Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Computer Networks

ENCS3320

First Semester 2024 / 2025

Project 1 (Socket Programming)

Prepared by:

Lara Foqaha	1220071	**Section:** 4
Veronica Wakileh	1220245	**Section:** 4
Fadi Bassous	1221005	**Section:** 4

Date: 1/12/2024

Abstract

This project explores network programming through a TCP-based web server and a UDP multiplayer trivia game. The web server handles HTTP requests, serving static files and managing concurrent client connections with threading and error handling. Advanced response generation, including dynamic content and custom error pages, is also implemented. The trivia game utilizes UDP for low-latency real-time interaction, incorporating message broadcasting and game state management. By combining TCP and UDP protocols, the project provides practical experience in socket programming, threading, and protocol implementation, offering a strong foundation in developing distributed systems.

Table of Contents

Table of Figures

1. Theory and Procedure

1.1 Task 1: Network Commands and Wireshark

Tools Used:

Wireshark: In this task we used Wireshark, which is a very popular network protocol analyzer that captures and inspects packets (discrete units of data exchanged over a network connection, such as between a computer and the internet or home office network) to capture a DNS query and reply for any hostname of our choice. It's known to be the most popular packet sniffer in the world, Wireshark is considered the tool of choice when analyzing and troubleshooting network communications [1].

Command Prompt (CMD): The Command Prompt in Windows is a command-line interface through which the user can give commands to the system, run advanced administration utility programs, and troubleshoot various problems [2]. In this task we used it to run the following commands:

1. **Ipconfig (ipconfig / all):** Displays all current TCP/IP network configuration values and refreshes Dynamic Host Configuration Protocol (DHCP) and Domain Name System (DNS) settings. Used without parameters, and displays Internet Protocol version 4 (IPv4) and IPv6 addresses, subnet mask, and default gateway for all adapters [3].

2. **Ping:** Tests the IP-level connectivity to another TCP/IP host by sending ICMP Echo Request messages and showing the corresponding Echo Reply messages along with round-trip times. It is a primary tool for troubleshooting connectivity, reachability, and name resolution issues in TCP/IP networks [4].

3. **tracert**: This command traces and lists the route taken by data packets from the source to the destination over an Internet Protocol network. It is a diagnostic tool that shows the possible routes, as well as measuring transit delays at each hop.

4. **telnet:** client-server protocol for connecting to remote systems over a network or the internet as if the user were using virtual terminals. It also enables sending HTTP requests directly by entering them after connecting to a website.

5. **nslookup:** A network administration tool used to query the Domain Name System (DNS) for mapping domain names to IP addresses and vice versa, as well as retrieving other DNS records. It assists in diagnosing problems with DNS infrastructure and is installed with the TCP/IP protocol [5].

Procedure:

To perform this task, we first opened the Command Prompt and ran some network commands. First, we ran the command **ipconfig /all** in order to get some information about the system's IP address, subnet mask, default gateway, and DNS server. Then, we used the **ping** command in order to test the connectivity with a local device connected on the same network and an external server, showing response times and the successful ping of devices. Then, we applied **tracert** to trace the route to an external server, identifying the intermediate hops and delays. Using **nslookup**, we fetched DNS details for a given domain and tried connecting to it using **telnet**. Using **Wireshark**, DNS query and response packets were captured and analyzed, starting a capture on an active network interface, performing a DNS query, and saving the relevant packet data for analysis. Results and observations are documented with screenshots in the results and discussion section.

1.2 Task 2: Web Server

The web server follows the client-server architecture, where the server continuously listens for incoming client requests and responds appropriately. Each client request is handled independently, with no memory of past interactions. The server responds with HTTP status codes, including:

- 200 OK: Sent when the requested resource is successfully delivered.
- 404 Not Found: Used when the requested file cannot be located.
- 307 Temporary Redirect: Redirects clients to an external resource, such as Google or YouTube, when specific files (images or videos) are unavailable.

The Transmission Control Protocol (TCP) provides a reliable transport service between a sending and receiving process, ensuring that data is delivered accurately and in the correct order. It incorporates flow control mechanisms to prevent the sender from overwhelming the receiver, and congestion control features to adjust the sender's transmission rate when the network becomes overloaded. This is why it was used in this task. TCP is a connection-oriented protocol, requiring an initial setup phase to establish a connection between the client and server before data exchange begins. However, TCP does not inherently provide guarantees for timing, minimum throughput, or security, making it suitable for applications where reliability is prioritized over speed or timing constraints like this web server task [6].

