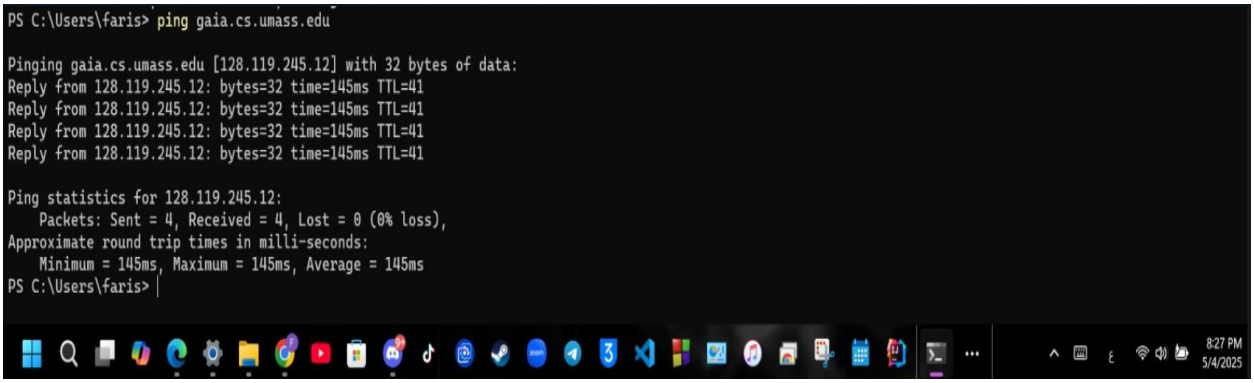


Figure 10: Ping to gaia.cs.umass.edu

A ping command was executed to the remote host gaia.cs.umass.edu (IP: 128.119.245.12) to assess end-to-end connectivity over the internet.

Results:

- Packets Sent: 4
- Packets Received: 4
- Packet Loss: 0% (No loss)
- Round Trip Time:
  - Minimum: 145 ms
  - Maximum: 145 ms
  - Average: 145 ms
- TTL (Time To Live): 41

Interpretation:

The server responded successfully to all requests, with no packet loss and consistent latency.

An average round trip time of **145 ms** is typical for international communication across multiple internet hops.

The TTL value of **41** suggests that the packet passed through multiple routers (hops) before reaching the destination, indicating the remote nature of the host.

This test confirms that gaia.cs.umass.edu is reachable and responsive over the internet.

The ping command to gaia.cs.umass.edu returned **four successful replies** from the IP address 128.119.245.12 with a **round-trip time of 145 ms** and **0% packet loss**.

This indicates that the host is reachable and responding normally to ICMP echo requests.

Since the replies consistently came from the IP address **128.119.245.12**, which matches the resolved address of gaia.cs.umass.edu, we can confidently say:

Yes, the response is most likely originating from the intended server gaia.cs.umass.edu.

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

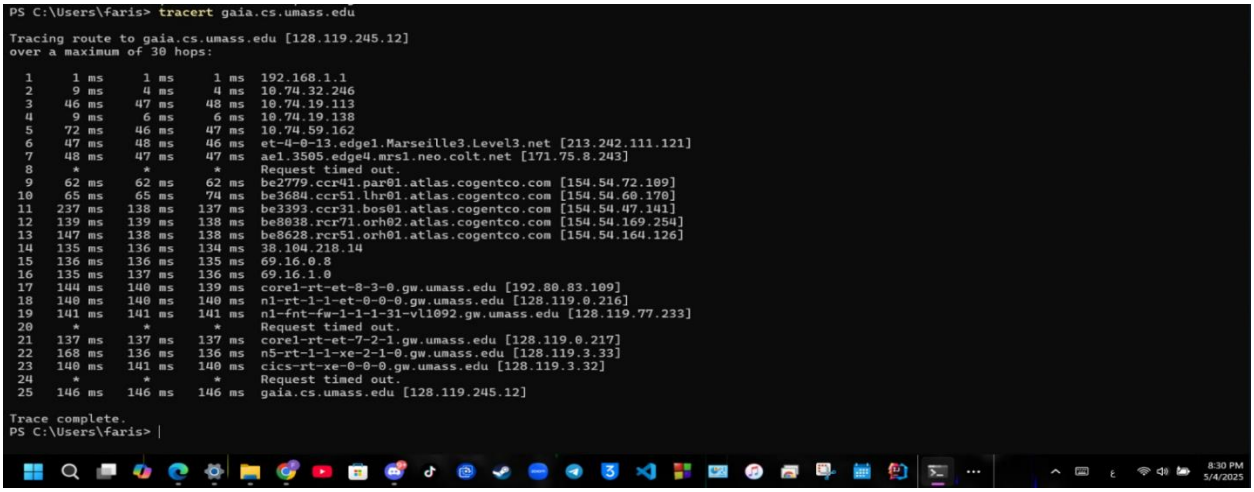


Figure 11: Traceroute to gaia.cs.umass.edu

The tracert (Traceroute) command was executed to trace the route taken by packets to reach the remote host gaia.cs.umass.edu (IP: 128.119.245.12).

Observations:

- The trace traverses **25 hops** from the local gateway to the remote host.
- The **first hop** (192.168.1.1) is the local router.
- The route passes through multiple intermediate Internet Service Providers (ISPs), including:
    - **Level3 (Marseille3.Level3.net)** at hop 6
    - **Neo.Colt.net** and **Cogentco.com** (global backbone networks)
    - **UMass routers** at hops 16–25

Important Nodes Noted:

- **Hop 6–14:** The packets travel through various backbone and Tier 1 network providers such as cogentco.com.
- **Hop 17–23:** The route reaches internal UMass routers, including:
    - n1-rt-1-1-et-0-0-0.gw.umass.edu
    - n1-fnt-fw-1-1-1-31-vl1092.gw.umass.edu
    - cics-rt-xe-0-0-0.gw.umass.edu

Request Timed Out:

- Some hops (e.g., hop 8, 20, 24) returned Request timed out, which is **common** in traceroute results. These routers may be **configured to not respond** to ICMP for security or load-balancing purposes. This **does not indicate packet loss** or connection failure.

Final Hop:

- The trace completed successfully at **hop 25**, reaching gaia.cs.umass.edu with a latency of **146 ms**, matching the earlier ping test results.

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

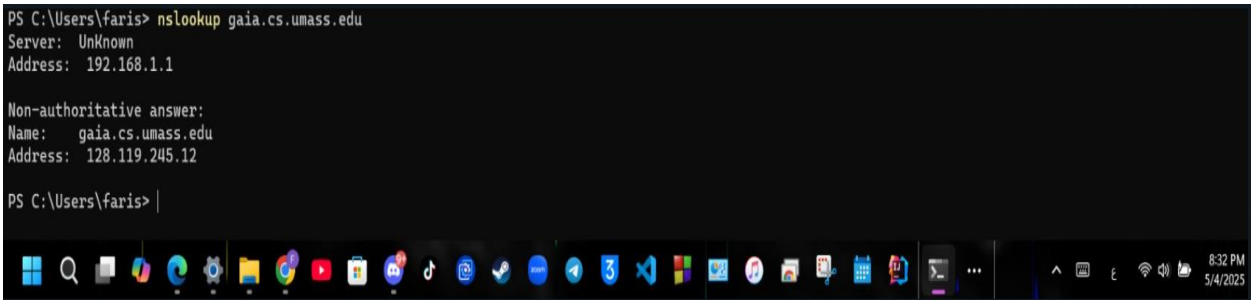


Figure 12: nslookup query for gaia.cs.umass.edu

The command nslookup gaia.cs.umass.edu was used to query the Domain Name System (DNS) and resolve the domain name to its corresponding IP address.

Results:

- **DNS Server Used:** 192.168.1.1 (local gateway/router)
- **Domain Name Queried:** gaia.cs.umass.edu
- **Resolved IP Address:** 128.119.245.12
- **Response Type:** Non-authoritative answer

Interpretation:

The DNS resolution was successful. The domain name gaia.cs.umass.edu was correctly translated into its IPv4 address 128.119.245.12. The response was marked as **non-authoritative**, which means the information was retrieved from a **cached source** or a DNS server that is **not the primary authority** for the domain.

This confirms that the domain is **active** and **properly configured** in the DNS system, allowing clients to locate and access its services over the internet.

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

Figure 13: Telnet connection to gaia.cs.umass.edu on port 80

Telnet Web Request to gaia.cs.umass.edu on Port 80 – Using macOS Terminal

remote web server gaia.cs.umass.edu on **port 80**, which is the standard port for HTTP services. After establishing the connection, the following HTTP request was manually sent:

GET / HTTP/1.1

Host: gaia.cs.umass.edu

Results:

- The server responded with a valid **HTTP/1.1 200 OK** status, confirming that the connection was successful and the server is reachable.
- It returned the full **HTML content of the homepage** of the Computer Network Research Group at UMass Amherst.
- The HTML response includes metadata, page structure, links to internal sections (e.g., research, publications), and text content rendered as raw HTML.
- The message Connection closed by foreign host. indicates that the server closed the connection gracefully after completing the response.

Interpretation:

This proves that the remote web server is:

- Online and reachable over the internet.
- Properly serving HTTP content over port 80.
- Able to handle raw HTTP requests made through Telnet, showcasing how low-level HTTP requests can be tested without a browser.

This method is commonly used to **test web server availability**, **inspect raw responses**, or troubleshoot firewall and application-level issues.

Provide details about the autonomous system (AS) number, number of IP addresses, prefixes, peers, and the name of Tier 1 ISP(s) associated with gaia.cs.umass.edu. Using any online tool.

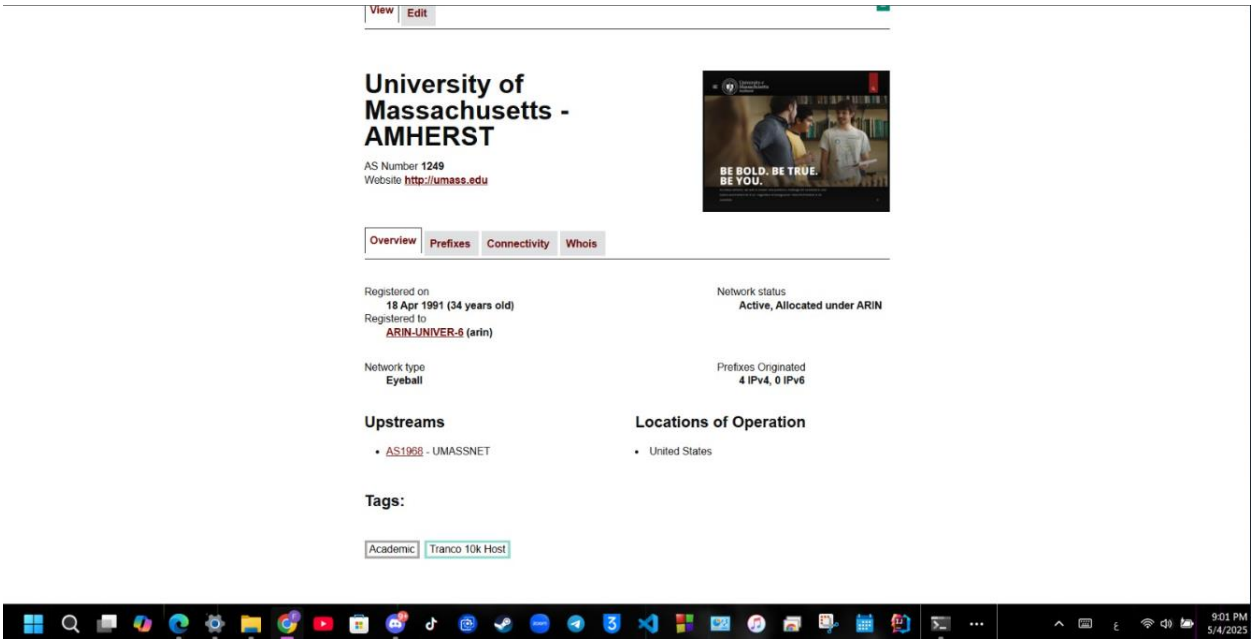


Figure 14: Autonomous System (AS) details of University of Massachusetts

The image displays detailed information about the **Autonomous System (AS)** associated with the host gaia.cs.umass.edu, retrieved from a BGP lookup tool.

AS Information for gaia.cs.umass.edu

- **AS Number:** AS1249 – University of Massachusetts - Amherst
- **Registered Since:** 18 April 1991 (34 years old)
- **Registered To:** ARIN-UNIVER-6 (ARIN)
- **Network Type:** Eyeball (end-user access)
- **Prefixes Originated:** 4 IPv4, 0 IPv6
- **Estimated IP Addresses:** ~1024 (based on standard prefix sizes)
- **Upstream Peer (Provider):** AS1968 – UMASSNET
- **Tier 1 ISP(s) via Routing Path:**
  - **Cogent Communications (AS174)**
  - **Level 3 Communications (AS3356)**
- **Location:** United States
- **Abuse Contact:** abuse@umass.edu

Using Wireshark to capture some DNS messages

This figure shows the DNS query initiated from the client machine (192.168.1.10) to the local DNS server (192.168.1.1) requesting the IP address for gaia.cs.umass.edu.

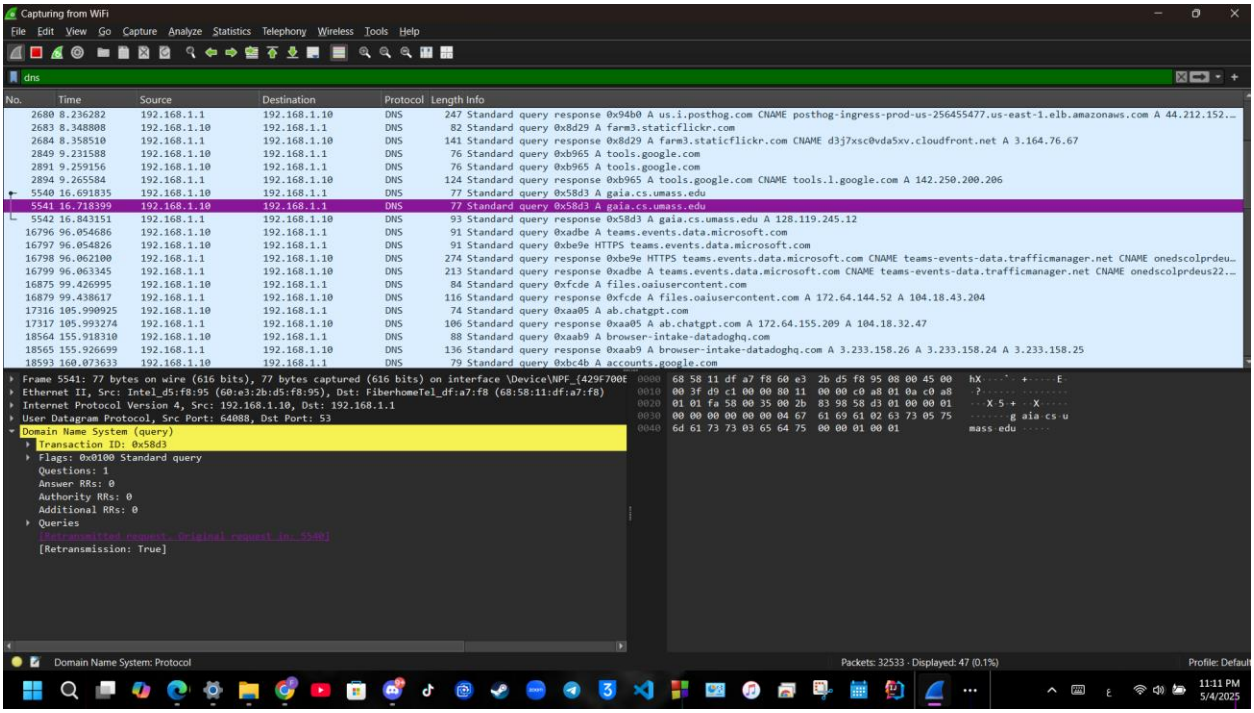


Figure 15: DNS query for gaia.cs.umass.edu

✅ What's Happening:

- The source device (192.168.1.10) is attempting to resolve the domain name gaia.cs.umass.edu by sending a DNS query to its configured DNS server (192.168.1.1).
- The protocol used is **DNS over UDP**, with **source port 64088** and destination **port 53** (standard DNS).
- This packet has a **Transaction ID of 0x58d3**, which is used to match requests and responses.

Retransmission Notice:

- Wireshark highlights that this is a **retransmitted DNS query**. The original request (frame 5541) might not have received a timely response, so the client retransmitted the same query.
- Retransmissions are common in UDP-based DNS because UDP does not guarantee delivery — clients may resend a query if they don't get a response quickly.

DNS Query Details:

- **Questions:** 1 (meaning one domain is being queried)
- **Query Name:** gaia.cs.umass.edu
- **Retransmission:** True (as shown in the bottom info box)

This figure displays the DNS response sent by the DNS server, resolving the domain name gaia.cs.umass.edu to the IP address 128.119.245.12.

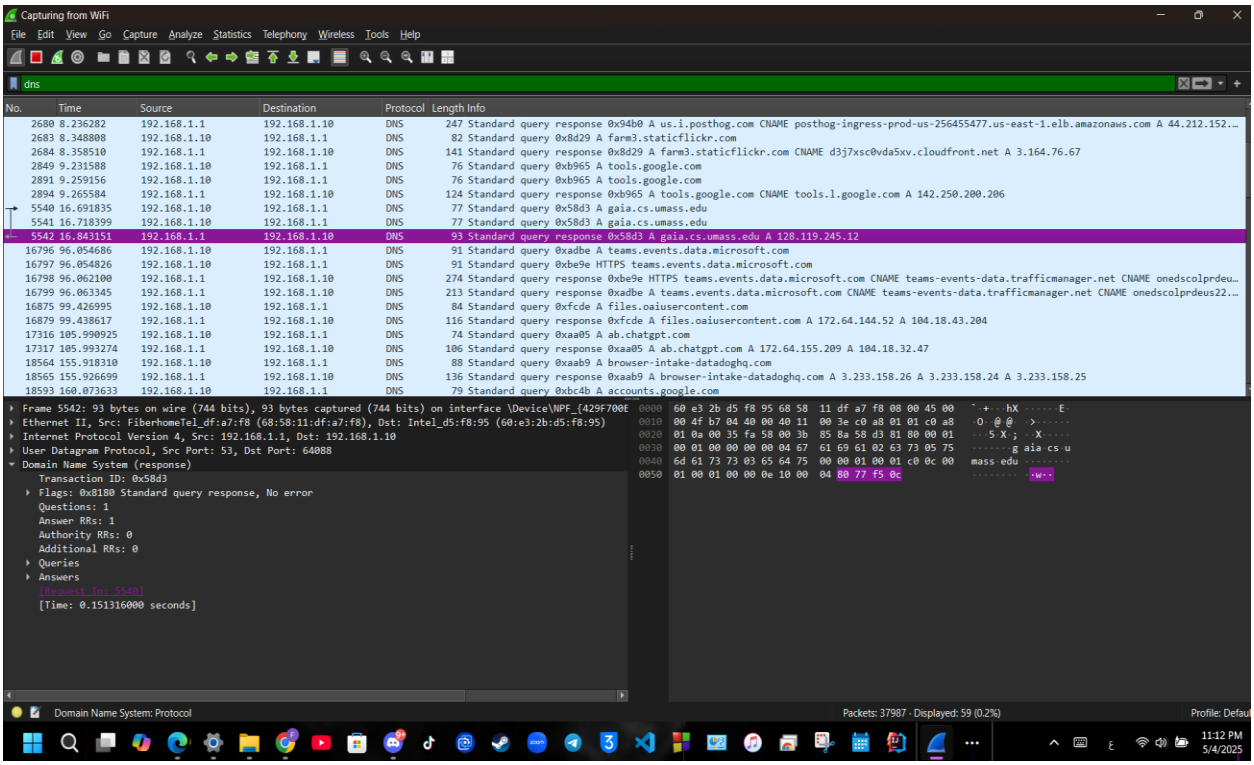


Figure 16: DNS response for gaia.cs.umass.edu showing IP address 128.119.245.12

✅ What's Happening:

- The DNS server (192.168.1.1) is responding to the client's earlier query from 192.168.1.10.
- The response contains a valid result for the domain:
  gaia.cs.umass.edu → 128.119.245.12
- This confirms that the DNS resolution was **successful**.

📌 Details in Wireshark:

- **Transaction ID:** 0x58d3 – matches the query ID from the previous screenshot, confirming it's the answer to that request.
- **Flags:** 0x8180 – indicates this is a standard response with **no error**.
- **Answer RRs:** 1 – one resource record (A record) is returned.
- **Answer Section:** shows the IP address 128.119.245.12 for the domain gaia.cs.umass.edu.
- **Response Time:** ~0.15 seconds from the retransmission (as indicated in the "Response In: 5542" field).

2) Task 2 – Web Server

In this task, we implemented a **lightweight web server** using **Python's socket programming** to simulate how a real **HTTP server** works at the **low level**. The server listens on a **custom port** derived from the last digits of the student ID (e.g., **9956**) and handles incoming **HTTP requests** by parsing them **manually**.

It responds to requests by serving either an **English web page** (main_en.html) or an **Arabic version** (main_ar.html), both of which include structured information such as **team member names**, **student IDs**, **personal skills**, and **project contributions**. Each page also contains a section presenting **educational content** from **Chapter 1** of the course book, mainly covering the **HTTP protocol**, along with embedded **images** and **ordered/unordered lists**.

The server is capable of returning **HTML**, **CSS**, and **image files** (e.g., .png, .jpg) from a /static/ directory using correct **Content-Type headers**. In case the requested file does not exist, the server responds with an appropriate **404 Not Found** error page.

Overall, this task gave us practical experience in:

- **Handling HTTP requests and responses manually**
- **Understanding how file serving and routing works on the web**
- **Learning the structure of HTTP messages**
- **Sending status codes and serving static content without using any external frameworks**

❖ Webpages

- **Main English Webpage**

This is the main page of the English version of our website. It includes the title, details about our team members, a link to the local version of the page (e.g., supporting materials), and other relevant information.