Task 2 utilizes **persistent HTTP connections**, which allow multiple objects to be sent over a single TCP connection between the client and server. This approach reduces the overhead of repeatedly opening and closing connections, improving efficiency and response time. Unlike **non-persistent HTTP**, where each object requires a separate connection, persistent HTTP keeps the connection open for subsequent requests, enabling faster and more streamlined communication for serving multiple resources like HTML files, images, and CSS styles.

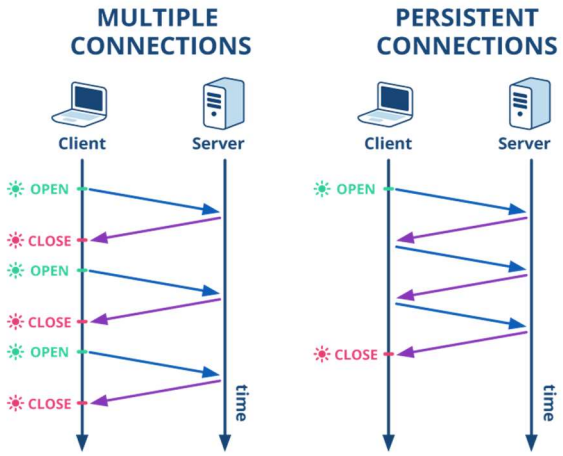


Figure 1: Persistent vs non-persistent http connections [7]

The implementation uses **socket programming**, using TCP sockets to establish reliable communication. The server binds to a specific IP address and port (in this case, 5698) and listens for incoming connections. TCP ensures that data transmitted between the server and clients is reliable and ordered, making it an ideal choice for serving web resources.

Socket programming with TCP enables reliable communication between a client and a server by establishing a connection-oriented protocol. For the communication to begin, the server process must be running and have created a welcoming socket, which acts as a "door" for incoming client connections. The client initiates contact by creating a TCP socket and specifying the server's IP address and port number. Upon connection, the server's TCP protocol creates a new socket dedicated to communicating with that specific client, allowing the server to manage multiple clients simultaneously. Each client is distinguished using unique source port numbers, ensuring seamless interaction in a multi-client environment [6].

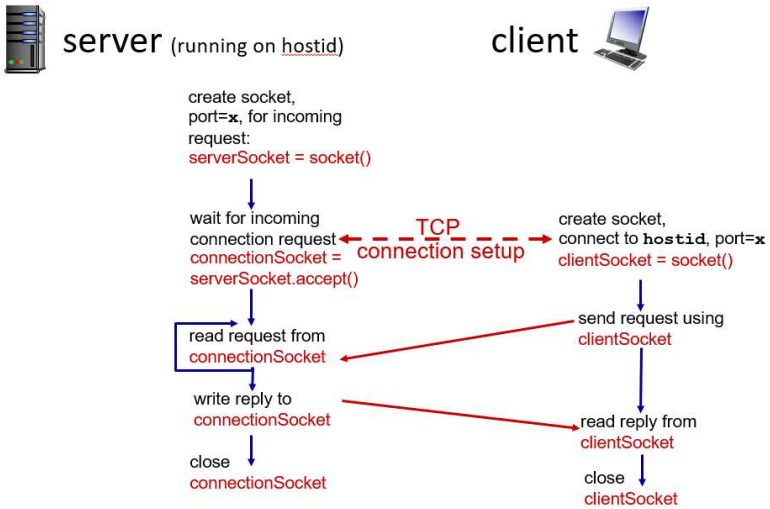


Figure 2: Client/server socket interaction: TCP [6]

Procedure:

We created the web server by following these steps:

1) **Server initialization**: We created a TCP socket using *socket.socket(socket.AF_INET, socket.SOCK_STREAM)* and bound it to the server's IP address and port 5698. Then the server started listening for incoming client connections, with a maximum size of 10.
2) **Accepting client connections**: The server started accepting connections using *server_socket.accept()*, and for each one, a new thread was spawned using *threading.Thread*. Ensuring concurrent processing of multiple requests.
3) **Handling client requests**: We parsed the received HTTP request (after decoding it) to extract the HTTP method (like GET) and the requested file path to map them to specific files. Then the file type was determined using *get_content_type* function to construct an appropriate HTTP response.
4) **Serving responses**: When the file exists, it is opened in binary mode, read its contents, and returned to the client with a (200 OK) response header. When the file doesn't exist, if it is an image or video, the client is redirected to an external URL (Google for images or YouTube for videos) with a (307 Temporary Redirect) status. For all other missing files, a (404 Not Found) error page is sent. This page includes the client's IP address and port.
5) **Closing connections**: After sending the response, the client socket is closed by using *client_socket.close()*

1.3 Task 3: UDP Client-Server Trivia Game Using Socket Programming

The purpose of this task is to develop a multiplayer trivia game using UDP socket programming, enabling real-time communication between a server and multiple clients. The server broadcasts trivia questions, then collects responses from the clients, calculates scores, and updates the leaderboard in a competitive, interactive environment. Also, it acts as a central hub for communication, coordinating interactions between all connected clients as there is no direct client-to-client communication allowed. Hence, this game follows the client-server architecture. UDP is crucial in the implementation of this game, since it is connectionless and lightweight providing fast and simultaneous communication with various clients, which is necessary for the real-time factor of the game. UDP is especially perfect for this application, considering that a little loss of data is acceptable while responsiveness is more important. It also places much emphasis on handling concurrency, real-time data exchange, and designing robust client-server interactions.

The User Datagram Protocol **UPD** is a connectionless transport protocol offering a minimal service with only basic multiplexing/demultiplexing and error checking mechanisms. It allows for the transmission of data without any preliminary handshaking or establishment of a connection, which makes it faster and lighter than TCP. UDP is suitable for real-time applications such as DNS, streaming, and online gaming, where speed is more important than reliability. For instance, DNS queries use UDP because they require fast, stateless transmission without delays caused by connection setup or congestion control, making it ideal for quick responses. However, UDP does not guarantee reliability, flow control, congestion control, timing, minimum throughput, security, or order of packets, and it doesn't perform retransmissions or error recovery, leaving that responsibility to the application layer if needed [6].

The primary advantage of **UDP** over TCP is its speed and efficiency for applications that can tolerate some degree of packet loss. Real-time applications, such as video conferencing or Internet phone calls, use **UDP** because it does not introduce delays associated with handshakes or congestion control before packets are sent. These applications can tolerate small losses of data, and if necessary, developers can make their error-handling mechanisms. **UDP** is also efficient in handling multiple clients with minimal overhead because it has a small header of 8 bytes without any connection state management. As it does not have the built-in reliability, **UDP** is mostly used in the cases where real-time performance is critical or minor losses do not result in noticeable degradation of an application's functionality [6].

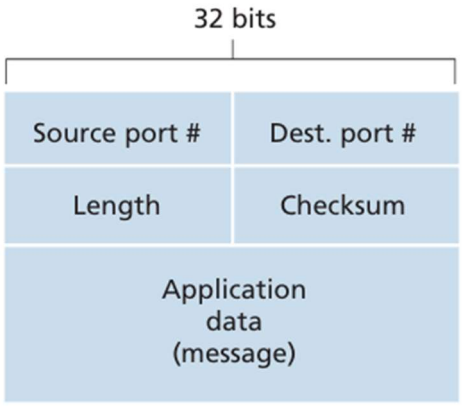


Figure 3: UDP segment structure

In **UDP** socket programming, communication between processes running on different machines is performed by sending messages through sockets, which serve like doors connecting the application and transport layers. For a process to send data, it attaches a destination address to the packet, which includes the IP address of the destination host and the port number of the destination socket. This therefore enables routers to route the packet to the correct host and to the appropriate application process. The sending process also includes its own source address, which includes IP address and port number, although this is done automatically by the operating system. Because there is no connection to be established, this will enable direct, fast communications, making **UDP** ideal for real-time applications [6].

In a client-server interaction using **UDP**, the client sends a message to the port on which the server is listening (in this case,5698). At that moment, the server should be up and listening on that port to receive the message and take further action. whereas, the client program sends only a simple message to the server. The client and the server exchange data over **UDP sockets** with minimal overhead. Hence, this protocol is suitable for applications that require speed and can afford to lose some packets, such as the real-time trivia game. error handling and connection management are normally minimal to avoid unnecessary delay.