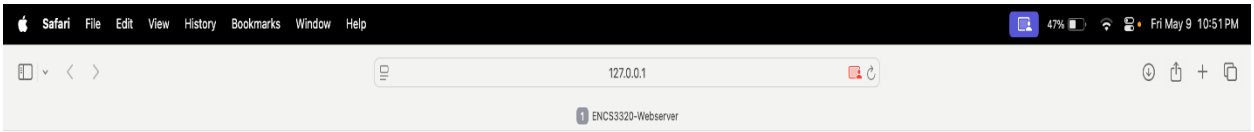


Figure 17: Main English Webpage

We see in figure 17, this is the main English webpage the web browser tab displayed (ENCS3320-Webserver), and the title of the page is (Welcome to ENCS3320 Computer Networks Webserver)

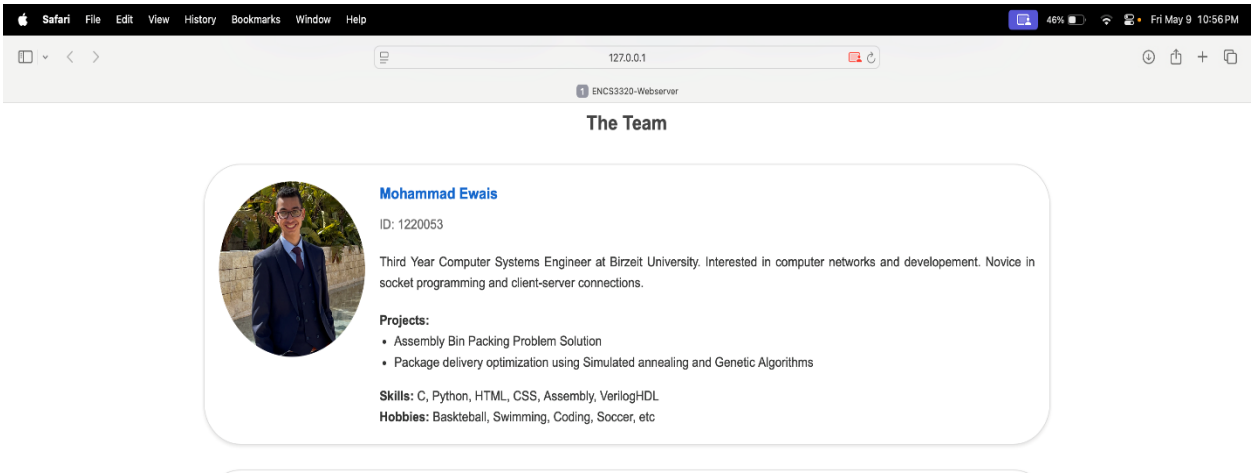


Figure 18: Main English Webpage (CONT.)

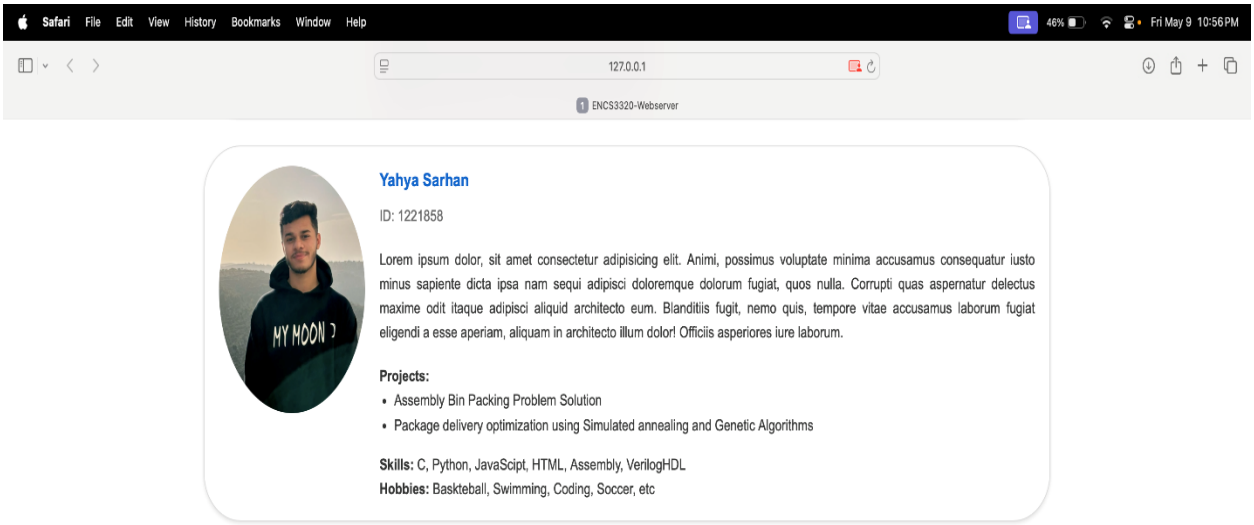


Figure 19: Main English Webpage (CONT.)

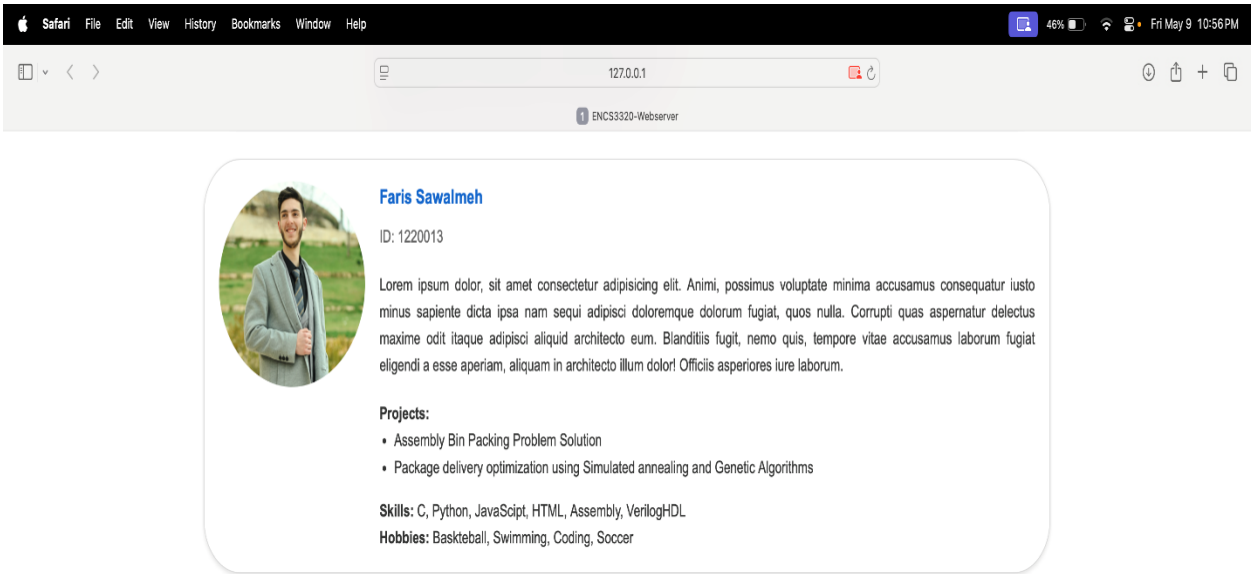


Figure 20: Main English Webpage (CONT.)

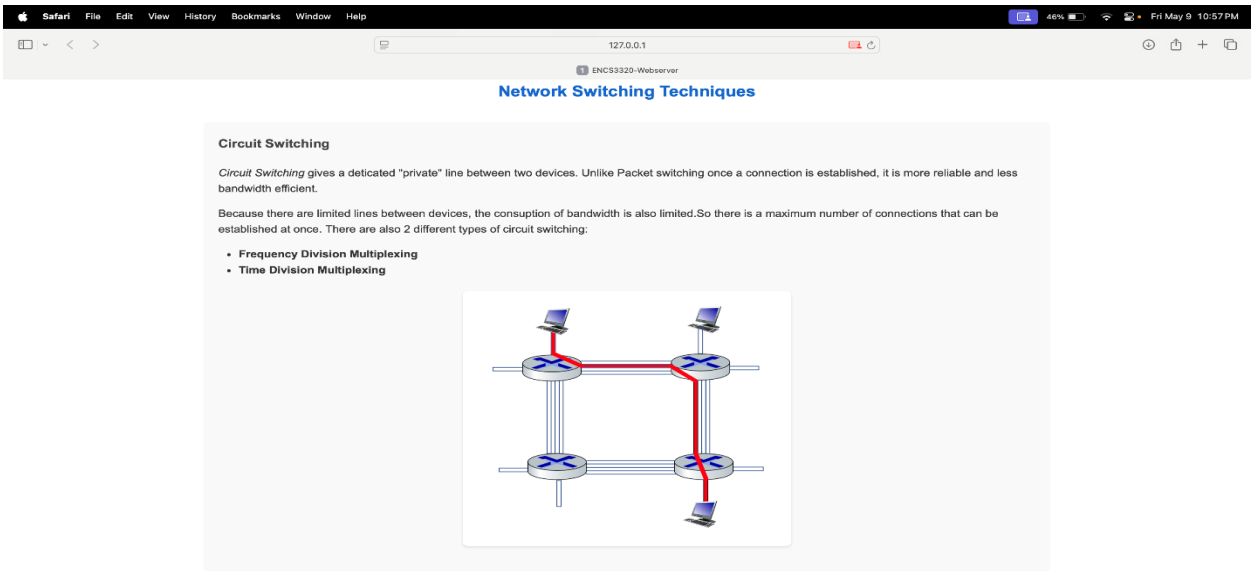


Figure 21: Main English Webpage (CONT.)

This image in figure 21 illustrates the concept of **Circuit Switching**, where a **dedicated and fixed line** is established between two devices for the entire duration of the communication. The diagram shows **routers** connected via **highlighted red paths**, representing the **exclusive circuit** created between devices. This visual emphasizes the **direct and continuous connection** characteristic of circuit-switched networks.

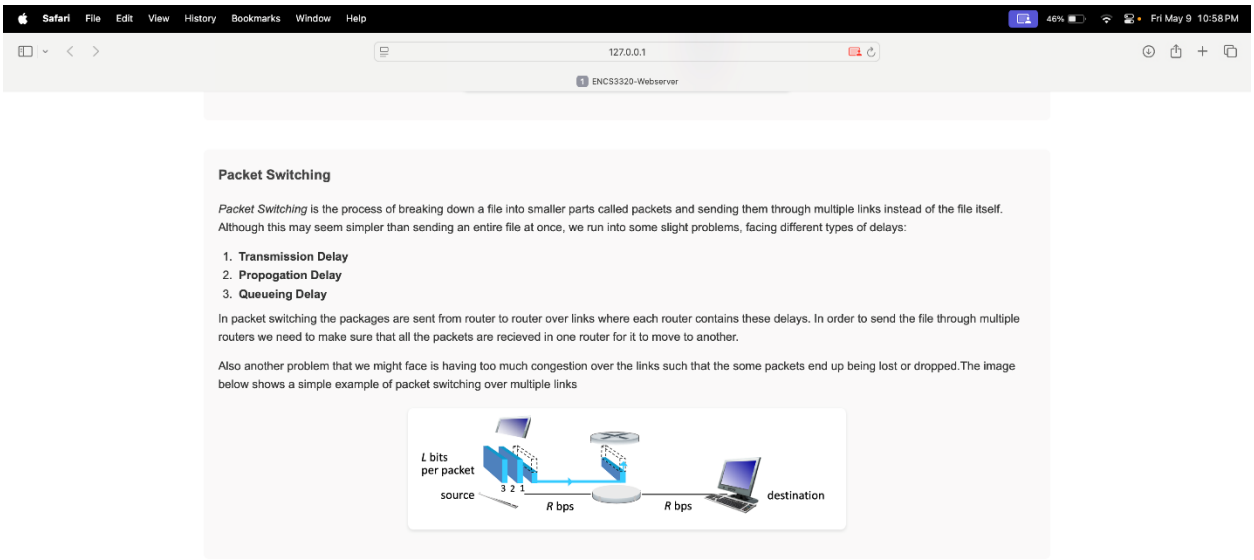


Figure 22: Main English Webpage (CONT.)

This image in figure 22 demonstrates the principle of **Packet Switching**, where data is divided into **smaller packets** that travel across multiple **network links** to reach their destination. The diagram shows the flow of packets from the **source to the destination**, with labels indicating **transmission rate (R bps)** and **packet size (L bits)**. It highlights the **non-sequential yet efficient** method of data transfer used in packet-switched systems.

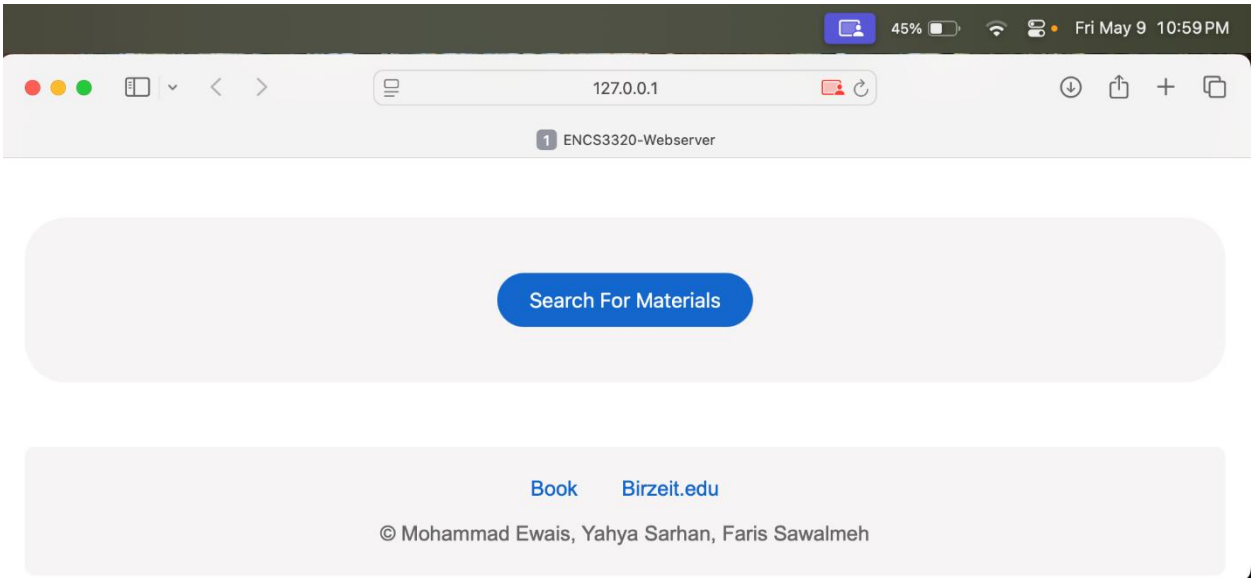


Figure 23: Main English Webpage (CONT.)

This image in figure 23 shows the lower section of the main_en.html page served by the custom web server. The page includes a centered **"Search For Materials"** button allowing users to input a keyword and search for related resources. Below it, external links to the **course textbook** and the **Birzeit University website** are provided.

- **Searching material English Webpage**
  This page serves as a **supporting material search interface**, allowing users to request a specific **image or video** related to networking by entering the filename into a text input field. The **web server processes the request**: if the requested file is found in the server's /static/ directory, it is returned and displayed; otherwise, the server issues a **307 Temporary Redirect**. Depending on the file type, the user is redirected to an external platform—**Google Images** for image files or **YouTube** for video files—where they can find related content. This feature demonstrates practical handling of **conditional routing**, **HTTP redirection**, and the server's ability to differentiate between file types based on extensions.

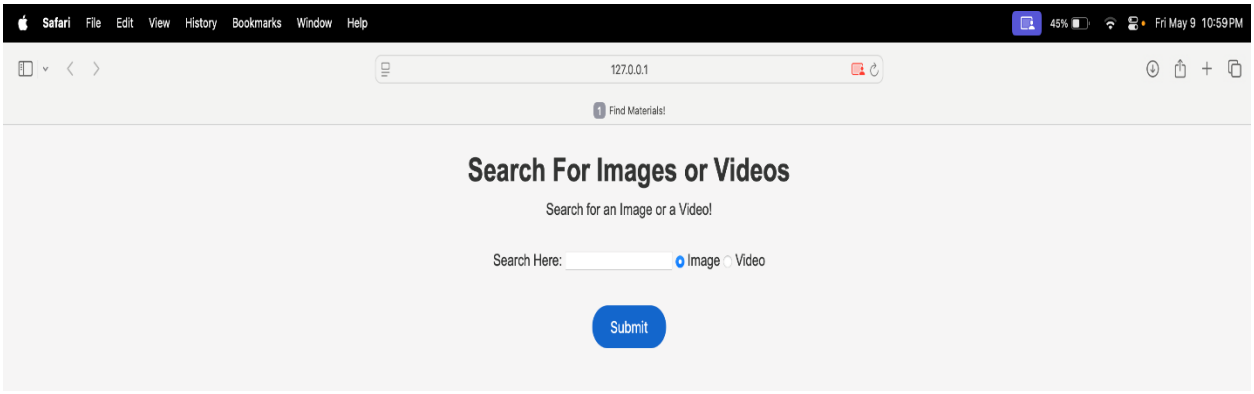


Figure 24: Searching Material English Webpage

In our project, the user is asked to **enter the filename** of a requested resource (either an image or a video) into an **input field**, and then choose the file type by selecting one of the **radio buttons** labeled "Image" or "Video." Once the user clicks the **Submit** button, the request is sent to the server. The server checks if the file exists in the /static/ directory. If it is available, the server responds by sending the requested file to the browser. If the file is **not found**, the server responds with a **307 Temporary Redirect**, guiding the browser to perform a **Google image search** or a **YouTube search**, depending on the selected file type. This feature highlights how our server handles **dynamic file requests**, **conditional responses**, and proper **HTTP redirection logic**.

- **Main Arabic Webpage**

This webpage serves as the **main page** of the **Arabic version**, containing the **same content** as the **main English webpage**, but fully **translated into Arabic**.

Figure 25: Main Arabic Webpage

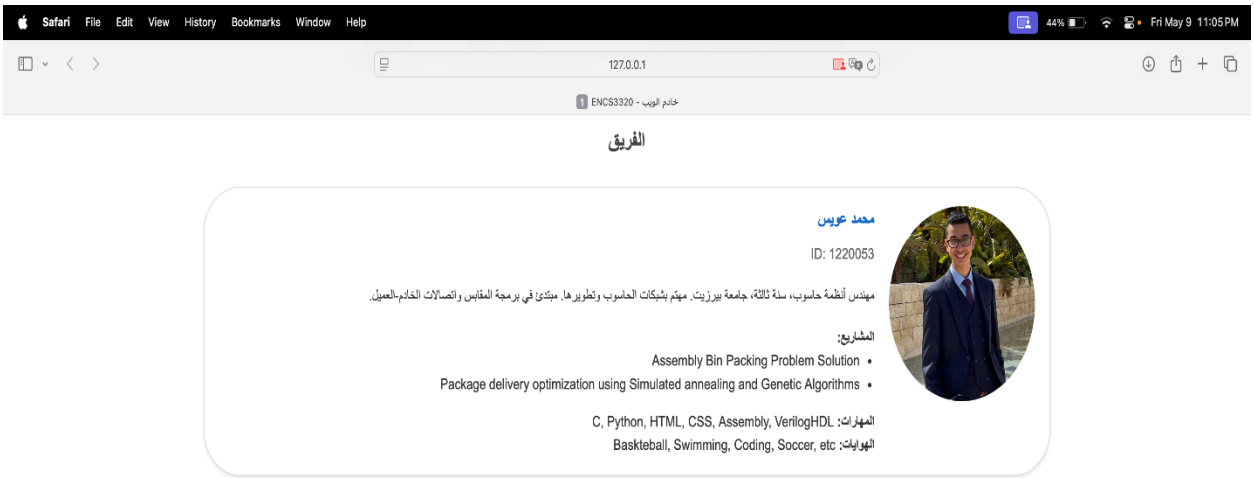


Figure 26: Main Arabic Webpage (CONT.)

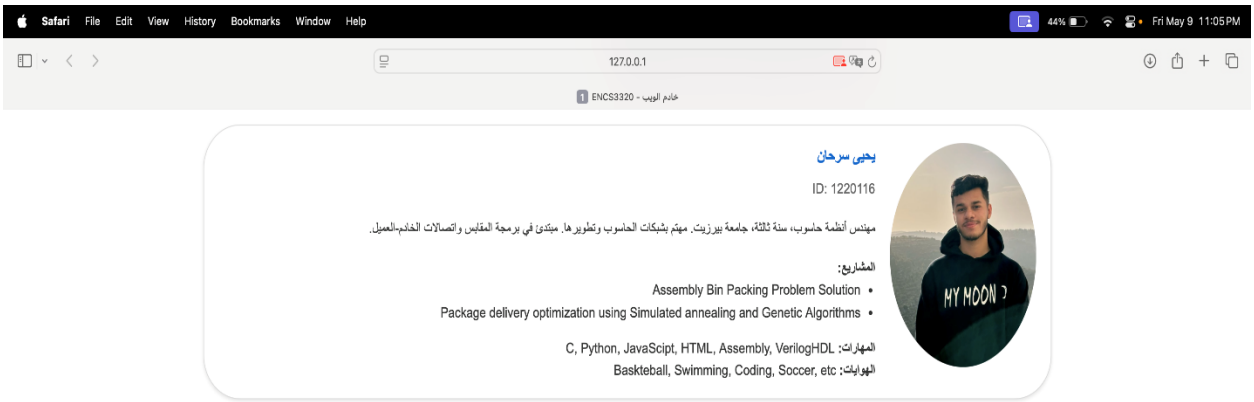


Figure 27: Main Arabic Webpage (CONT.)