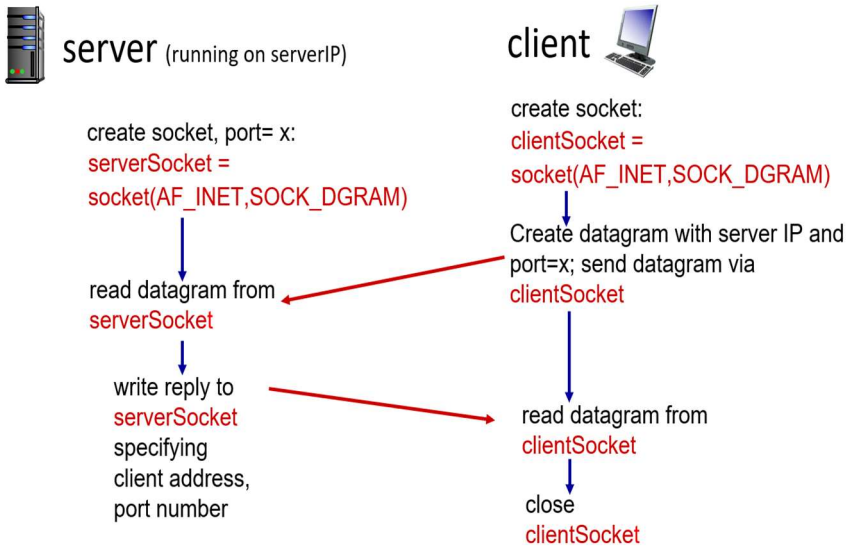


Figure 4: Client/server socket interaction: UDP

Procedure:

We created the Client-Server Trivia Game by following these steps:

1) **Server setup and client registration**: We created a UDP socket using *serverSocket = socket.socket(AF_INET, SOCK_DGRAM)* and bound it to the server's IP address and port 5698. Then the server started listening for incoming client connections. When a client joins, they send their username, and the server adds them to the *active_clients* list, assigns them a score of zero, and broadcasts the new player information to all clients.

2) **Game round setup**: After at least two players join, the server broadcasts a countdown message to start the game. The server selects a set of questions and broadcasts them to all active clients. Each question has a time limit (set by *QUESTION_TIME*), and the server starts a timer when the question is broadcast.

3) **Answer collection and timeout handling:** The server waits for client answers during the allowed time. Correct answers are awarded points, with faster responses earning more points. If no answer is given within the time limit, the server marks the client as inactive for that question. The correct answer and updated leaderboard are broadcast after each question.

4) **Inactive clients and client exit:** Clients who do not answer any questions or choose to exit by typing "exit" are removed from the game. The server broadcasts a message to inform the remaining clients of the exit or inactivity. If there are fewer than two active clients, the round is canceled.

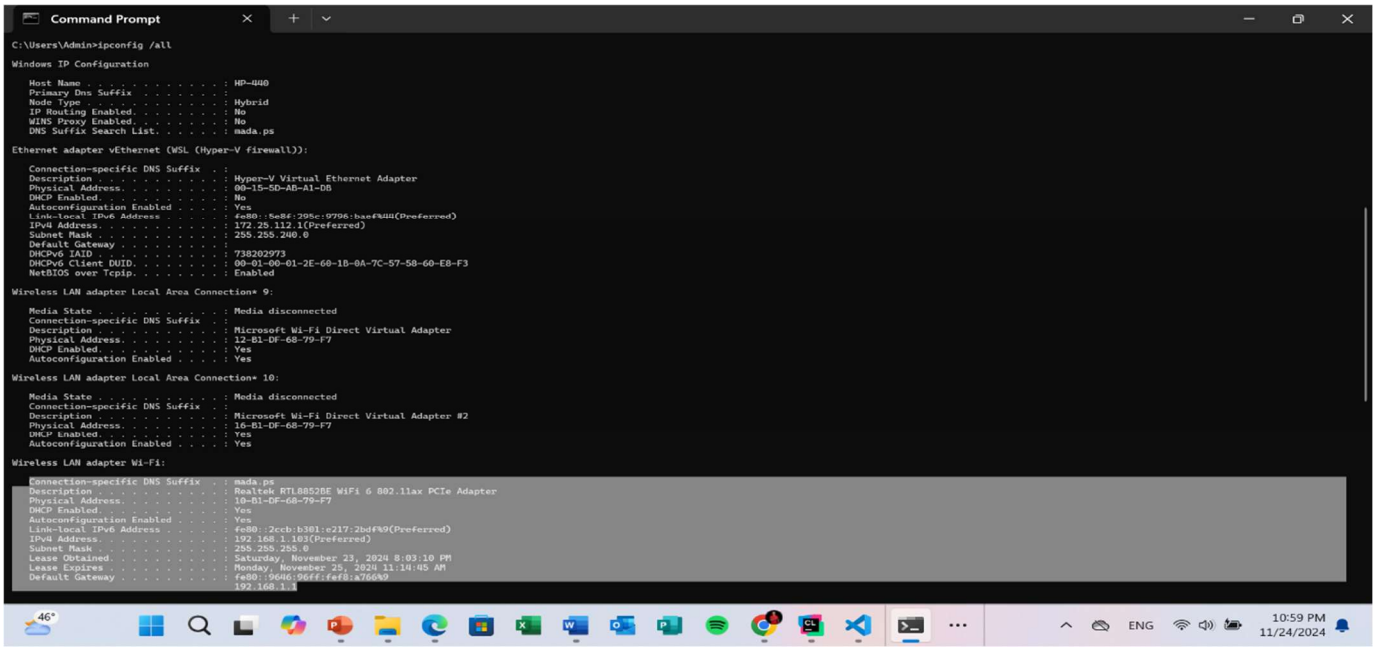
5) **Winner announcement:** After all questions are asked, the server determines the winner based on the highest score and broadcasts the winner(s) to all clients. If no active players remain, the game ends. Otherwise, the server may start a new round.