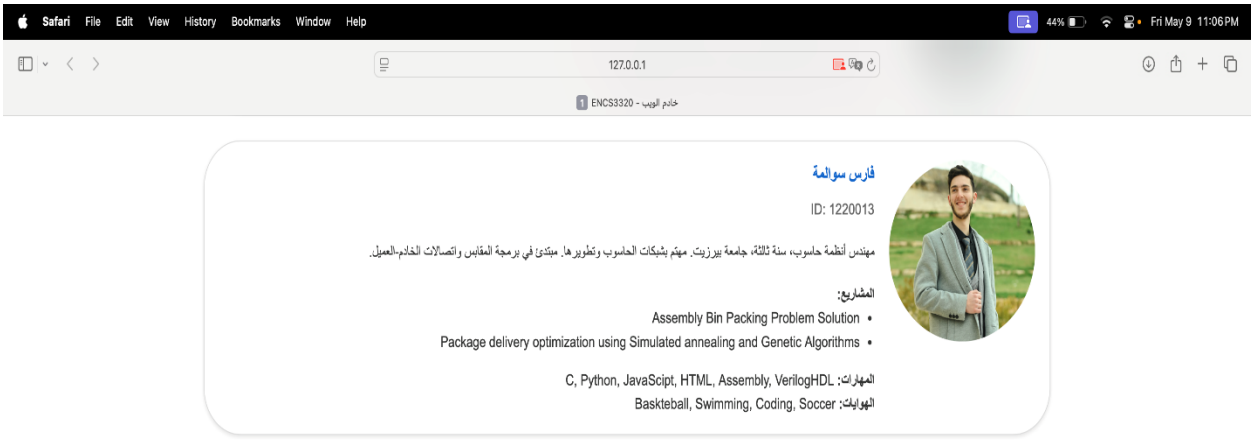


Figure 28: Main Arabic Webpage (CONT.)

In the three figures above (25, 26 and 27), there is the team members details in Arabic language

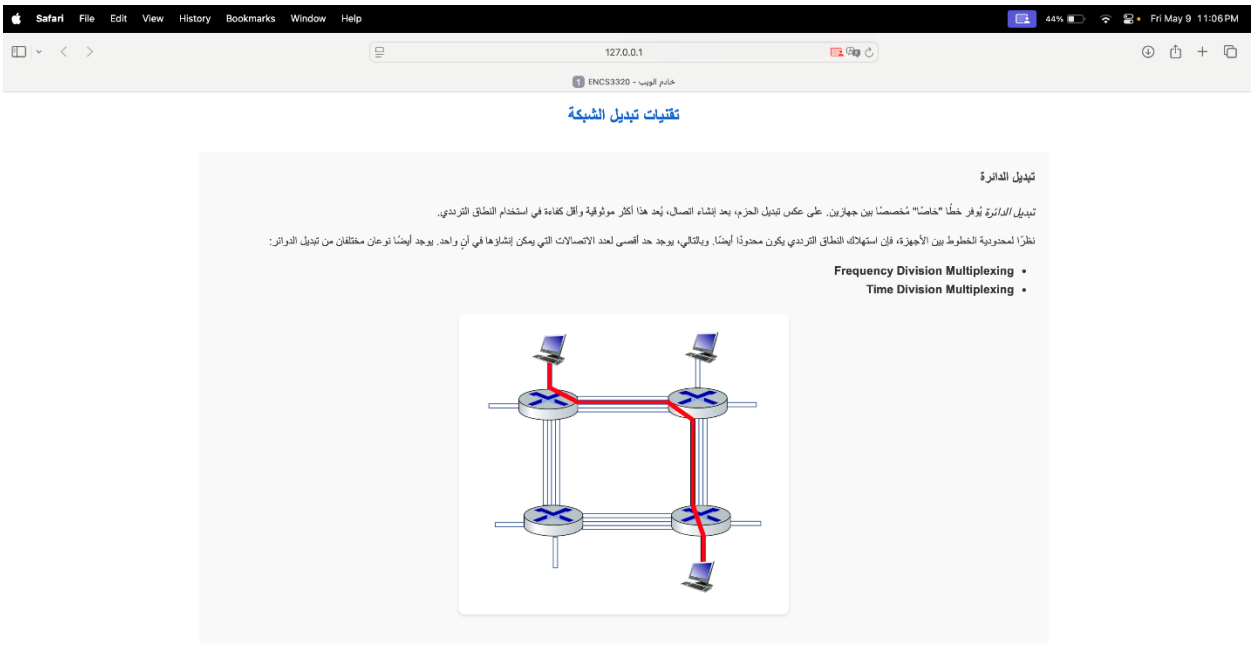


Figure 29: Main Arabic Webpage (CONT.)

This image in figure 28 illustrates the concept of **Circuit Switching,** but in Arabic version and the same explanation that I wrote in English.

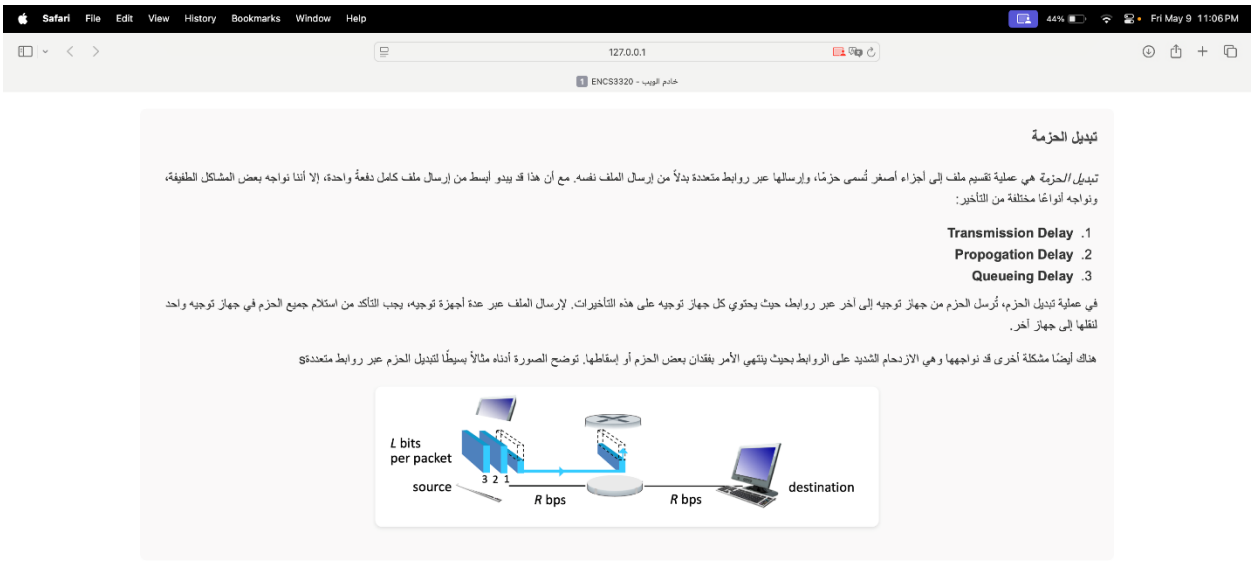


Figure 30: Main Arabic Webpage (CONT.)

This image in figure 29 illustrates the concept of **Packet Switching,** but in Arabic version and the same explanation that I wrote in English.

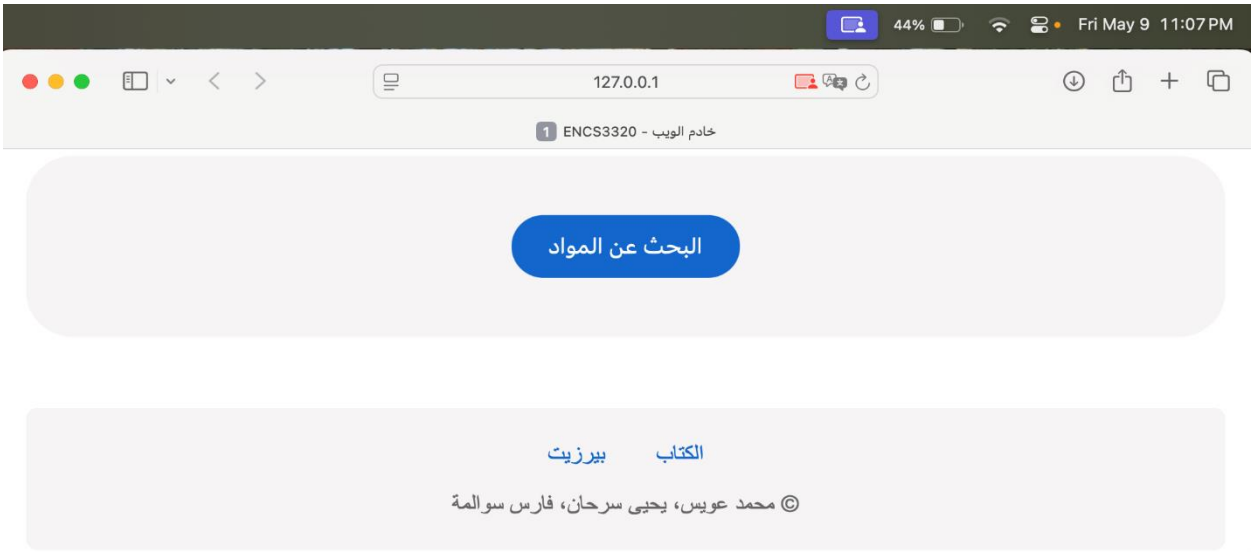


Figure 31: Main Arabic Webpage (CONT.)

This image in Figure 31 shows the lower section of the main_ar.html page served by the custom web server. The page includes a centered "Search For Materials" button, allowing users to input a keyword and search for related resources. The layout and structure are fully adapted for **Arabic language support**, with proper **right-to-left (RTL)** alignment. Below the search section, the page provides external links to the course textbook and the **Birzeit University** website, both presented in **Arabic interface** for better accessibility.

- **Search Material Arabic Webpage**

  This webpage is the main page in the Arabic version, it includes exactly the same content of main English webpage but in the Arabic language.

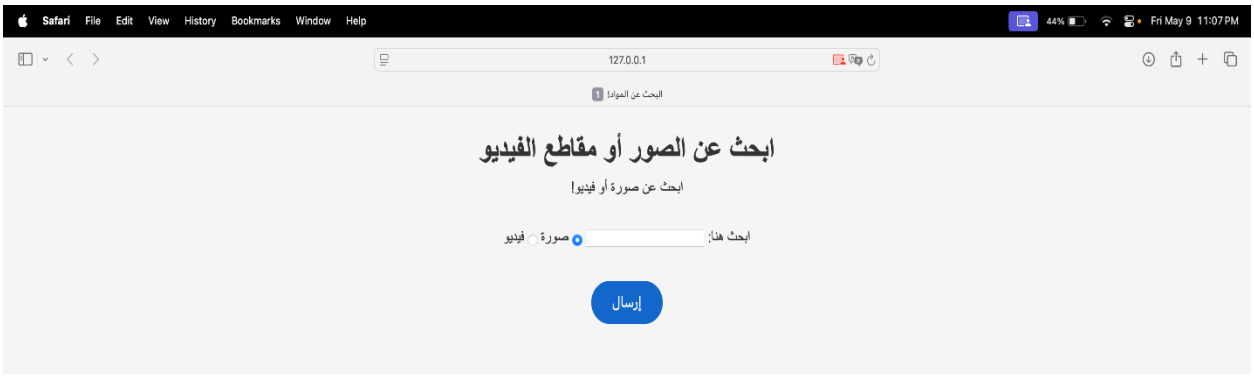
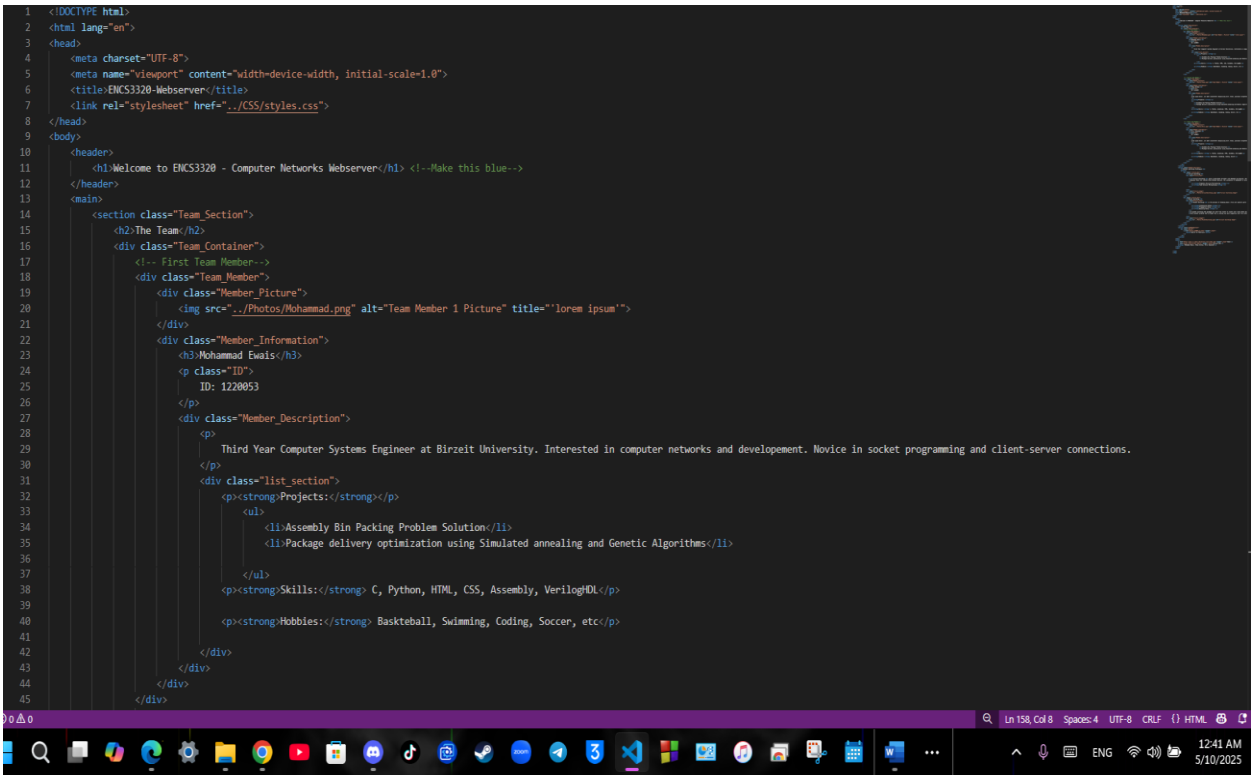


Figure 32: Search Material Arabic Webpage

All of these **webpages** (e.g., **Main English**, **Main Arabic**, **Supporting Material English**, and **Supporting Material Arabic**) are **designed** and **visually styled** in an **attractive format** using a **separate CSS file**.

❖ 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">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ENCS3320-Webserver</title>
    <link rel="stylesheet" href="../CSS/styles.css">
</head>
<body>
    <header>
        <h1>Welcome to ENCS3320 - Computer Networks Webserver</h1> <!--Make this blue-->
    </header>
    <main>
        <section class="Team_Section">
            <h2>The Team</h2>
            <div class="Team_Container">
                <!-- First Team Member-->
                <div class="Team_Member">
                    <div class="Member_Picture">
                        <img src="../Photos/Mohammad.png" alt="Team Member 1 Picture" title="'lorem ipsum'">
                    </div>
                    <div class="Member_Information">
                        <h3>Mohammad Ewais</h3>
                        <p class="ID">
                            ID: 1220053
                        </p>
                        <div class="Member_Description">
                            <p>
                                Third Year Computer Systems Engineer at Birzeit University. Interested in computer networks and developement. Novice in socket programming and client-server connections.
                            </p>
                            <div class="list_section">
                                <p><strong>Projects:</strong></p>
                                    <ul>
                                        <li>Assembly Bin Packing Problem Solution</li>
                                        <li>Package delivery optimization using Simulated annealing and Genetic Algorithms</li>

                                    </ul>
                                <p><strong>Skills:</strong> C, Python, HTML, CSS, Assembly, VerilogHDL</p>

                                <p><strong>Hobbies:</strong> Baskteball, Swimming, Coding, Soccer, etc</p>

                            </div>
                        </div>
                    </div>
                </div>
```

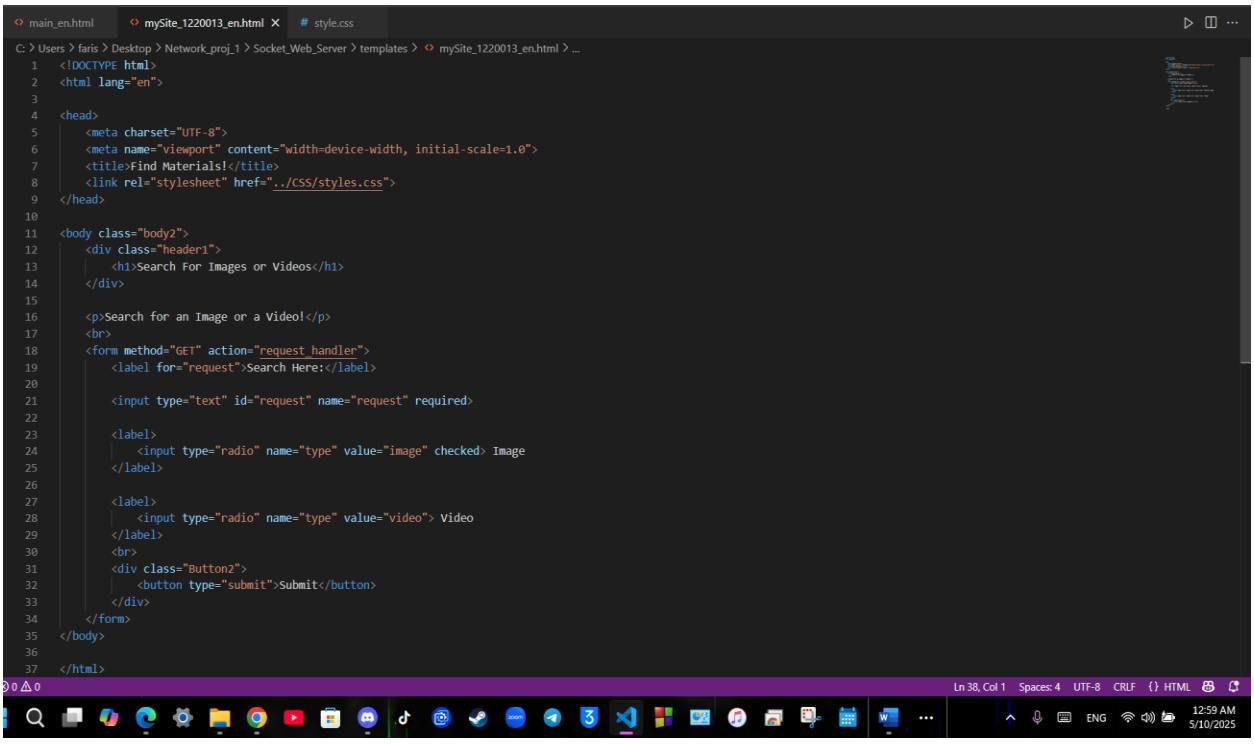
```html
                <!-- Second Team Member-->
                <div class="Team_Member">
                    <div class="Member_Picture">
                        <img src="../Photos/Yahya.png" alt="Team Member 1 Picture" title="'lorem ipsum'">
                    </div>
                    <div class="Member_Information">
                        <h3>Yahya Sarhan</h3>
                        <p class="ID">
                            ID: 1221858
                        </p>
                        <div class="Member_Description">
                            <p>
                            Lorem ipsum dolor, sit amet consectetur adipisicing elit. Animi,
                            possimus voluptate minima accusamus consequatur iusto minus sapiente dicta ipsa nam sequi adipisci doloremque dolorum fugiat,
                            quos nulla. Corrupti quas aspernatur delectus maxime odit itaque adipisci aliquid architecto eum. Blanditiis fugit, nemo quis,
                            tempore vitae accusamus laborum fugiat eligendi a esse aperiam, aliquam in architecto illum dolor! Officiis asperiores iure laborum.
                            </p>
                            <p><strong>Projects:</strong></p>
                            <ul>
                                <li>Assembly Bin Packing Problem Solution</li>
                                <li>Package delivery optimization using Simulated annealing and Genetic Algorithms</li>

                            </ul>
                            <p><strong>Skills:</strong> C, Python, JavaScipt, HTML, Assembly, VerilogHDL</p>

                            <p><strong>Hobbies:</strong> Baskteball, Swimming, Coding, Soccer, etc</p>

                        </div>
                    </div>
                </div>
```

```
            <!-- Third Team Member-->
            <div class="Team_Member">
                <div class="Member_Picture">
                    <img src="../Photos/Faris.png" alt="Team Member 3 Picture" title="'lorem ipsum'">
                </div>
                <div class="Member_Information">
                    <h3>Faris Sawalmeh</h3>
                    <p class="ID">
                        ID: 1220013
                    </p>
                    <div class="Member_Description">
                        <p>
                        Lorem ipsum dolor, sit amet consectetur adipisicing elit. Animi, possimus voluptate minima accusamus consequatur iusto minus sapiente dicta ipsa nam sequi adipisci doloremque dolorum fugiat,
                        quos nulla. Corrupti quas aspernatur delectus maxime odit itaque adipisci aliquid architecto eum. Blanditiis fugit,
                        nemo quis, tempore vitae accusamus laborum fugiat eligendi a esse aperiam, aliquam in architecto illum dolor! Officiis asperiores iure laborum.
                        </p>
                        <p><strong>Projects:</strong></p>
                                <ul>
                                    <li>Assembly Bin Packing Problem Solution</li>
                                    <li>Package delivery optimization using Simulated annealing and Genetic Algorithms</li>

                                </ul>
                        <p><strong>Skills:</strong> C, Python, JavaScipt, HTML, Assembly, VerilogHDL</p>

                        <p><strong>Hobbies:</strong> Baskteball, Swimming, Coding, Soccer</p>

                    </div>
                </div>
            </div>
        </div>
    </section>
```

```
    <section class="Chapter_One_Topic">
        <h2>Network Switching Techniques</h2>
        <article>
            <div class=" Article_div">
                <h3>Circuit Switching</h3>
                <div class="Article_Text">

                    <p><em>Circuit Switching</em> gives a deticated "private" line between two devices. Unlike Packet switching once a connection is established, it is more reliable and less bandwidth efficient.</p>
                    <p>Because there are limited lines between devices, the consuption of bandwidth is also limited.So there is a maximum number of connections that can be established at once.
                      There are also 2 different types of circuit switching:</p>
                    <ul>
                        <li><strong>Frequency Division Multiplexing</strong></li>
                        <li><strong>Time Division Multiplexing</strong></li>
                    </ul>

                </div>
                <div class="Article_Image">
                    <img src="../Photos/CircuitSwitching.jpeg" alt="Circuit Switching Image">
                </div>
            </div>
            <div class=" Article_div">
                <h3>Packet Switching</h3>
                <div class="Article_Text">
                    <p><em>Packet Switching</em> is the process of breaking down a file into smaller parts called packets and sending them through multiple links instead of the file itself.
                          Although this may seem simpler than sending an entire file at once, we run into some slight problems, facing different types of delays:</p>
                    <ol>
                        <li><strong>Transmission Delay</strong></li>
                        <li><strong>Propogation Delay</strong></li>
                        <li><strong>Queueing Delay</strong></li>
                    </ol>
                    <p>In packet switching the packages are sent from router to router over links where each router contains these delays.
                       In order to send the file through multiple routers we need to make sure that all the packets are recieved in one router for it to move to another. </p>
                    <p>Also another problem that we might face is having too much congestion over the links such that the some packets end up being lost or dropped.The image below shows a simple example of packet switching over multiple links </p>

                </div>
                <div class="Article_Image2">
                    <img src="../Photos/PacketSwitching.jpg" alt="Circuit Switching Image">
                </div>
            </div>
        </article>
    </section>
```

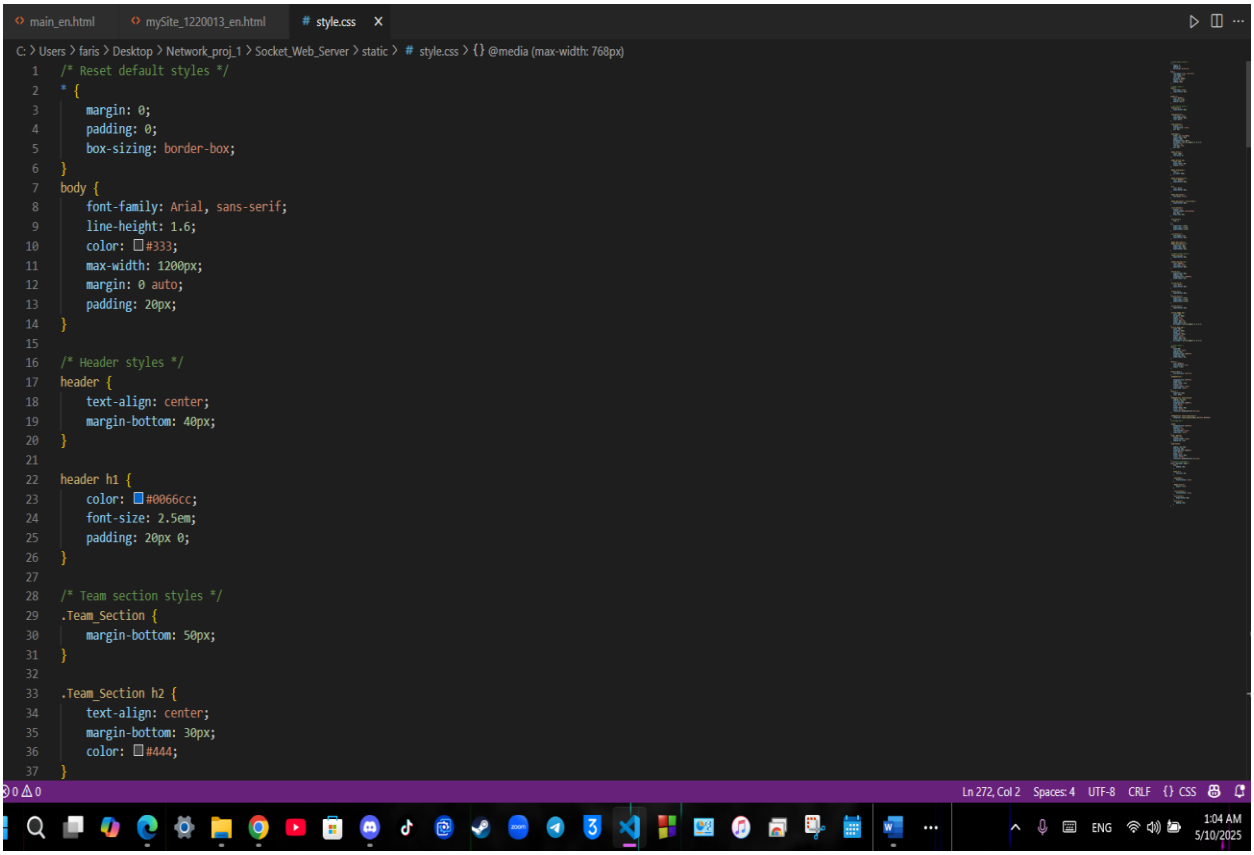
```
        <section class="SubPageSection">
            <div class="Button">
                <a href="mySite_1220053_en.html" target="_blank">
                    <button>Search For Materials</button>
                </a>
            </div>
        </section>
    </main>
    <footer>
        <a href="https://gaia.cs.umass.edu/kurose_ross/index.php" target="_blank">Book</a>
        <a href="https://www.birzeit.edu/ar" target="_blank">Birzeit.edu</a>
        <p>&copy; Mohammad Ewais, Yahya Sarhan, Faris Sawalmeh</p>
    </footer>
    </main>

</body>
</html>
```

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



❖ CSS Code

The CSS Code that style all the Webpages in attractive way:



```css
.Team_Container {
    display: flex;
    flex-direction: column;
    gap: 30px;
}

.Team_Member {
    border: 1px solid #ddd;
    border-radius: 50px;
    padding: 20px;
    background-color: #fff;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
    display: flex;
    flex-wrap: wrap;
    gap: 20px;
}

.Member_Picture {
    width: 200px;
    flex-shrink: 0;
}

.Member_Picture img {
    width: 100%;
    border-radius: 50%;
    display: block;
}

.Member_Information {
    flex: 1;
    min-width: 300px;
}
```

```css
.Member_Information h3 {
    color: #0066cc;
    margin-bottom: 10px;
}

.ID {
    color: #666;
    margin-bottom: 20px;
}

.Member_Description {
    text-align: justify;
}

.Member_Description > p:first-child {
    margin-bottom: 20px;
}

.lists-container {
    display: flex;
    justify-content: space-between;
    gap: 20px;
    margin-top: 20px;
}

.list-section {
    flex: 1;
}
ol{
    margin-left: 1.75rem;
    margin-right: 1.75rem;
    margin-bottom: 0.5rem;
}
```

```css
.list-section p {
    font-weight: bold;
    margin-bottom: 10px;
}

.Member_Description ul,
.Member_Description ol {
    margin-left: 20px;
    margin-right: 20px;
    margin-bottom: 15px;
}

/* Chapter section styles */
.Chapter_One_Topic {
    margin-bottom: 50px;
}

.Chapter_One_Topic h2 {
    color: #0066cc;
    text-align: center;
    margin-bottom: 30px;
}

.Article_div {
    margin-bottom: 40px;
    padding: 20px;
    background-color: #f9f9f9;
    border-radius: 8px;
}

.Article_div h3 {
    color: #444;
    margin-bottom: 20px;
}

.Article_Text {
    margin-bottom: 20px;
}
.Article_Text ul{
    margin-left: 1.75rem;
    margin-right: 1.75rem;
    margin-bottom: 0.5rem;
}
```

```css
.Article_Text p {
    margin-bottom: 15px;
}

.Article_Image2 img {
    width: 100%;
    max-width: 600px;
    height: auto;
    display: block;
    margin: 20px auto;
    border-radius: 8px;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}
.Article_Image img {
    width: 100%;
    max-width: 450px;
    height: auto;
    max-height: 450px;
    display: block;
    margin: 20px auto;
    border-radius: 8px;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

/* Footer styles */
footer {
    width:100%;
    text-align: center;
    padding: 20px;
    background-color: #f5f5f5;
    margin-top: 50px;
    border-radius: 8px;
}

footer a {
    color: #0066cc;
    text-decoration: none;
    margin: 0 15px;
}
```

```css
footer a:hover {
    text-decoration: underline;
}
.SubPageSection {

    background-color: #f5f5f5;
    height:8rem;
    border-radius: 2rem;
    display:flex;
    justify-content: center;
    align-items: center;
}
footer p {
    margin-top: 10px;
    color: #666;
}
.SubPageSection .Button Button{
    padding: 12px 25px;
    font-size: 16px;
    background-color: #0066cc;
    color: white;
    border: none;
    border-radius: 30px;
    cursor: pointer;
    transition: background-color 0.3s ease;
}

.SubPageSection .Button button:hover {
    background: linear-gradient(20deg, #4CAF50, #FF9800);
}
/* Sub page CSS */

.body2{
    background-color: #f5f5f5;
    display:flex;
    flex-wrap: wrap;
    flex-direction: column;
    align-items: center;
}
.body2 .Button2{
    display: flex;
    justify-content: center;
    padding-top: 2rem;
}
```

```
.body2 Button{

    padding: 12px 25px;
    font-size: 16px;
    background-color: #0066cc;
    color: white;
    border: none;
    border-radius: 30px;
    cursor: pointer;
    transition: background-color 0.3s ease;
}
/* Responsive adjustments */
@media (max-width: 768px) {
    body {
        padding: 10px;
    }