6) **Client participation**: Clients enter the server's IP address, port, and their username to join. After receiving questions, they provide answers within the given time. If they fail to answer or type "exit," they are removed from the game. The client is notified of the winner once the game ends.

2. Results and Discussion

2.1 Task 1: Network Command and Wireshark

- Run the **ipconfig /all** command on your computer and identify the IP address, subnet mask, default gateway, and Domain Name System (DNS) server addresses for your primary network interface.



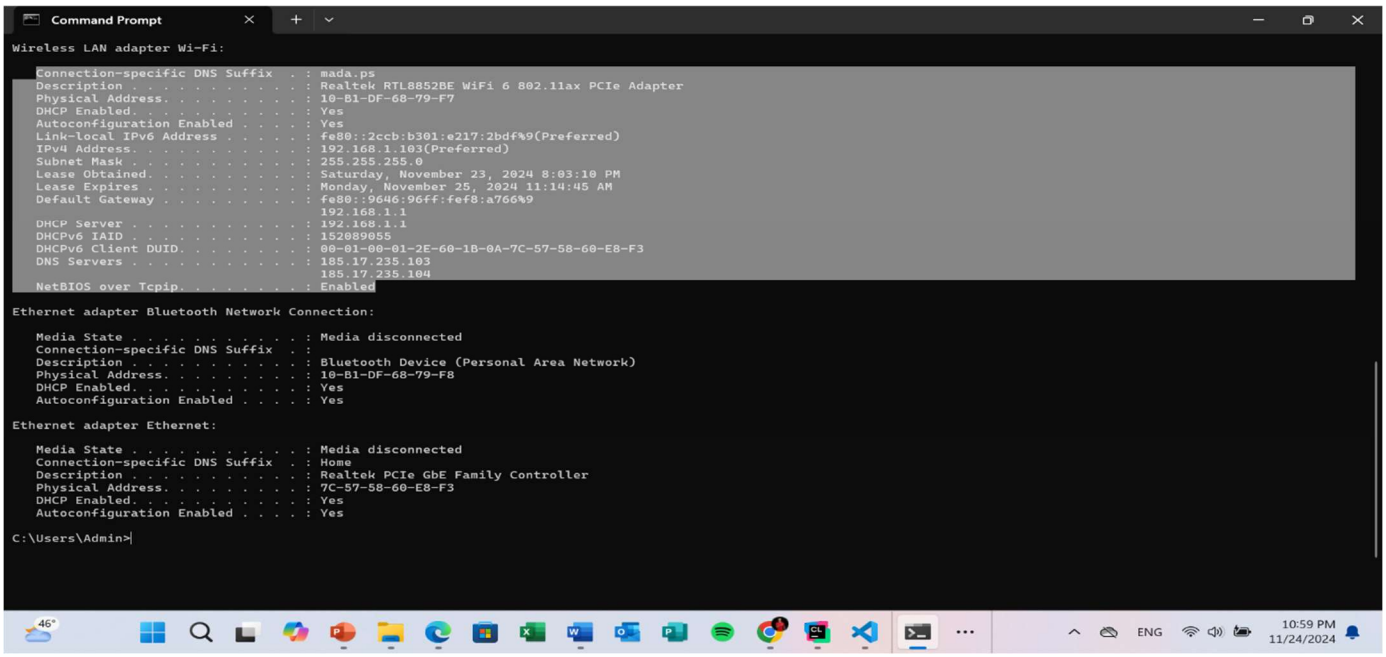


Figure 5: ipconfig/all command result

The **ipconfig /all** command was executed to present detailed network configuration for the system's adapters. The relevant details for the active **Wireless LAN adapter Wi-Fi** are as follows:

IP Address: 192.168.1.103, assigned dynamically through the DHCP server to communicate over a local network.

Subnet Mask: 255.255.255.0, defining the network range and thus allowing communication with devices in the subnet 192.168.1.x (x ranges from 1 to 254) without needing a router.

Default Gateway: 192.168.1.1, the router address enabling access to external networks, including the internet.

DNS Servers: 185.17.235.103 and 185.17.235.104, used for translating domain names into IP addresses for internet communication.

This command confirmed that the network configuration is correct to allow connectivity within the local network and to the internet.

- **Ping** a device within your local network (e.g., from your laptop to a smartphone on the same Wi-Fi network).

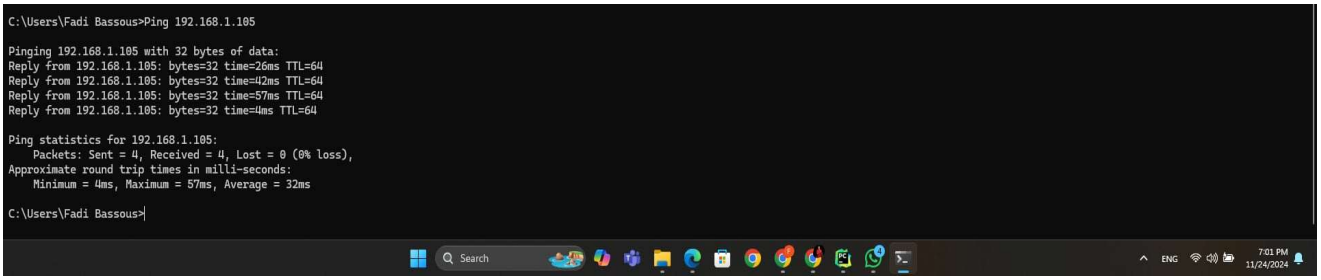


Figure 6: ping another device on the same network (cmd) result

The **ping** command tested the connectivity to the local device with the IP address **192.168.1.105**. The device responded successfully, with no packet loss recorded 0% loss. Round-trip times fluctuated between 4 ms minimum and 57 ms maximum, while the average response time was 32 ms, meaning that the local device is reachable and communicating effectively on the network.

- **Ping** *discover.engineering.utoronto.ca.* Based on the results, briefly explain whether you believe the response originates from Canada.

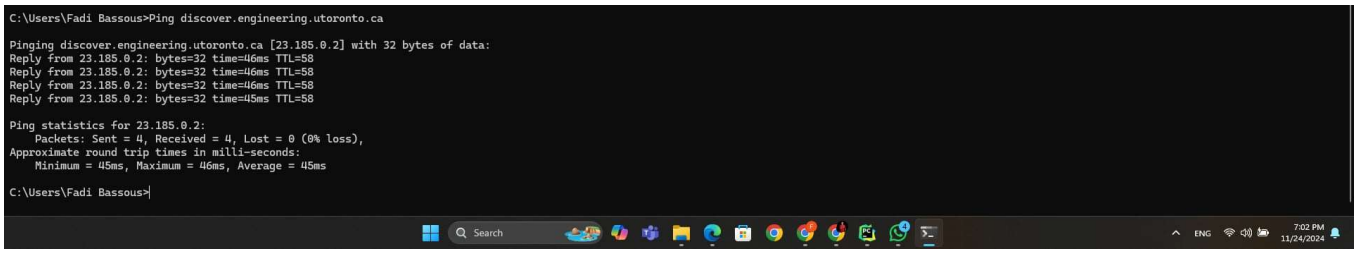


Figure 7: ping website (cmd) result

When executing this command, it sent ICMP Echo Request packets to the server at *discover.engineering.utoronto.ca* with IP **23.185.0.2**, which responded with Echo Reply packets. The RTT was about **45 - 46** ms, indicating a fast and stable connection with no packet loss. A latency of 46ms indicates that the server is geographically close to Palestine. Given the significant physical distance between Palestine and Canada, it is highly unlikely for RTTs to be this low if the server were in Canada. The server is almost certainly not located in Canada. Instead, the response likely comes from a nearby Fastly CDN edge server located much closer to Palestine, possibly in Europe or the Middle East.