    header h1 {
        font-size: 2em;
    }

    .Team_Member {
        flex-direction: column;
    }

    .Member_Picture {
        margin: 0 auto;
    }

    .lists-container {
        flex-direction: column;
    }
    .list-section {
        margin-bottom: 20px;
    }
    .Article_div {
        padding: 15px;
    }
}
```

❖ Web Server Implementation & Code

➢ Introduction

The objective of this task is to implement a complete **web server** using **socket programming**. The server should establish a **client-server connection** and **listen** on port **9903**, which is derived from a **teammate's ID**.

➢ Theoretical Concepts and Background

1. Socket

A **socket** serves as an **endpoint** for **sending** and **receiving** data across a **network**. It is uniquely defined by a combination of:

- An **IP address**
- A **port number**

There are two primary types of sockets:

- **TCP Socket** (**Transmission Control Protocol**):
    - Provides **reliable**, **connection-oriented** communication.
    - Ensures data is received in **order** and **without loss**.
    - Suitable for applications requiring **complete data transfer**.
- **UDP Socket** (**User Datagram Protocol**):
    - Offers **connectionless**, **faster** communication.
    - Less reliable as it does **not guarantee delivery** or **order**.
    - Useful in **real-time applications** like **video streaming** or **online gaming**.

2. Common Socket Functions

- Socket (): for creating a socket
- Bind (): binds a socket to a certain port and IP address
- Listen (): listens for connections
- Accept (): accepts a connection
- Send (), Recv (): sends or receives data over the socket
- Close (): closes the socket

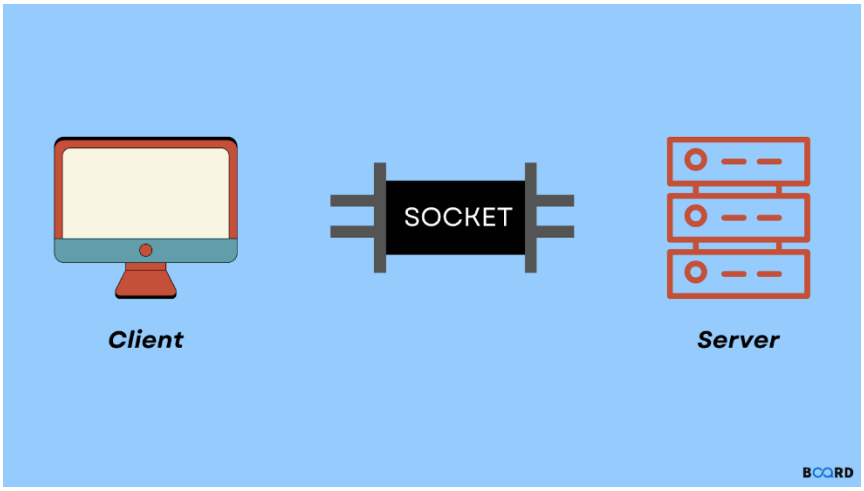


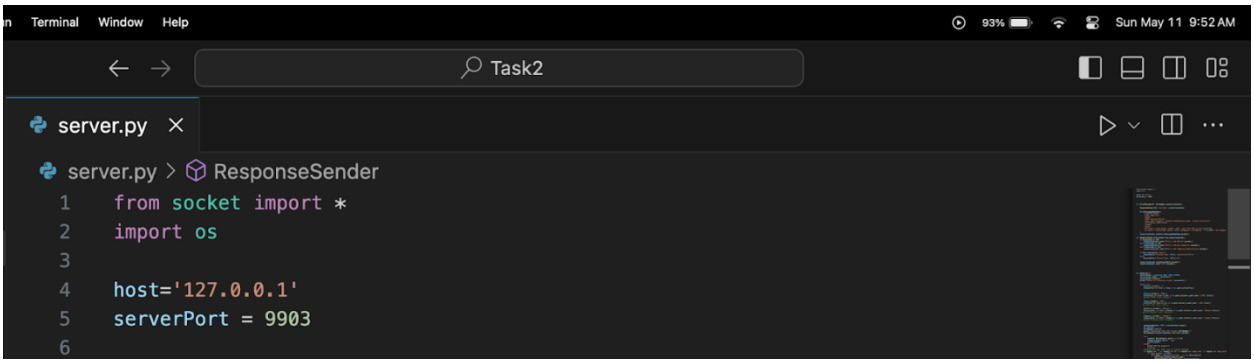
Figure 33: Illustration of client-server communication using a socket connection.

Server Implementation

In our server code (server.py) there exists 3 functions:

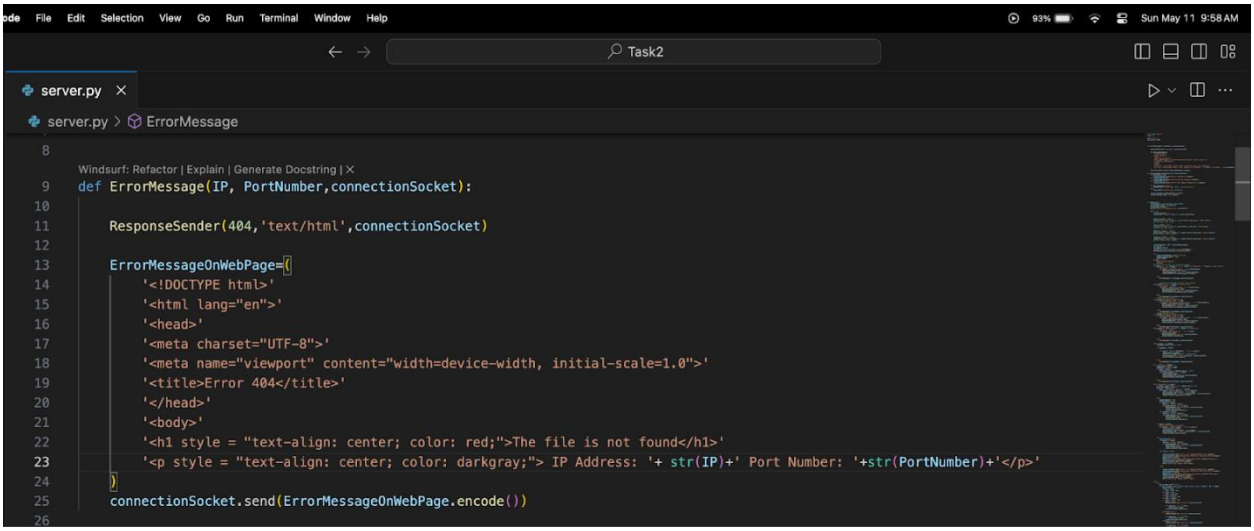
1. ErrorMessage
2. ResponseSender
3. RunServer

Before we begin with the functions we initially imported all the functions from the socket library and also imported the os



We also gave the server a port number to listen to which was chosen 99(03) from a teammate's ID.

Now, For the First Function “ErrorMessage” we parsed the IP address and the PortNumber of the client and the connection socket.



We then sent a 404 status code to the server through a function we will discuss soon and printed an error message in red on the webpage with the port number and IP address. This shows up as follows:

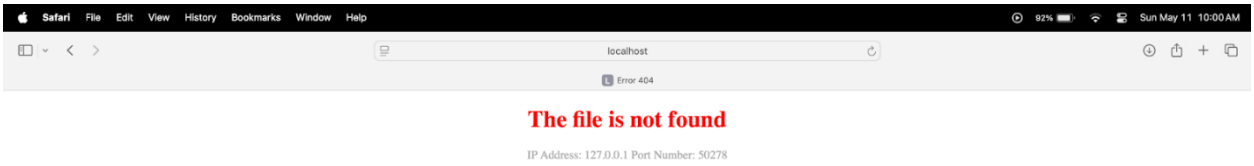
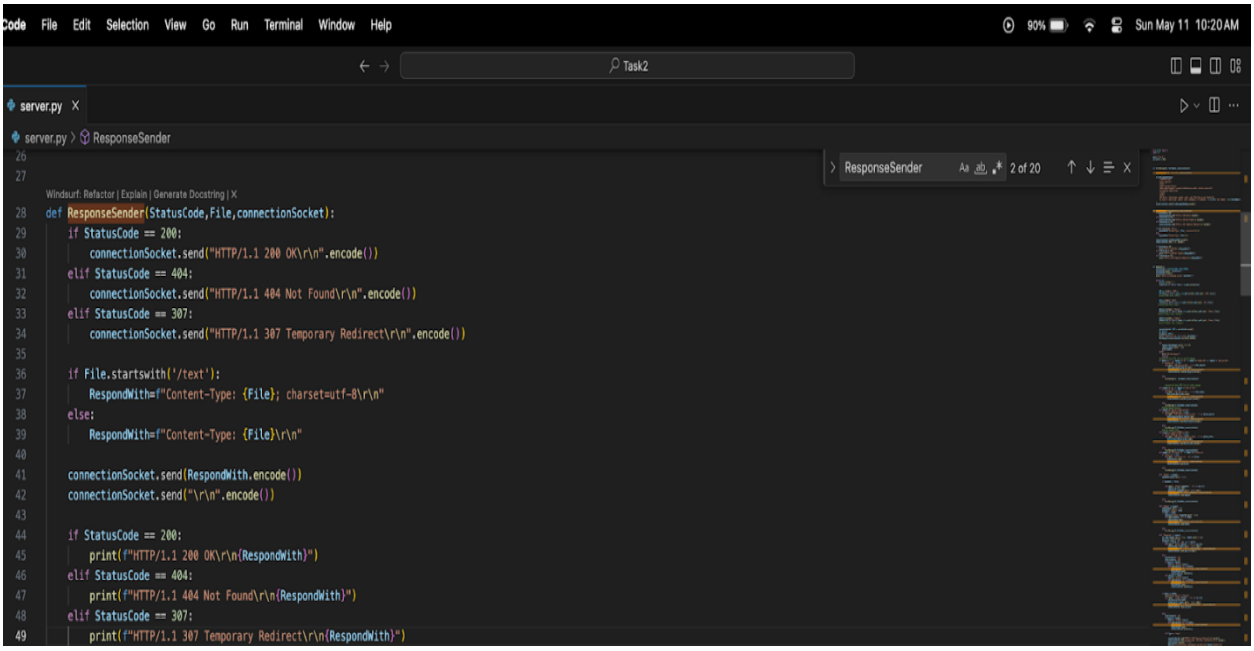


Figure 34: 404 Error Page Displayed When Requested File is Not Found on Localhost

Moving onto the second function we parsed the status code, the type of file, and the connection socket (where to send
response).



HTTP Response Handling and Header Construction

In a web server, the **response varies based on the HTTP status code**:

- **200 OK** – Indicates a **successful** connection.
- **404 Not Found** – Means the **requested resource was not found**.
- **307 Temporary Redirect** – Indicates the resource is **temporarily redirected** to another location.

Header for Text-Based Responses

If the **requested file path starts with /text**, the following **HTTP header** should be included in the response:

Content-Type: text/html; charset=utf-8\r\n

Sample Response Format

The function constructs and sends an HTTP response similar to the following:

HTTP/1.1 200 OK\r\n

Content-Type: text/html; charset=utf-8\r\n

\r\n

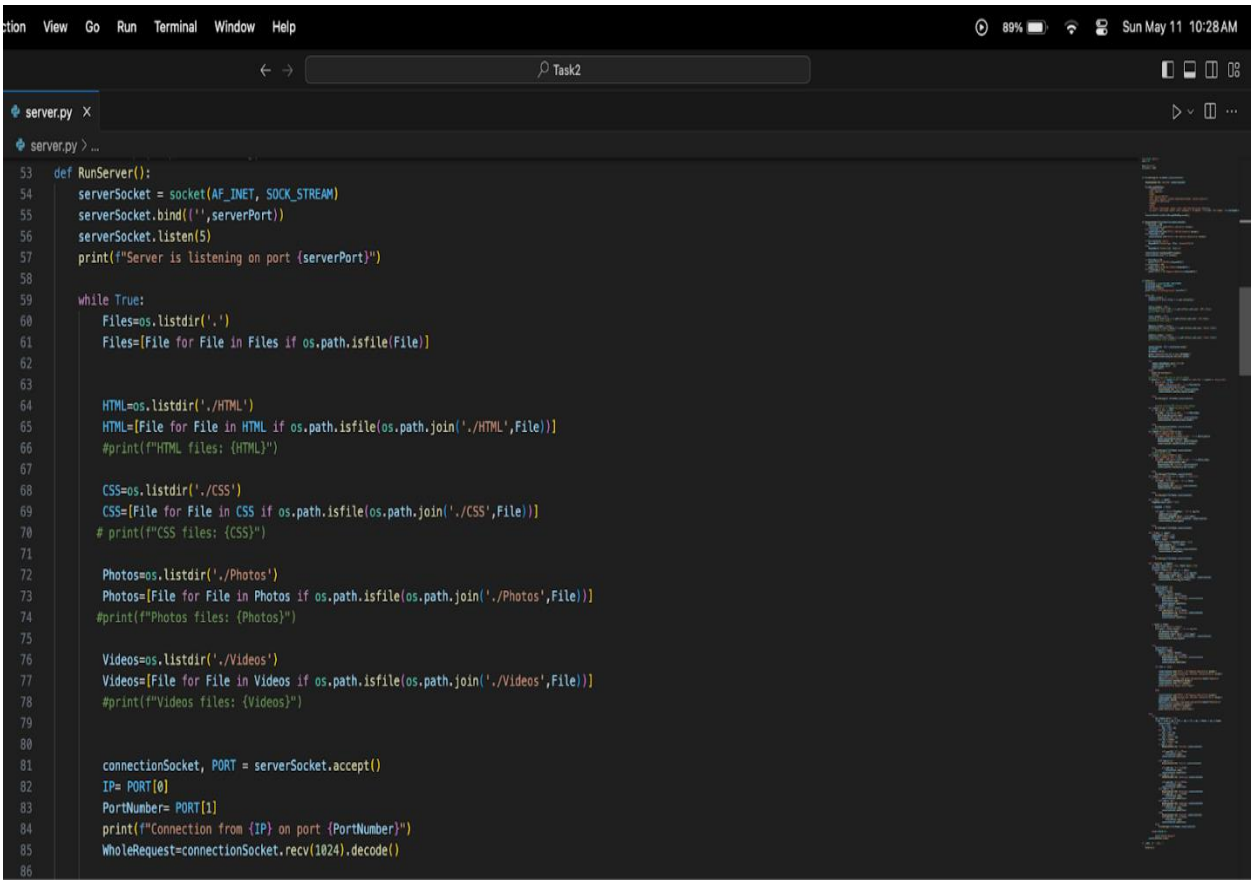
This response is also **printed to the terminal** for **tracing and debugging purposes**.

RunServer() Function

The final function, **RunServer()**, is responsible for:

- **Starting** the server
- **Binding** it to port **9903**
- **Listening** for incoming client connections
- **Accepting** connections and **handling requests**
- **Generating and sending appropriate responses**

It integrates all previous components (socket setup, request handling, response generation) to run the server **continuously** and **serve client requests**.



Server Initialization and Connection Handling

1. Creating and Binding the Socket

- A **TCP socket** is created using the **AF_INET** address family, which supports **IPv4**.
- The socket is **bound** to a specific **server port** to start accepting connections.
- The server is configured to **accept connections from any client IP address**.

---

2. Listening for Connections

- The server begins **listening** for incoming connections using:

socket.listen(5)

- This sets the **maximum queue size** for **pending connections** to **5**.

---

3. Main Server Loop

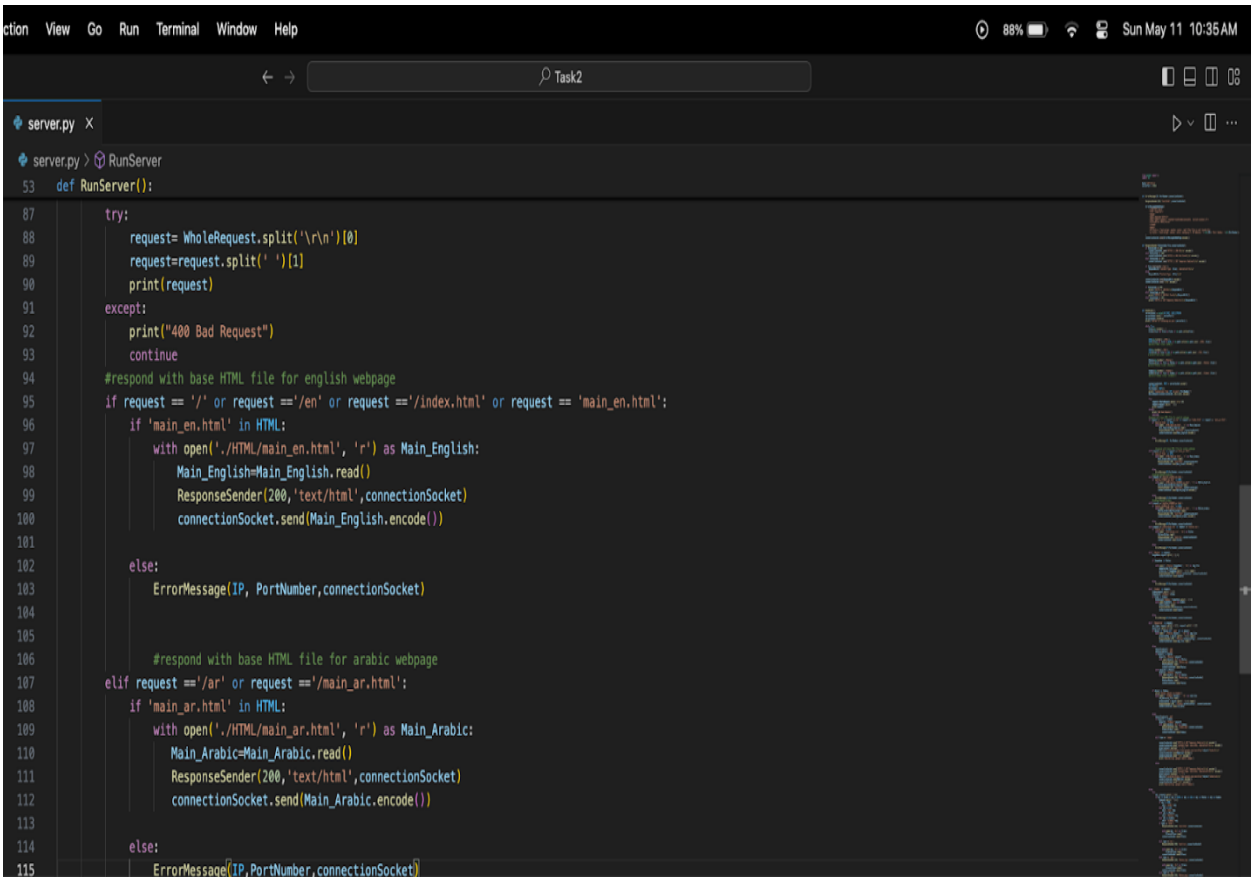
- The server enters an **infinite loop (while True)** to continuously handle client requests.
- In each iteration:
    - It **lists all available files** from the following directories:
        - **Files**
        - **HTML**
        - **CSS**
        - **Photos**
        - **Videos**

---

4. Accepting and Receiving Client Data

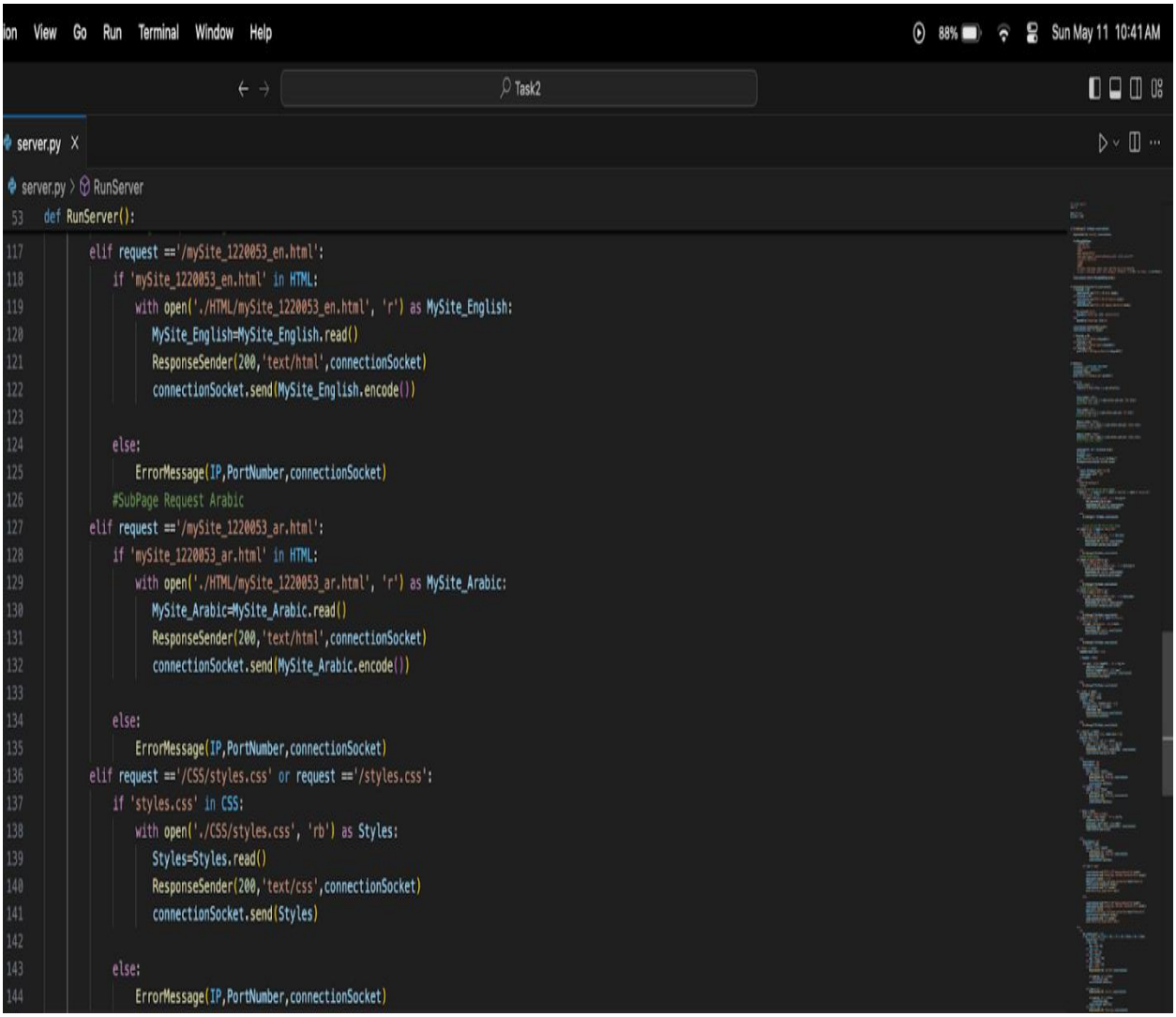
- At **line 81** (in the code), the server:
    - **Accepts** a new client connection.
    - **Retrieves** the client's **IP address** and **port number**.

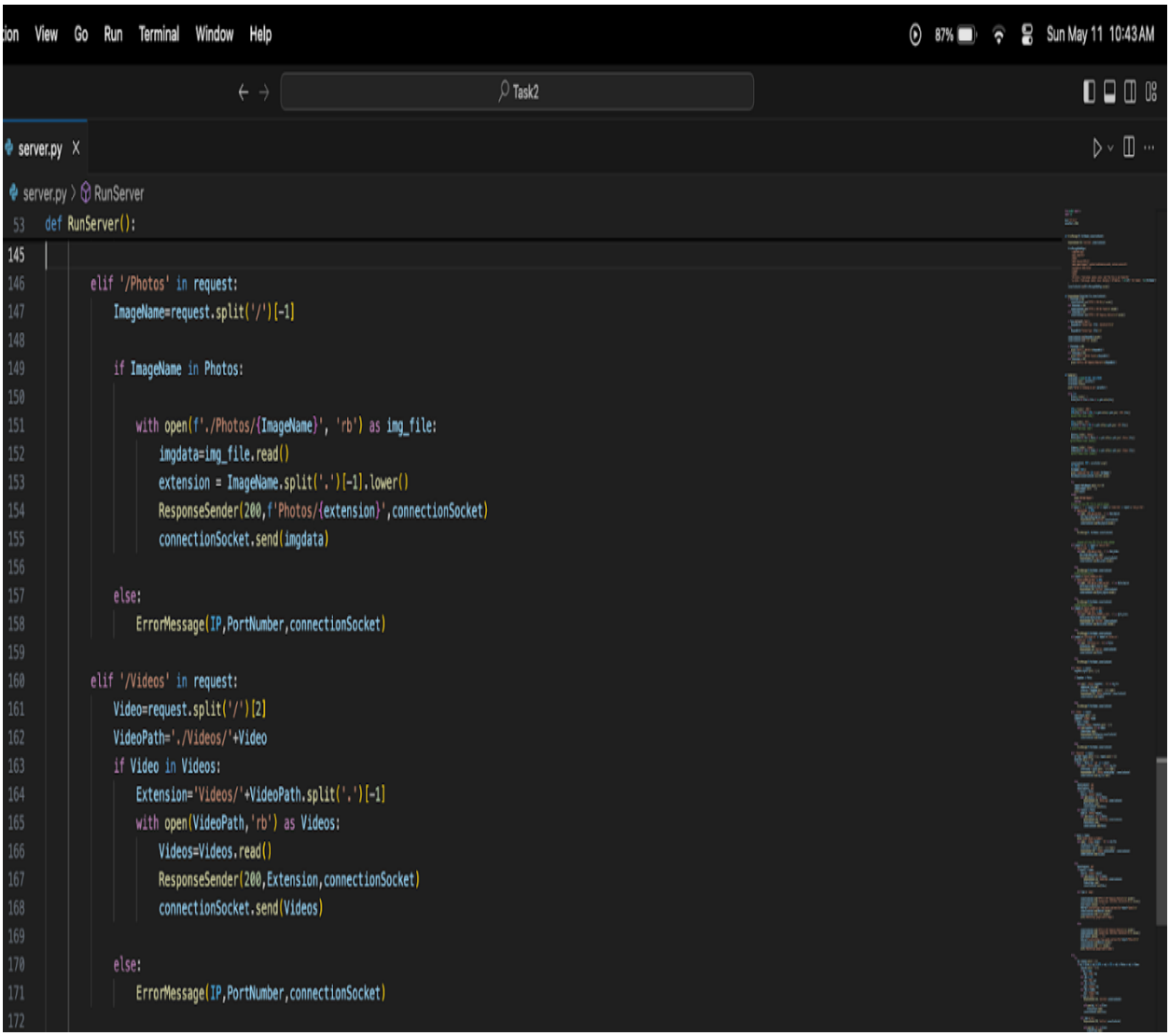
- It then:
    - **Receives** up to **1024 bytes** of data from the HTTP request.
    - **Decodes** the data from **bytes to string** format for further processing.



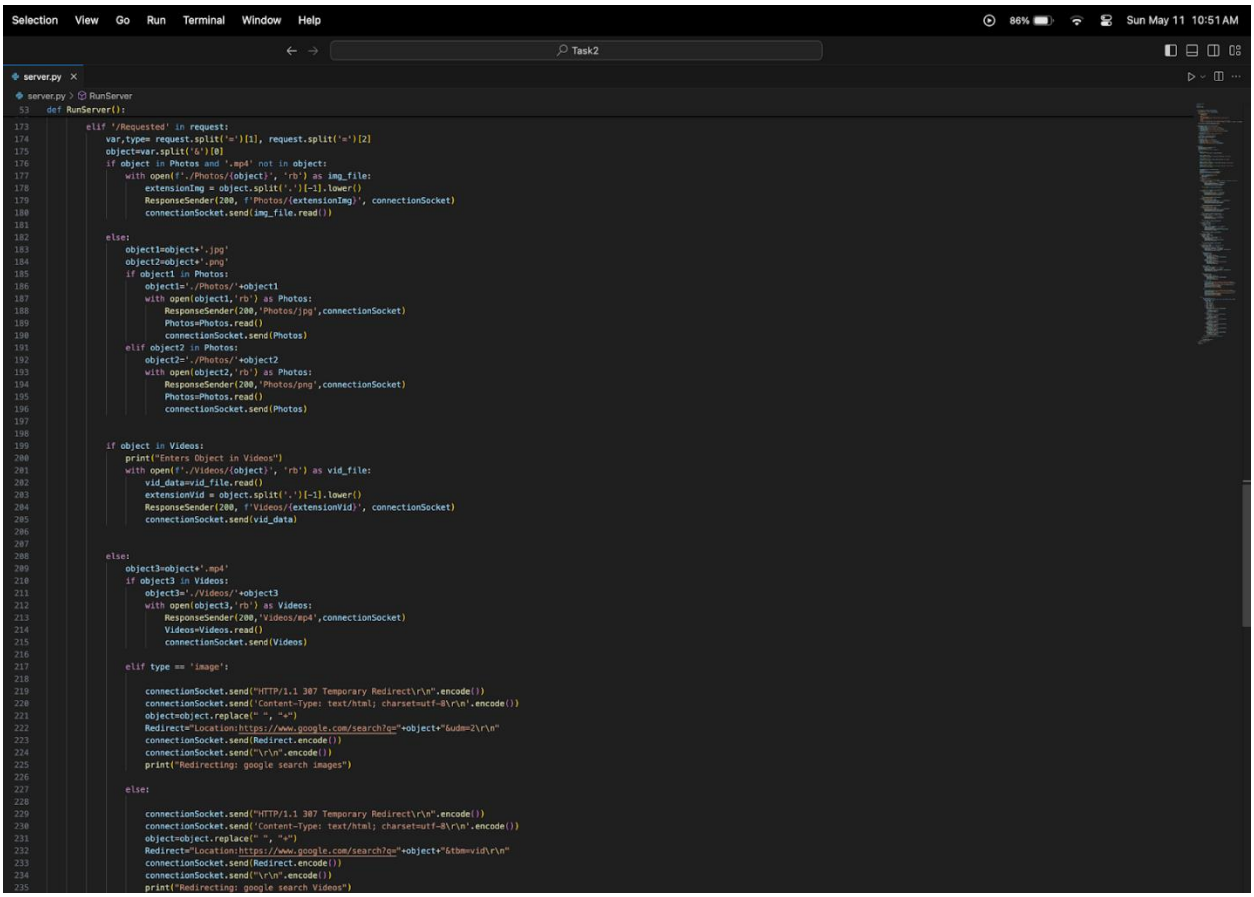
Next, the server **parses** the client's request by **splitting the request line** after the **server port** to determine the **requested path**. If the path is /, /en, /index.html, or /main_en.html, the server responds by sending the **main_en.html** file as the **base HTML page** and proceeds to load all **linked resources** within it, such as **CSS**, **images**, or **scripts**. On the other hand, the **Arabic version** of the webpage is accessible **only** through /ar or /main_ar.html, in which case the serverserves the **main_ar.html** file as the response.

The way we call the subpages is the same format:

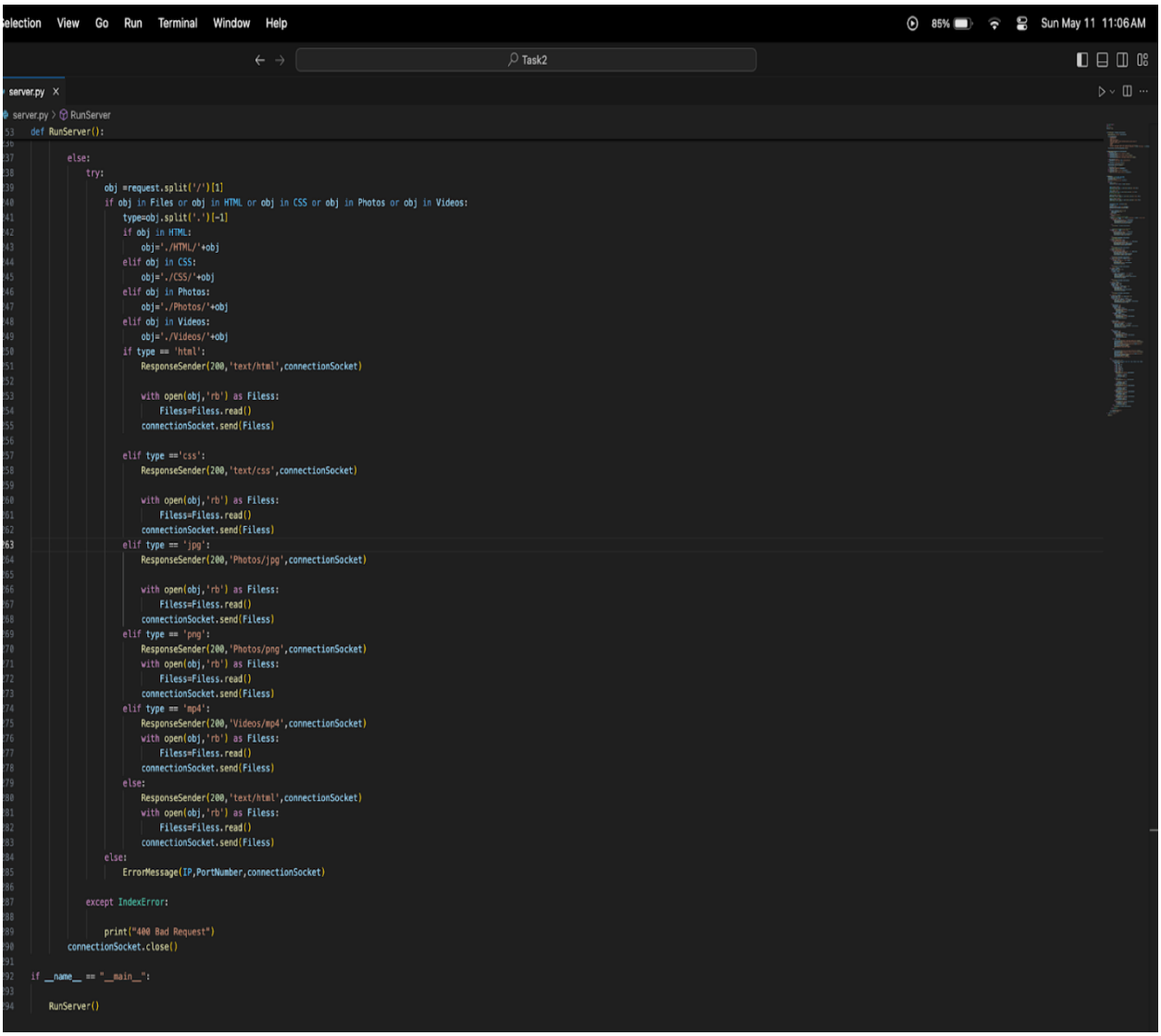



ion View Go Run Terminal Window Help
87%
Sun May 11 10:43 AM
Task2
server.py X
server.py > RunServer
53 def RunServer():
145
146 elif '/Photos' in request:
147 ImageName=request.split('/')[-1]
148
149 if ImageName in Photos:
150
151 with open(f'./Photos/{ImageName}', 'rb') as img_file:
152 imgdata=img_file.read()
153 extension = ImageName.split('.')[-1].lower()
154 ResponseSender(200,f'Photos/{extension}',connectionSocket)
155 connectionSocket.send(imgdata)
156
157 else:
158 ErrorMessage(IP,PortNumber,connectionSocket)
159
160 elif '/Videos' in request:
161 Video=request.split('/')[2]
162 VideoPath='./Videos/'+Video
163 if Video in Videos:
164 Extension='Videos/'+VideoPath.split('.')[-1]
165 with open(VideoPath,'rb') as Videos:
166 Videos=Videos.read()
167 ResponseSender(200,Extension,connectionSocket)
168 connectionSocket.send(Videos)
169
170 else:
171 ErrorMessage(IP,PortNumber,connectionSocket)
172

Because our HTML file contains images and videos we have to handle them. First we see what is in the request, is it a Photo or a Video. If it's a photo we open the folder and “read binary”, we read the image adding its extension we want then send it as a response. Same goes for the Videos.



As for the **image above**, it illustrates how the server handles **photo and video requests** from a **subpage**. First, the server **splits the request** and extracts the **file name (object)** in the form of "example.extension". If the **object** is located in the **Photos** folder and does **not** have a .mp4 extension (a constraint added for extra security), the server **opens the Photos folder** and allows the client to **download the image**. If the object is missing its extension, the server **adds the appropriate extension** and **searches again**. The **same logic** applies to the **Videos** folder for handling video files. If the **requested image or video** is **not found** in either folder, the server **redirects the user** to a **Google search page** using their **query** as the search keyword.



As a **last resort** or **backup mechanism**, the server **splits the object** again to determine the **type of request** by examining its **file extension**. It then attempts to **traverse all possible file types**, including **.html**, **.css**, **.png**, **.jpg**, and **.mp4**, to locate the **requested resource**. If none of these attempts succeed and an **IndexError** or similar issue occurs, the server **calls the initial handler function**, **prints a "Bad Request"**, and then **closes the connection**. Importantly, the **connection is closed after each request** to ensure **clean** and **isolated communication cycles**.

❖ Requests and Response Testing

We saw the not found request from before now lets see that happens when we request an image that exists:

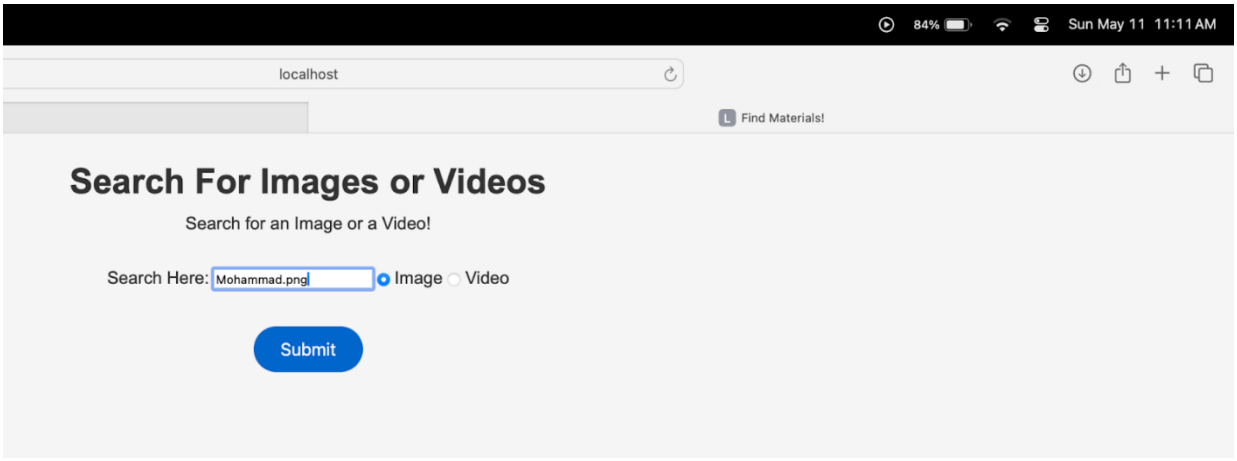


Figure 35: Localhost Media Search Interface

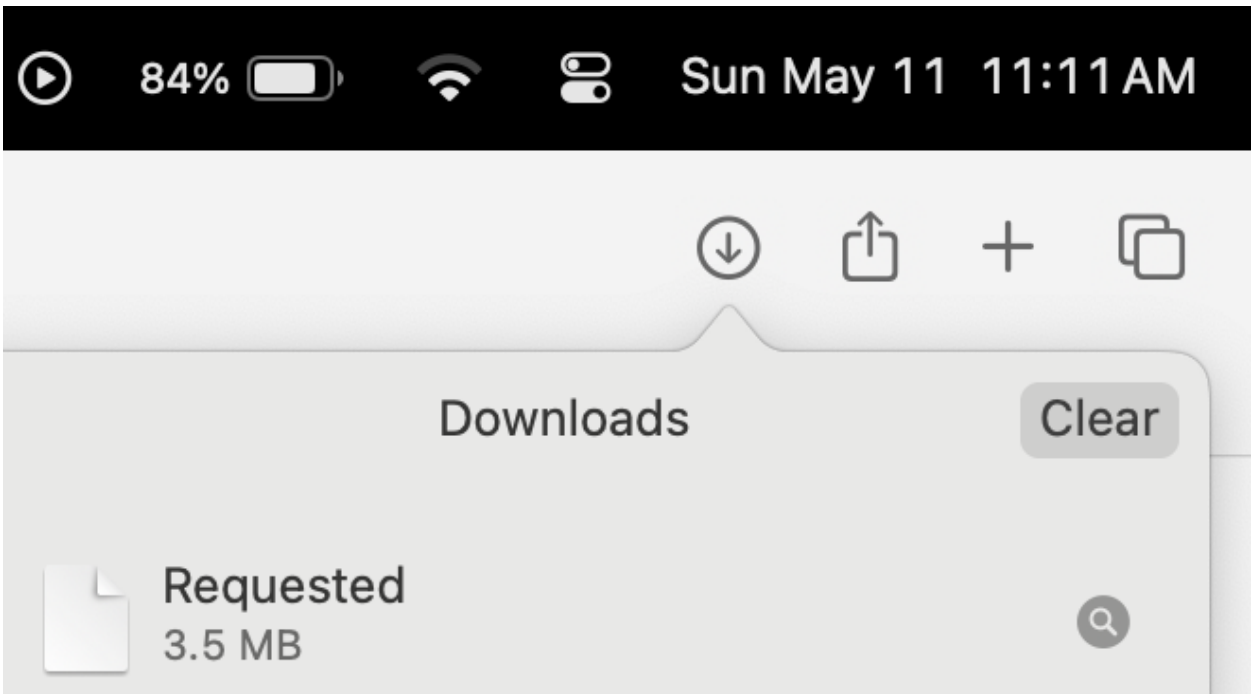


Figure 36 : Downloaded File Displayed in Safari's Downloads Panel

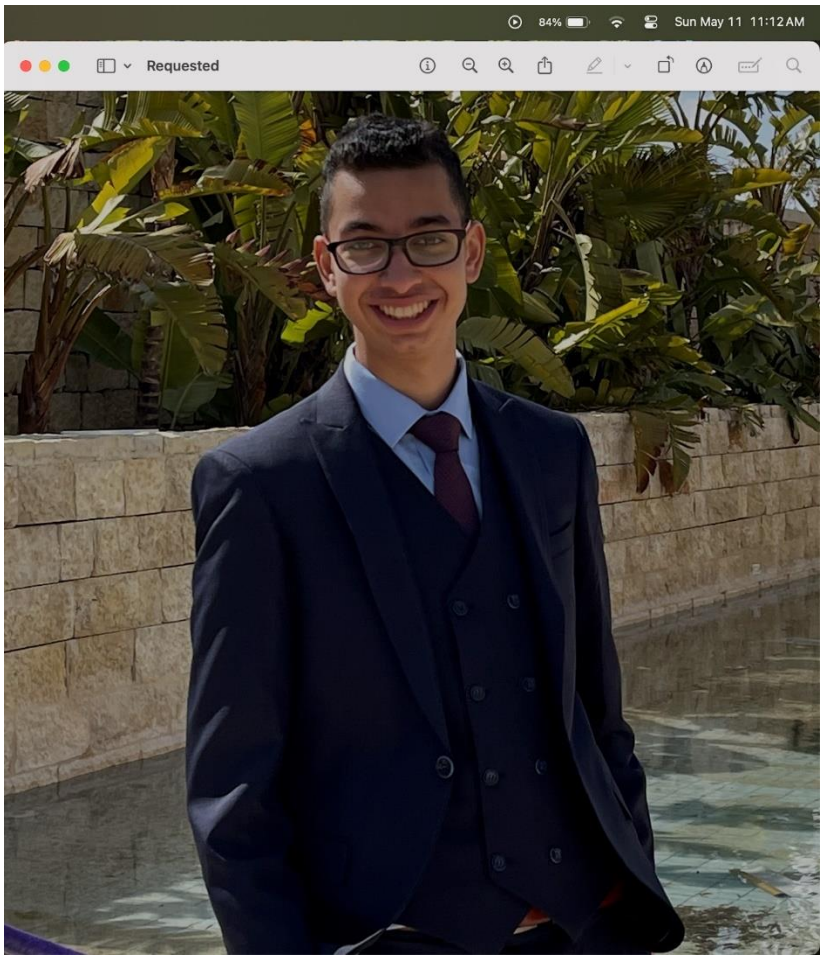


Figure 37: Formal Portrait in an Outdoor Setting

When the image doesnt exist:

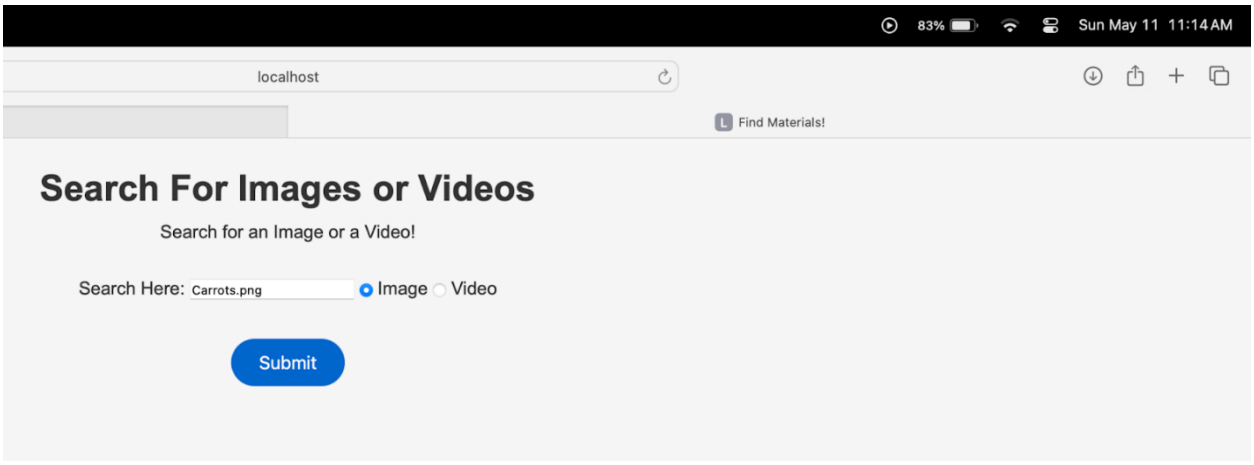


Figure 38: Search for not exist image.

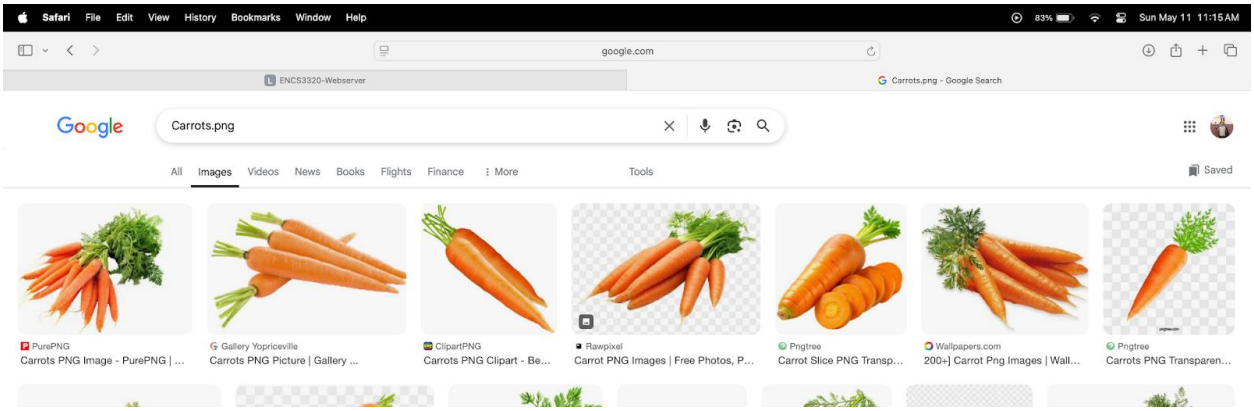


Figure 39: No Matching Image Found for "Carrots.png

With URL:https://www.google.com/search?q=Carrots.png&udm=2

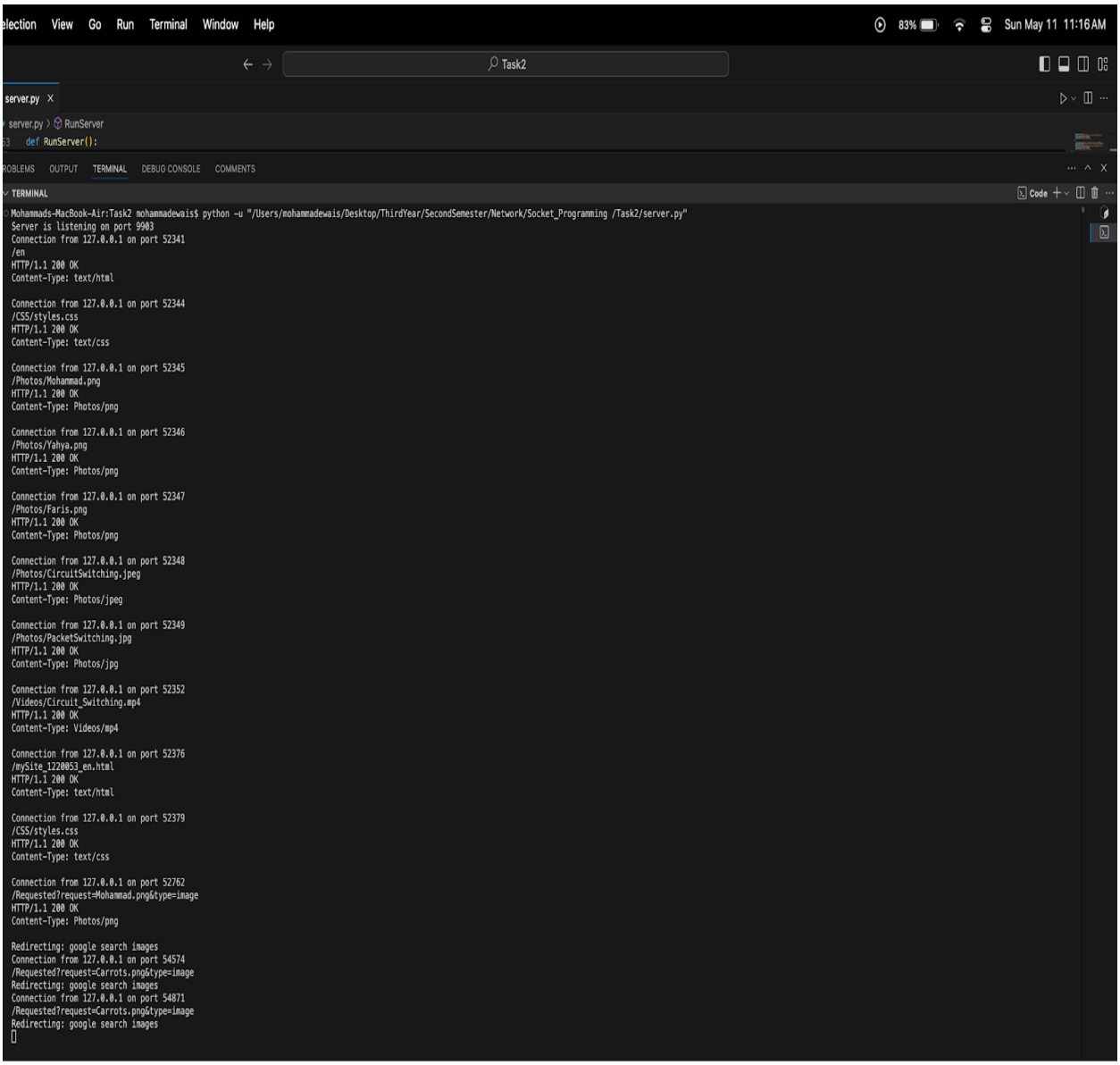


Figure 40: Terminal Output of Python HTTP Server Handling Image and Video Requests

This picture shows each request and connection, from the base HTML file, to the images and redirections of the images to google.

As for the videos:

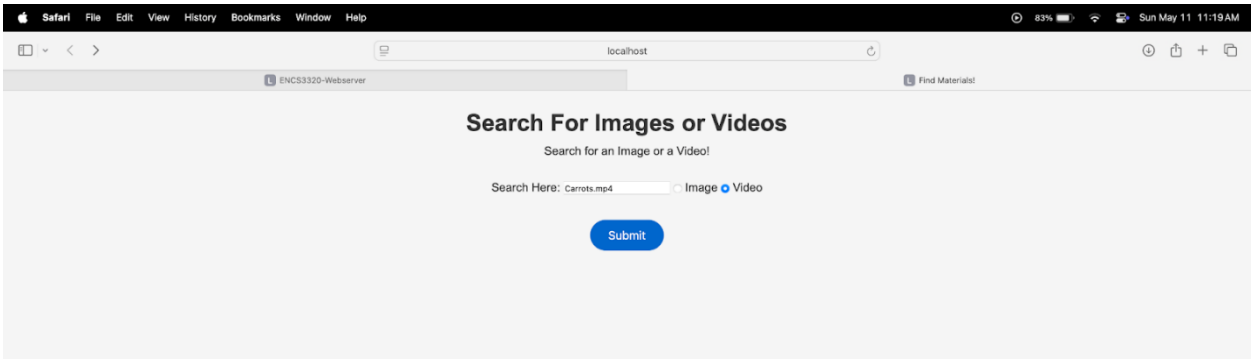


Figure 41: Subpage video redirect input

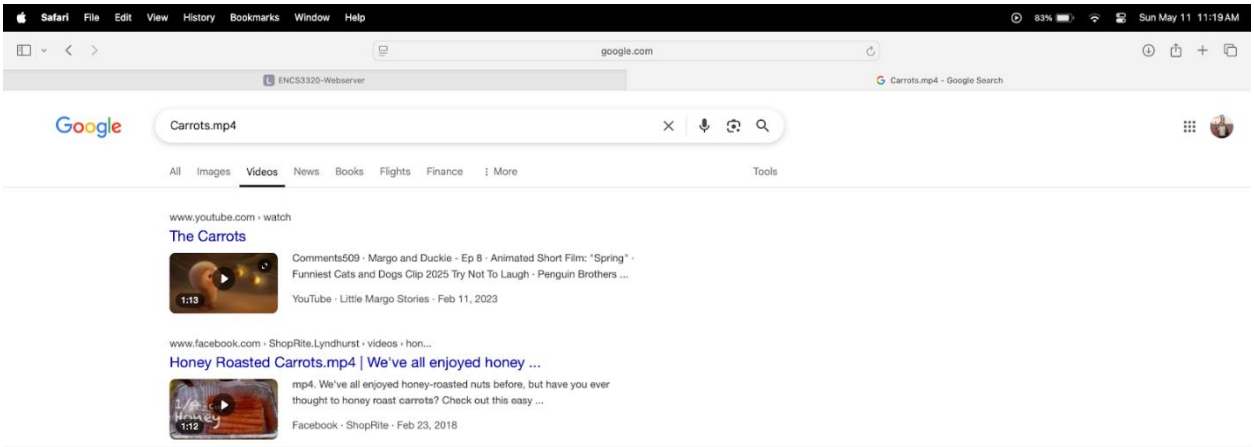


Figure 42: Redirected Response

With URL: https://www.google.com/search?q=Carrots.mp4&udm=7

When it exists:

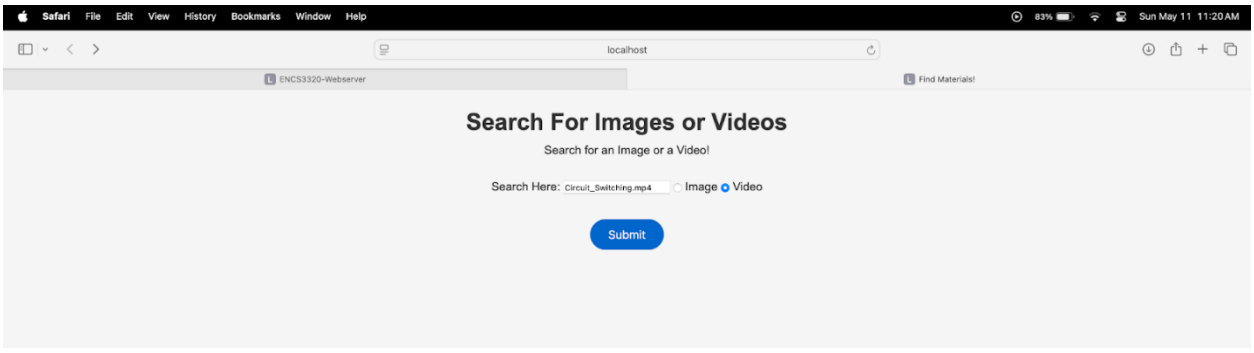


Figure 43: Another subpage video redirect input

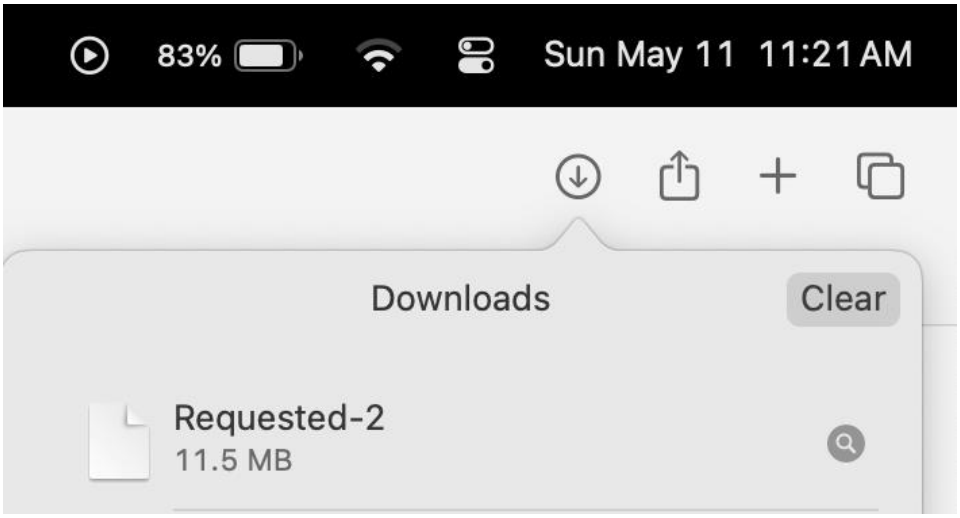


Figure 44: Downloaded response

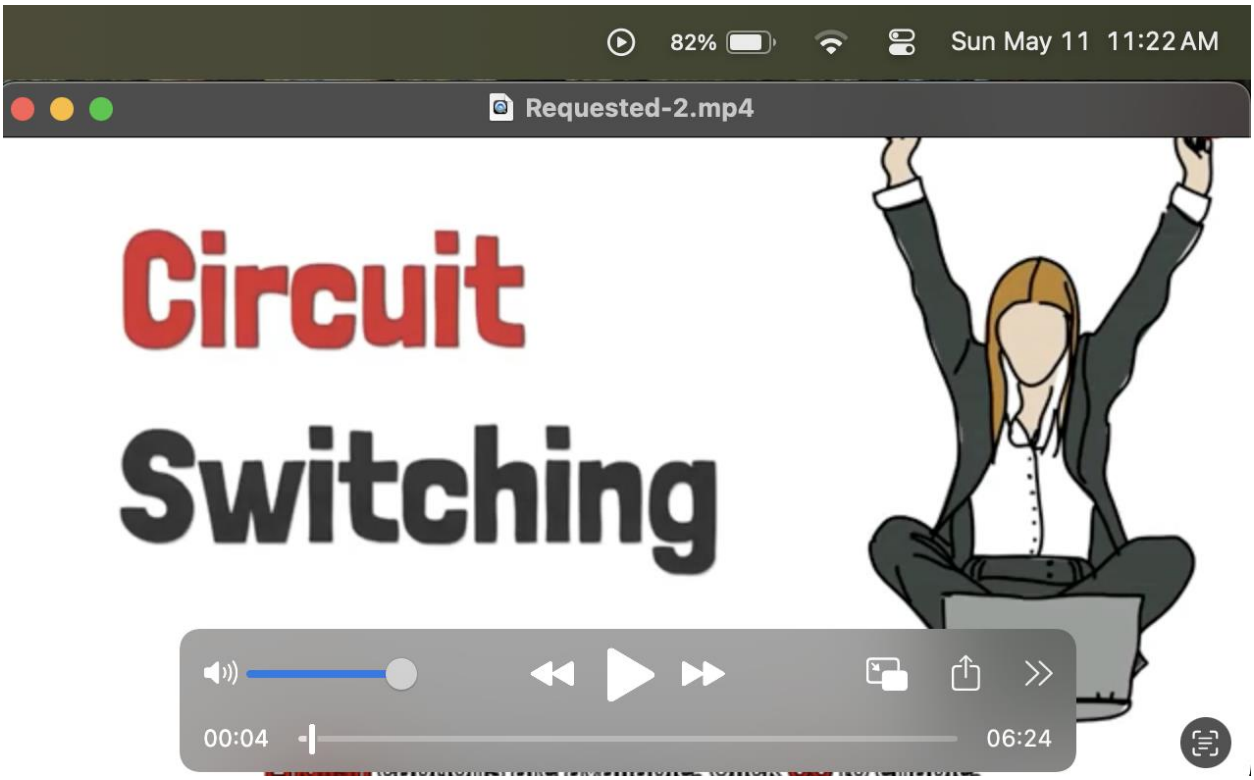


Figure 45: Subpage opened Request -2

In the terminal:

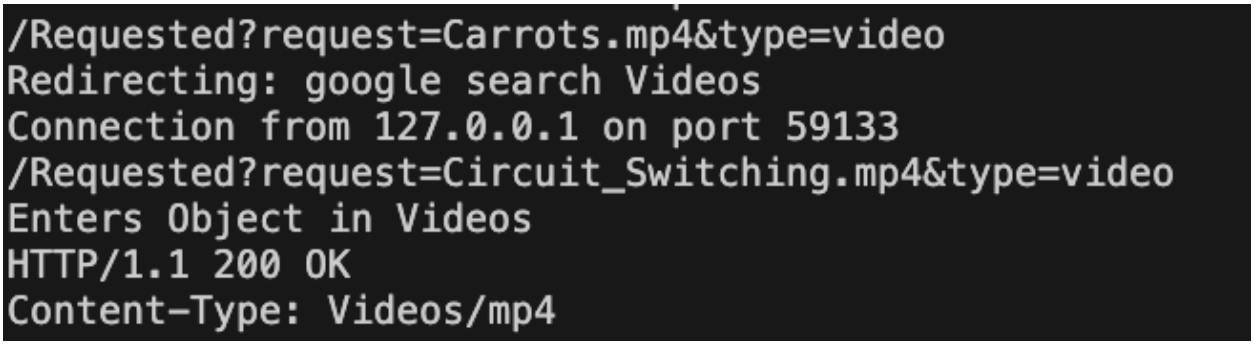


Figure 46: Terminal response to videos

For Part E we do everything, as you can call the website by different names and still return to the same one, and we also handle the error page as well.

3) Task 3 – Web Server

❖ Introduction for this task

The goal of this project was to create a multiplayer number-guessing game that made use of both the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) protocols. Utilizing the advantages of both protocols was the primary goal, with UDP being used for real-time communication during games (for quick and effective feedback sharing) and TCP managing reliable connections (for player registration, game initiation, etc.).

The client-server design of the game allows several players (clients) to connect to the server to play in competition with one another. Game logic, including player registration, game state broadcasting, and game flow management, is handled by the server. Through UDP, clients communicate with the server, sending predictions and getting feedback on whether they are accurate or too high or low.

The following essential actions are involved in the game:

1. Player Registration: After creating a connection with the server, clients provide their login and watch for the game to begin.
2. Game Start: The server starts the game and transmits the target number range (1–100) to every player after the minimum number of players (for example, 2) have joined.
3. The objective number is guessed by each player in turn. Each guess is processed by the server, which uses UDP for real-time communication to provide feedback such as "Higher," "Lower," or "Correct".
4. Game End: The game is over when the player guesses the target number accurately or when the time limit has passed.

We can guarantee stability and efficiency by combining UDP for fast-paced action with TCP for dependable connections. UDP guarantees that player guesses and feedback are sent fast and without delay, which is essential in a time-sensitive game, while TCP guarantees reliable and safe game setup and communication between the server and clients.

Through this project, we can look into and put into practice both TCP and UDP protocols in a practical setting, highlighting their advantages and disadvantages. Ensuring seamless gaming and effective communication between the server and clients also required the capacity to use multithreading to manage several players at once.

❖ Server Implementation & Code

1. Initialization & Configuration