This happens because most universities and organizations provide content through content delivery networks, or **CDN**s, like Fastly, that provide optimized and often accelerated web content delivery worldwide. In this case, the server from Fastly responded to the ping request so that it could provide better performance no matter where users were located in the world, rather than the server directly at the University of Toronto [8].

- Run tracert on *discover.engineering.utoronto.ca*.

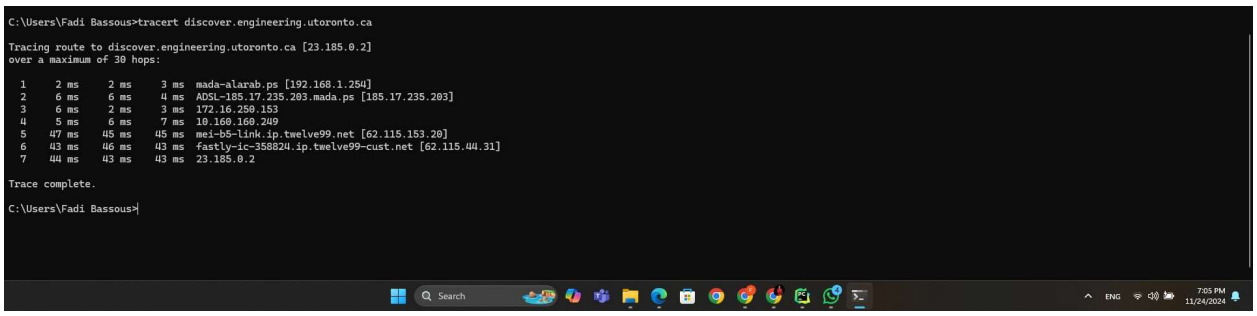


Figure 8: tracert command (cmd) result

Executing the **tracert** *discover.engineering.utoronto.ca* command revealed the path taken by packets from my device to the University of Toronto server (**23.185.0.2)**. This utility traced each hop along the route, starting from the local gateway **192.168.1.254**, intermediary routers like **185.17.235.203** and **172.16.250.153**, passing through **62.115.44.31** as an external node. Each hop's IP address and round-trip time were documented with progressively growing latency from 2 ms locally to 43 ms at the destination. This is very informative output about intermediate stops that can give valuable insights into the network path, locating a bottleneck and confirming connectivity to the target server.

- Use **nslookup** to retrieve the DNS information for *discover.engineering.utoronto.ca*

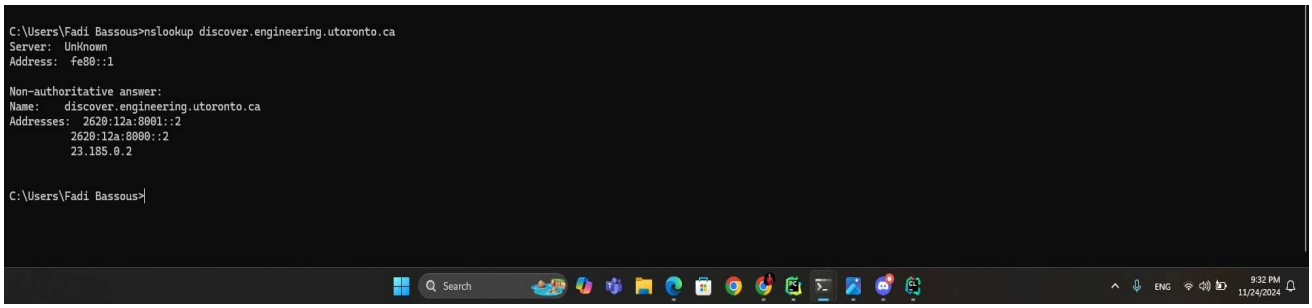


Figure 9: nslookup on (cmd) result

Executing this command on the command prompt was for querying DNS information for the host *discover.engineering.utoronto.ca* university of Toronto faculty of applied science and engineering website, which translates the domain name to its corresponding IP addresses. The command output includes a non-authoritative answer, indicating the data was retrieved from a cached DNS server rather than an authoritative source. Those IP addresses resolved on this domain are **2620:12a:8001::2** and **2620:12a:8000::2** (IPv6), while it is **23.185.0.2** in (IPv4). This tool will be essential in troubleshooting DNS-related problems, confirmation of domain configurations, and details related to DNS records and associated IP addresses.