```
# Server configuration
TCP_PORT = 6000
UDP_PORT = 6001
HOST = 'localhost'
TOTAL_GAME_TIME = 60  # Game duration in seconds
GUESSES_PER_PLAYER = 6  # Each player gets 6 guesses
GUESS_COOLDOWN = 10  # 10 seconds between guesses
```

- **Purpose**: Sets up game parameters and network ports.
- **Key Features**:
    - TCP for reliable connection setup/teardown.
    - UDP for low-latency guess submissions.
    - Game constraints (time, guesses, cooldown).

2. Player Management

```
clients = {}  # Dictionary to store player names (keys) and their TCP sockets (values)
addresses = {}  # Dictionary to store player names (keys) and their addresses (values)
target_number = random.randint(1, 100)  # The target number players need to guess
game_started = False  # Game started flag
game_over = False  # Game over flag

# Track player guesses
player_guesses = {}  # {player_name: [list_of_guesses]}
last_guess_time = {}  # {player_name: last_guess_timestamp}
```

- clients: Active TCP connections.
- addresses: IP/port mapping for UDP replies.
- player_guesses: Tracks guess history.
- last_guess_time: Enforces cooldowns.
- game_over: Flag to end the game

3. **Message Broadcasting and Player Disconnection Management**

```python
def send_message_to_all_players(msg: str):
    """Send a message to all connected players and handle disconnections."""
    disconnected_players = []
    for name, conn in clients.items():
        try:
            conn.sendall(msg.encode())  # Send message to player
        except:
            disconnected_players.append(name)  # Track disconnected players

    # Handle disconnections
    for name in disconnected_players:
        handle_player_disconnect(name)
```

This function, send_message_to_all_players, sends a message to all connected players. It first attempts to send the message to each player. If a player is disconnected, their name is added to a list of disconnected players. After the message is sent, the function checks the list of disconnected players and calls another function to handle their disconnections. This ensures that disconnected players are properly managed.

4. Handling Player Disconnections and Managing Game Continuation

This function, handle_player_disconnect, manages player disconnections during the game. It removes the disconnected player from the game and handles the game state accordingly. If no players remain, the game ends. If one player remains, it asks whether they want to continue alone. If they agree, they proceed; if not, the game ends.

```python
def handle_player_disconnect(name):
    """Handle player disconnections during the game."""
    global game_started, game_over
    with lock:
        if name in clients:
            print(f"[DISCONNECT] {name} disconnected mid-game.")
            del clients[name]  # Remove the player from the dictionary
            del addresses[name]  # Remove the player address
            if name in player_guesses:
                del player_guesses[name]
            if name in last_guess_time:
                del last_guess_time[name]

    # If no players remain
    if len(clients) < 1:
        game_over = True
        print("[GAME] All players disconnected. Game ended.")
        return

    # If only one player remains and game was started
    if len(clients) == 1 and game_started and not game_over:
        remaining_player = next(iter(clients))
        conn = clients[remaining_player]

        try:
            # Ask the remaining player if they want to continue alone
            conn.sendall(b"PLAYER_LEFT: " + name.encode() + b" left the game. Do you want to play alone? (Y/N)\n")
            conn.settimeout(30.0)  # Give them 30 seconds to respond
            response = conn.recv(1024).decode().strip().upper()

            if response == 'Y':
                print(f"[GAME] {remaining_player} chose to continue alone")
                conn.sendall(b"CONTINUE: Game continues with you alone. You have 60 seconds to guess!\n")
            else:
                print(f"[GAME] {remaining_player} chose to end the game")
                conn.sendall(b"GAME_ENDED: Game ended by your choice. No winner.\n")
                game_over = True
        except:
            print(f"[GAME] {remaining_player} didn't respond, ending game")
            game_over = True
```

5. **Checking Player's Guess Eligibility**

The function can_player_guess checks if a player can make a guess based on two conditions: the number of guesses they've made and the cooldown time. It first checks if the player has made any guesses, then verifies if the player has reached the maximum allowed guesses (GUESSES_PER_PLAYER). Finally, it ensures that the player has waited long enough since their last guess based on the defined cooldown time (GUESS_COOLDOWN).

```
def can_player_guess(player_name):
    """Check if player can make a guess (cooldown and guess count)."""
    if player_name not in player_guesses:
        return True  # First guess

    # Check guess count
    if len(player_guesses[player_name]) >= GUESSES_PER_PLAYER:
        return False

    # Check cooldown
    current_time = time.time()
    last_time = last_guess_time.get(player_name, 0)
    return (current_time - last_time) >= GUESS_COOLDOWN
```

6. Managing Game Round, Guessing Logic, and Player Interactions

```python
def handle_game_round():
    """Manages the game round with cooldowns and guess limits."""
    global game_over
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_socket.bind((HOST, UDP_PORT))

    start_time = time.time()  # Track the game start time

    while not game_over:
        # Calculate remaining time
        elapsed_time = time.time() - start_time
        remaining_time = TOTAL_GAME_TIME - elapsed_time

        if remaining_time <= 0:
            send_message_to_all_players("TIME_UP: Time's up! No winners.\n")
            print(f"[!] Game ended. Time's up! The target number was {target_number}.")
            game_over = True
            break

        # Check if all players have used all their guesses
        all_guesses_used = True
        for player in clients:
            if player not in player_guesses or len(player_guesses[player]) < GUESSES_PER_PLAYER:
                all_guesses_used = False
                break

        if all_guesses_used:
            send_message_to_all_players("GUESSES_UP: All guesses used! No winners.\n")
            print(f"[!] Game ended. All guesses used! The target number was {target_number}.")
            game_over = True
            break

        # Process any incoming guesses
        try:
            udp_socket.settimeout(1)  # Short timeout to check game state frequently
            data, addr = udp_socket.recvfrom(1024)

            try:
                name, guess_str = data.decode().split(':')
                guess = int(guess_str)

                # Validate player can guess
                if not can_player_guess(name):
                    udp_socket.sendto(b"Error: You must wait 10 seconds between guesses", addr)
```

This function, handle_game_round, manages the entire game round. It first tracks the time, calculating the remaining time for the round. If time runs out or if all players have made their guesses, the game ends. The function also processes incoming guesses from players, ensuring they follow the cooldown period between guesses and that their guesses fall within the valid range (1-100). It compares each player's guess with the target number and gives feedback (higher or lower) or announces the winner if the guess is correct.

```
                    udp_socket.sendto(b"Error: You must wait 10 seconds between guesses", add
                    continue

                # Track the guess
                if name not in player_guesses:
                    player_guesses[name] = []
                player_guesses[name].append(guess)
                last_guess_time[name] = time.time()

                # Validate guess range
                if guess < 1 or guess > 100:
                    udp_socket.sendto(b"Error: Guess must be between 1-100", addr)
                    continue

                # Check guess against target
                if guess < target_number:
                    udp_socket.sendto(b"Higher", addr)
                    send_message_to_all_players(f"GUESS:{name} guessed {guess} (Higher)\n")
                elif guess > target_number:
                    udp_socket.sendto(b"Lower", addr)
                    send_message_to_all_players(f"GUESS:{name} guessed {guess} (Lower)\n")
                else:
                    # Correct guess
                    print(f"\n[!] {name} won! The target number was {target_number}.")
                    udp_socket.sendto(b"Correct", addr)
                    result_message = f"[RESULT] {name} guessed the number {target_number}! Ga
                    send_message_to_all_players(result_message + "\n")
                    game_over = True
                    break

            except (ValueError, IndexError):
                udp_socket.sendto(b"Error: Invalid input! Only numbers are allowed.", addr)

        except socket.timeout:
            continue  # Continue game loop if no data received

    udp_socket.close()
```

7. Handling New TCP Connections and Game Initialization

The function handle_new_tcp_connection manages new players connecting via TCP. It ensures that the player's name is unique and valid, and checks if the game is already full or not. If the name is invalid or the game has reached the maximum number of players, the connection is rejected. If the conditions are met, the player is added to the game, and once there are enough players, the game starts. A separate thread is created to handle the game round once the game is initialized.

```
def handle_new_tcp_connection(conn, addr):
    """Handles new players joining via TCP."""
    global game_started, game_over
    try:
        name = conn.recv(1024).decode().strip()
    except:
        conn.close()
        return

    with lock:
        # Ensure the name is unique and valid
        if name in clients or name.strip() == "":
            conn.sendall(b"NAME_ERROR: Name already taken or invalid. Please reconnect with a different name.")
            conn.close()
            print(f"[REJECTED] Duplicate name '{name}' tried to join.")
            return

        if len(clients) >= 4:  # Reject players if the game is full
            conn.sendall(b"Game is full. Cannot join.")
            conn.close()
            print(f"[REJECTED] {name} tried to join (max players reached).")
            return

        clients[name] = conn
        addresses[name] = addr
        player_guesses[name] = []

    print(f"[TCP] {name} joined from {addr}")

    # Start game when minimum players (2) have joined
    with lock:
        if len(clients) >= 2 and not game_started and not game_over:
            game_started = True
            send_message_to_all_players(f"GAME_START: {','.join(clients.keys())}\n")
            send_message_to_all_players(f"Game started! You each have {GUESSES_PER_PLAYER} guesses with {GUESS_COOLDOWN}s cooldown.\n")
            threading.Thread(target=handle_game_round, daemon=True).start()
```

8. Starting the Server and Handling Player Connections

```
def start_server():
    """Starts the server and listens for incoming player connections."""
    tcp_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    tcp_sock.bind((HOST, TCP_PORT))
    tcp_sock.listen(5)
    print(f"[SERVER] Listening on {HOST}:TCP {TCP_PORT},UDP {UDP_PORT}")
    print(f"[DEBUG] Target number: {target_number}")

    while True:
        conn, addr = tcp_sock.accept()
        threading.Thread(target=handle_new_tcp_connection, args=(conn, addr), daemon=True).start()
```

The start_server function sets up the server to listen for incoming player connections. It creates a TCP socket, binds it to a specific host and port, and listens for incoming connections. When a new player connects, it accepts the connection and starts a new thread to handle the connection using the handle_new_tcp_connection function. The server continues to listen for more connections in an infinite loop.

❖ Client Implementation & Code

1. Client Configuration and Network Setup

```
# Client configuration
TCP_PORT = 6000
UDP_PORT = 6001
SERVER = 'localhost'

name = input("Enter your name: ")
game_over = False

# TCP connection setup
tcp_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcp_sock.connect((SERVER, TCP_PORT))
tcp_sock.sendall(name.encode())

# UDP connection setup
udp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_sock.bind(('0.0.0.0', 0))
```

This code snippet handles the client-side configuration and network setup. It asks the player to input their name and initializes the game state as not over (game_over = False). It then sets up two types of connections:

1. **TCP Connection:** A TCP socket is created and connected to the server at the specified SERVER and TCP_PORT.
2. **UDP Connection:** A UDP socket is created and bound to the local address (0.0.0.0), allowing it to listen for UDP messages. The player's name is sent via the TCP connection to the server.

2. **Listening for and Handling Incoming TCP Messages**

The listen_for_tcp_messages function listens for incoming messages from the server. It continuously receives data, processes each line, and performs specific actions based on the message type:

- **GAME_START:** Informs players that the game has started and provides game rules.
- **GUESS:** Processes player guesses.
- **PLAYER_LEFT:** Notifies when a player leaves the game.
- **CONTINUE:** Handles whether a remaining player wishes to continue playing alone.
- **GAME_ENDED:** Announces the end of the game with the winner or if no one guessed correctly.
- **RESULT:** Displays the results, such as who won the game and the correct number.
- **TIME_UP:** Handles scenarios where the time runs out without a winner.
- **GUESS_UP:** Handles the situation when all guesses have been used with no winner.
- **GAME_COMPLETED:** Ends the game if no winners are determined.

In all cases, it updates the game state and checks if the game is over based on these messages. If invalid data is received, an error message is printed.

```python
def listen_for_tcp_messages():
    global game_over
    buffer = ""
    while not game_over:
        data = tcp_sock.recv(1024).decode()
        if not data:
            break
        buffer += data
        while '\n' in buffer:
            line, buffer = buffer.split('\n', 1)
            if line.startswith("GAME_START:"):
                players = line.split(":",1)[1].split(',')
                print("\n=== GAME STARTED ===")
                print("Players:", ", ".join(players))
                print(f"You have 6 guesses with 10s cooldown between guesses.")
            elif line.startswith("GUESS:"):
                print(f"\n{line.replace('GUESS:', '')}")
            elif line.startswith("PLAYER_LEFT:"):
                left_player = line.split(":", 1)[1]
                print(f"\nPlayer {left_player} left the game.")
                response = input("Do you want to play alone? (Y/N): ").upper()
                tcp_sock.sendall(response.encode())
            elif line.startswith("CONTINUE:"):
                print("\n" + line.split(":", 1)[1])
            elif line.startswith("GAME_ENDED:"):
                print("\n" + line.split(":", 1)[1])
                game_over = True
                break
            elif line.startswith("[RESULT]"):
                try:
                    result_text = line.replace("[RESULT] ", "")
                    winner_part = result_text.split(" guessed the number ")[0]
                    number_part = result_text.split(" guessed the number ")[1].split("!")[0]

                    print("\n" + "=" * 40)
                    print("===== GAME ENDED =====")
                    print(f"🎉 Winner: {winner_part} 🎉")
                    print(f"🔢 Correct Number: {number_part}")
                    print("=" * 40 + "\n")
                    game_over = True
                except IndexError:
```

```
                except IndexError:
                    print("\n[ERROR] Invalid result format received.")
                break
            elif line.startswith("TIME_UP:"):
                print("\n" + "=" * 40)
                print("===== GAME ENDED =====")
                print("⏰ Time's up! No winners.")
                print("=" * 40 + "\n")
                game_over = True
                break
            elif line.startswith("GUESSES_UP:"):
                print("\n" + "=" * 40)
                print("===== GAME ENDED =====")
                print("🔢 All guesses used! No winners.")
                print("=" * 40 + "\n")
                game_over = True
                break
            elif line.startswith("Game completed."):
                if "No winners" in line:
                    print("\n" + "=" * 40)
                    print("===== GAME ENDED =====")
                    print("⏰ Game completed. No winners.")
                    print("=" * 40 + "\n")
                else:
                    winner = line.split("Winner: ")[1].strip()
                    print("\n" + "=" * 40)
                    print("===== GAME ENDED =====")
                    print(f"🎉 Winner: {winner} 🎉")
                    print("=" * 40 + "\n")
                game_over = True
                break
            else:
                print("\nServer says:", line)
```

3. **Listening for UDP Messages and Handling Feedback**

The listen_for_udp_messages function listens for incoming UDP messages. It processes the messages as follows:

- **Higher/Lower Feedback:** If the message is either "Higher" or "Lower", it prints the feedback to the player indicating whether their guess is too high or too low.
- **Correct Guess:** If the message is "Correct", it notifies the player that they have won the game and sets the game state to over.
- **Error Handling:** If the message starts with "Error:", it prints the error message received from the server.

The function continues to listen for messages until the game is over.

4. **Player Input Loop for Guessing**

The player_input_loop function continuously prompts the player to input their guess (between 1 and 100) or 'q' to quit the game. If the player chooses to quit by typing 'q', the game ends. The player's guess is then sent to the server via UDP. The loop continues until the player either submits a guess or chooses to quit. If any error occurs during the input process, the loop exits.

```
def player_input_loop():
    global game_over
    while not game_over:
        try:
            guess = input("\nEnter your guess (1-100) or 'q' to quit: ")
            if guess.lower() == 'q':
                game_over = True
                break

            udp_sock.sendto(f"{name}:{guess}".encode(), (SERVER, UDP_PORT))
        except:
            break
```

5. **Starting Threads for Concurrent Operations:**

This code starts three threads to handle different tasks concurrently:

1. TCP Message Listener: A thread is created to listen for incoming TCP messages from the server.
2. UDP Message Listener: A thread is created to listen for incoming UDP messages from the server.
3. Player Input Loop: A thread is created to handle the player's input for guesses.

After starting the threads, it prints "Connected. Waiting for game to start..." and enters a loop that continuously checks if the game is over. It uses time.sleep(1) to pause for 1 second before checking again. The threads run in the background, enabling concurrent processing while the player waits for the game to start.

```
# Start threads
threading.Thread(target=listen_for_tcp_messages, daemon=True).start()
threading.Thread(target=listen_for_udp_messages, daemon=True).start()
threading.Thread(target=player_input_loop, daemon=True).start()

print("Connected. Waiting for game to start...")
while not game_over:
    time.sleep(1)
```

❖ Results & Testing

In these examples, I set the minimum number of players two and started the game automatically. This means I only added two players, Yahya and Faris, without including the third player, Mohammad.

1. **<u>Two players register, game starts, one wins:</u>**

```
PS C:\Users\yahya_k6rln48\OneDrive\Desktop\University\3st\Netwotk> python Server.py
[SERVER] Listening on localhost:TCP 6000,UDP 6001
[DEBUG] Target number: 5
[TCP] Sarhan_Yahya joined from ('127.0.0.1', 64379)
[TCP] Sawalmeh_Faris joined from ('127.0.0.1', 64381)

[!] Sarhan_Yahya won! The target number was 5.
```

Figure 47: Server output showing players joining and the winner announcement.

```
Enter your name: Sarhan_Yahya
Connected. Waiting for game to start...

Enter your guess (1-100) or 'q' to quit:
=== GAME STARTED ===
Players:  Sarhan_Yahya, Sawalmeh_Faris
You have 6 guesses with 10s cooldown between guesses.

Server says: Game started! You each have 6 guesses with 10s cooldown.
30

Enter your guess (1-100) or 'q' to quit:
Feedback: Lower
Sarhan_Yahya guessed 30 (Lower)


Sawalmeh_Faris guessed 3 (Higher)
5

Enter your guess (1-100) or 'q' to quit:
🎉 Correct! You won! 🎉

========================================
===== GAME ENDED =====
🎉 Winner: Sarhan_Yahya 🎉
🔢 Correct Number: 5
========================================
```

Figure 48: Client-side game interaction showing player guesses and final winner announcement.

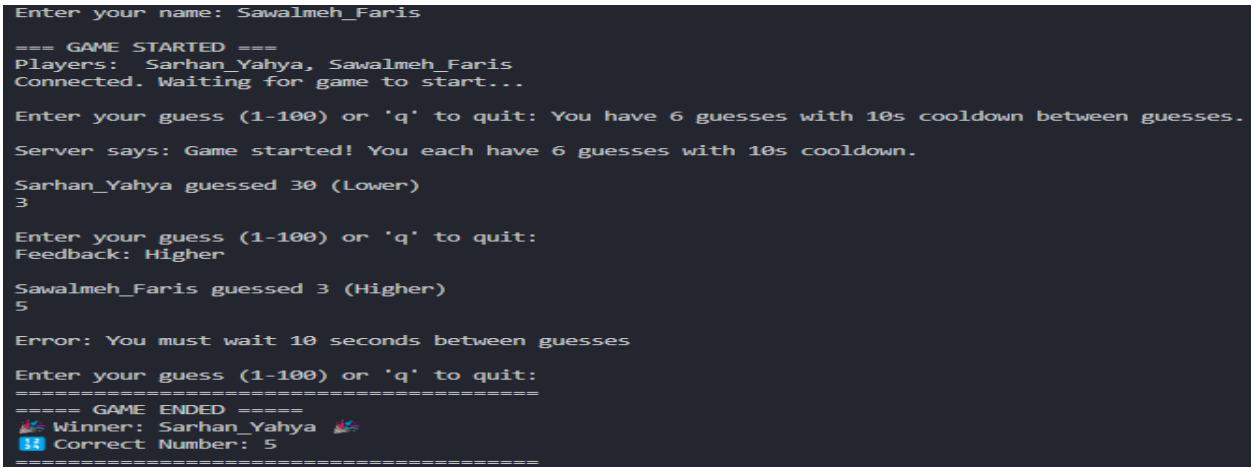


Figure 49: Game session showing cooldown enforcement and final game result with the winner.

Two players, "Sarhan_Yahya" and "Sawalmeh_Faris," sign up for the game in this test example. The server broadcasts the game start message as soon as both players are connected, and the game starts. After a predetermined number of guesses, each player is informed if their estimates are higher or lower than the target number.

Player Behavior:

Sarhan_Yahya makes a prediction of 30, which is over the desired number.

Sawalmeh_Faris makes a guess of 3, which is less than the desired number.

Sawalmeh_Faris makes a guess before the cooldown"10sec", so the server print: Error: You must wait 10 seconds between guesses

Sarhan_Yahya then guesses five, which matches the target number and wins the game.

Server Output:

- The server confirms the game start and informs both players of the number of guesses they can make and the cooldown time between guesses.
- The correct guess is announced when Sarhan_Yahya guesses the target number (5).

Player Output:

- **Sarhan_Yahya** sees feedback that their guess of 30 is Higher and later guesses correctly with 5.
- **Sawalmeh_Faris** is told their guess of 3 was Lower and attempts again after receiving a warning about the cooldown period between guesses.

Expected Outcome:

- **Sarhan_Yahya** wins the game as they correctly guess the target number.

- The game ends with a "Correct!" message, and the correct number (5) is revealed.

2. **Players guess out of bounds → get warning:**

```
Enter your name: Sarhan_Yahya
Connected. Waiting for game to start...

Enter your guess (1-100) or 'q' to quit:
=== GAME STARTED ===
Players:  Sarhan_Yahya, Sawalmeh_Faris
You have 6 guesses with 10s cooldown between guesses.

Server says: Game started! You each have 6 guesses with 10s cooldown.
50

Feedback: Higher

Sarhan_Yahya guessed 50 (Higher)
Enter your guess (1-100) or 'q' to quit:
120

Error: You must wait 10 seconds between guesses

Enter your guess (1-100) or 'q' to quit: 120

Error: Guess must be between 1-100

Enter your guess (1-100) or 'q' to quit: 87

Enter your guess (1-100) or 'q' to quit:
🎉 Correct! You won! 🎉

========================================
===== GAME ENDED =====
🎉 Winner: Sarhan_Yahya 🎉
🔢 Correct Number: 87
========================================
```

Figure 50: Player session showing error handling for cooldown violation and invalid input range.

This test case validates how the game handles an invalid input scenario, where a player submits a guess outside the accepted range of 1 to 100. The system should reject the guess, notify the player with an appropriate error message, and allow the game to continue without affecting the guess count or the gameplay for the other player.

Player Actions:

- Sarhan_Yahya starts the game and submits a valid guess of 50 (Higher).
- Before the cooldown period ends, Sarhan_Yahya attempts to guess 120, which is invalid (out of bounds).
- The server displays an error for early input (10 seconds cooldown),
- After the cooldown period ends, Sarhan_Yahya attempts to guess 120, which is invalid (out of bounds).
- The server displays an error message (Guess must be between 1-100).
- Sarhan_Yahya submits a valid guess of 87, which is correct.

Expected Outcome:

- The out-of-range guess (120) is rejected with a clear warning.
- Game logic is unaffected by the invalid input.
- A valid guess after the error (87) is accepted, and the game concludes correctly.
- Sarhan_Yahya wins, and the correct number (87) is revealed.

3. **<u>One player disconnects mid-game → game continues or ends</u>**

Objective:
Ensure that the game correctly handles the scenario when one player disconnects during gameplay, and checks if the game continues with one player or ends based on the decision of the remaining player.

Player Actions:

1. Sarhan_Yahya starts the game, and both players are connected.
2. Both players make their first guesses.
3. During the game, Sarhan_Yahya disconnects.
4. The server detects that Sarhan_Yahya has disconnected.
5. The server prompts Sawalmeh_Faris, the remaining player, asking if they want to continue the game alone.
6. Sawalmeh_Faris confirms they want to continue alone.
7. The game continues with Sawalmeh_Faris making guesses alone, and they eventually guess the correct number.

Expected Outcome:

- When Sarhan_Yahya disconnects, the server handles the disconnect appropriately and notifies the remaining player (Sawalmeh_Faris).
- The remaining player (Sawalmeh_Faris) is given the option to continue playing alone or end the game.
- If Sawalmeh_Faris chooses to continue, the game continues without interruption.
- The gameplay continues without affecting the progress of the game, and Sawalmeh_Faris successfully guesses the correct number (8), winning the game.

```
[SERVER] Listening on localhost:TCP 6000,UDP 6001
[DEBUG] Target number: 8
[TCP] Sarhan_Yahya joined from ('127.0.0.1', 64752)
[TCP] Sawalmeh_Faris joined from ('127.0.0.1', 64754)
[DISCONNECT] Sarhan_Yahya disconnected mid-game.
[GAME] Sawalmeh_Faris chose to continue alone

[!] Sawalmeh_Faris won! The target number was 8.
```

Figure 51: Player left; game continued solo to win.

```
Enter your name: Sarhan_Yahya
Connected. Waiting for game to start...

Enter your guess (1-100) or 'q' to quit:
=== GAME STARTED ===
Players:  Sarhan_Yahya, Sawalmeh_Faris
You have 6 guesses with 10s cooldown between guesses.

Server says: Game started! You each have 6 guesses with 10s cooldown.
50

Enter your guess (1-100) or 'q' to quit:
Feedback: Lower
Sarhan_Yahya guessed 50 (Lower)


Sawalmeh_Faris guessed 2 (Higher)
q
PS C:\Users\yahya_k6rln48\OneDrive\Desktop\University\3st\Netwotk>
```

Figure 52: Early guesses, then player quit.

- **First scenario when Faris select to continue:**

```
Enter your name: Sawalmeh_Faris

=== GAME STARTED ===
Players:  Sarhan_Yahya, Sawalmeh_Faris
You have 6 guesses with 10s cooldown between guesses.

Server says: Game started! You each have 6 guesses with 10s cooldown.
Connected. Waiting for game to start...

Enter your guess (1-100) or 'q' to quit:
Sarhan_Yahya guessed 50 (Lower)
2

Feedback: Higher
Enter your guess (1-100) or 'q' to quit:

Sawalmeh_Faris guessed 2 (Higher)
79

Feedback: Lower

Sawalmeh_Faris guessed 79 (Lower)
Enter your guess (1-100) or 'q' to quit:

Player  Sarhan_Yahya left the game. Do you want to play alone? (Y/N) left the game.
Y
Do you want to play alone? (Y/N): Y

 Game continues with you alone. You have 60 seconds to guess!
Enter your guess (1-100) or 'q' to quit:

Error: Invalid input! Only numbers are allowed.
8

Enter your guess (1-100) or 'q' to quit:
🎉 Correct! You won! 🎉

========================================
===== GAME ENDED =====
🎉 Winner: Sawalmeh_Faris 🎉
🔢 Correct Number: 8
========================================
```

Figure 53: First scenario when Faris select to continue

- **Second scenario when Faris select to stop the game:**

```
Enter your name: Sawalmeh_Faris

=== GAME STARTED ===
Connected. Waiting for game to start...
Players:  Sarhan_Yahya, Sawalmeh_Faris
You have 6 guesses with 10s cooldown between guesses.

Enter your guess (1-100) or 'q' to quit:
Server says: Game started! You each have 6 guesses with 10s cooldown.
30

Feedback: Higher

Enter your guess (1-100) or 'q' to quit:
Sawalmeh_Faris guessed 30 (Higher)

Player  Sarhan_Yahya left the game. Do you want to play alone? (Y/N) left the game.
N
Do you want to play alone? (Y/N): N

Enter your guess (1-100) or 'q' to quit:
 Game ended by your choice. No winner.
PS C:\Users\yahya_k6rln48\OneDrive\Desktop\University\3st\Netwotk>
```

Figure 54: Second scenario when Faris select to stop the game

4. **Time Up → Game Ends without a Winner**

```
[SERVER] Listening on localhost:TCP 6000,UDP 6001
[DEBUG] Target number: 25
[TCP] Sarhan_Yahya joined from ('127.0.0.1', 64840)
[TCP] Saw_Faris joined from ('127.0.0.1', 64842)
[!] Game ended. Time's up! The target number was 25.
```

Figure 55: Game ended due to time out.

```
Enter your name: Saw_Faris

=== GAME STARTED ===
Players:  Sarhan_Yahya, Saw_Faris
Connected. Waiting for game to start...
You have 6 guesses with 10s cooldown between guesses.

Enter your guess (1-100) or 'q' to quit:
Server says: Game started! You each have 6 guesses with 10s cooldown.

========================================
===== GAME ENDED =====
⏰ Time's up! No winners.
========================================
```

Figure 56: Time ran out with no winner.

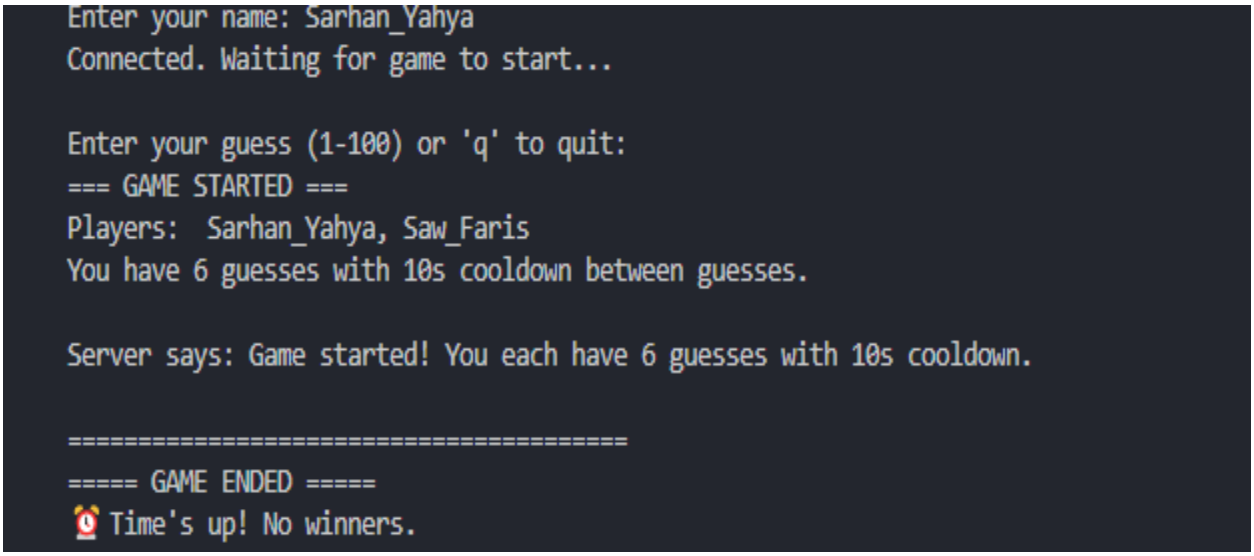


Figure 57: Game started but ended with no winners.

Objective:
Ensure that the game correctly handles the scenario when the time limit is reached and no player has guessed the target number. The game should end with no winner, and the target number should be revealed.

Player Actions:

1. Both players, Sarhan_Yahya and Saw_Faris, start the game.
2. Each player has 6 guesses, with a 10-second cooldown between guesses.
3. The players make their guesses(in this example they don't guesses), but neither player is able to guess the correct number within the given time.
4. Once all guesses are exhausted or the time limit is reached, the game ends.
5. The server displays a message that the time is up and reveals that there is no winner.

Expected Outcome:

- The game ends after the time limit is reached, and the target number is revealed.
- Since neither player guesses correctly in time, the server announces "No winners."
- The game concludes gracefully with no errors or issues, and players are informed that time is up.

Alternative Solutions, Issues, and Limitations

In our **network project**, we faced several **challenges**, including **team task coordination** and **grasping complex concepts** within the project. To overcome these issues, we conducted **additional research** and engaged in **collaborative discussions** to explore various ideas. We also experimented with **alternative approaches** to ensure we stayed on the **right path**. Although we encountered some **limitations**, such as the **time required** to understand specific components, these difficulties significantly enhanced our **teamwork** and **problem-solving abilities**.

Teamwork

We ensured **equal task distribution** among team members, assigning roles such as **coding**, **testing**, and **report writing** fairly. We held **regular meetings** at the **university** to discuss our **progress** and address any **issues** that came up. When **in-person meetings** weren't possible, we organized **online sessions** to maintain **coordination** and continue working **collaboratively**.

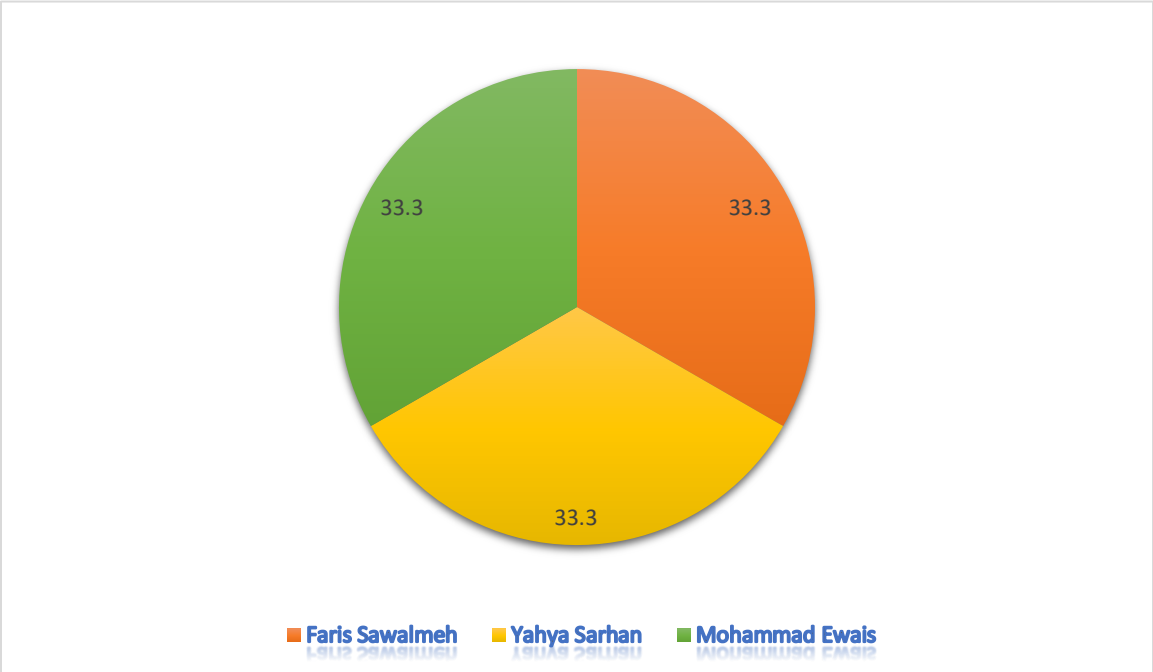


Figure 58: Teamwork Chart

References

[1] : **https://www.geeksforgeeks.org/**

[2] : **https://www.scaler.com/topics/socket-programming-in-c/**

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

[4] : **https://www.comptia.org/content/articles/what-is-wireshark-and-how-to-use-it**