- Attempt to connect with **telnet** to *discover.engineering.utoronto.ca.*

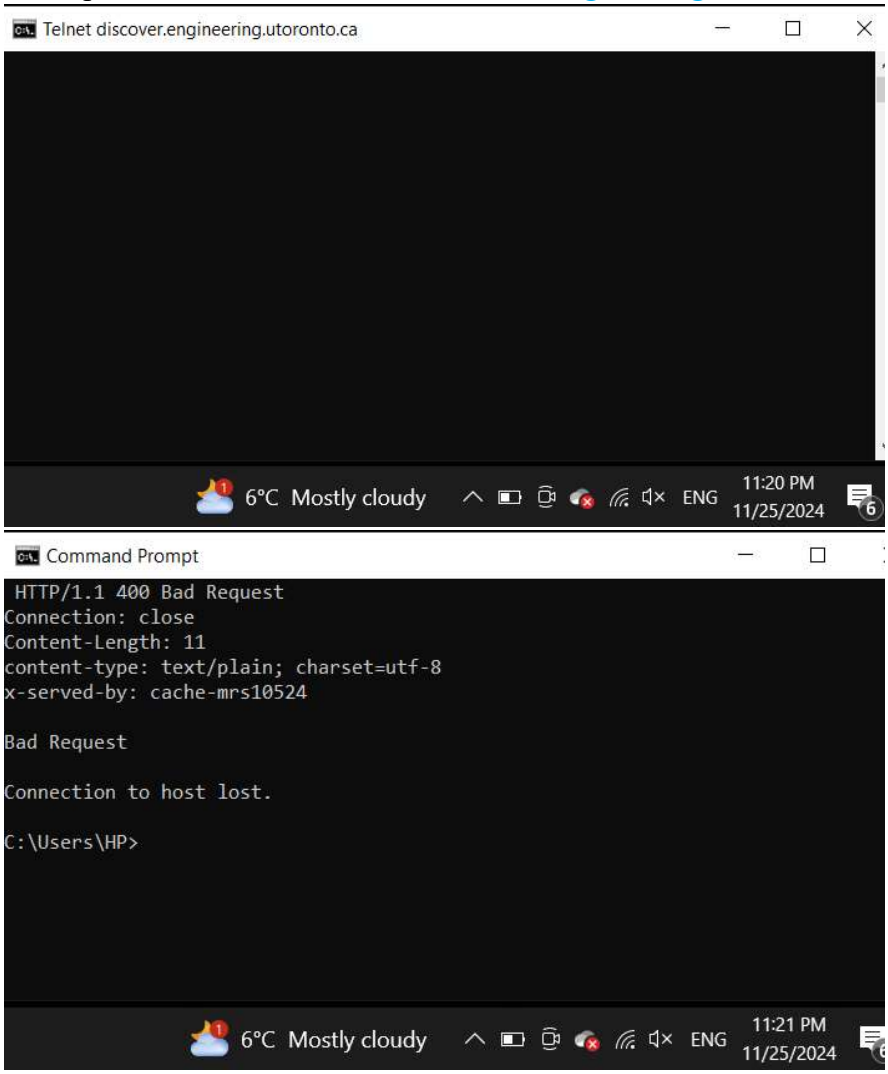


Figure 10: telnet command result

This command tried to initiate a **telnet** session with the server at *discover.engineering.utoronto.ca* on port 80, commonly used for HTTP connections. The output indicates that the connection to the server on port 80 (HTTP) was successfully established, but the server returned a 400 Bad Request error. The Bad Request body confirms that no valid HTTP request (like a GET or POST) was sent. This outcome highlights that while the server is reachable and listening on the specified port, the lack of a properly formatted HTTP request from the telnet command led to the error. This is expected behavior since telnet does not inherently generate HTTP-compliant requests.

- Use the **Wireshark** packet analyzer to capture a DNS query and reply for any hostname of your choice.

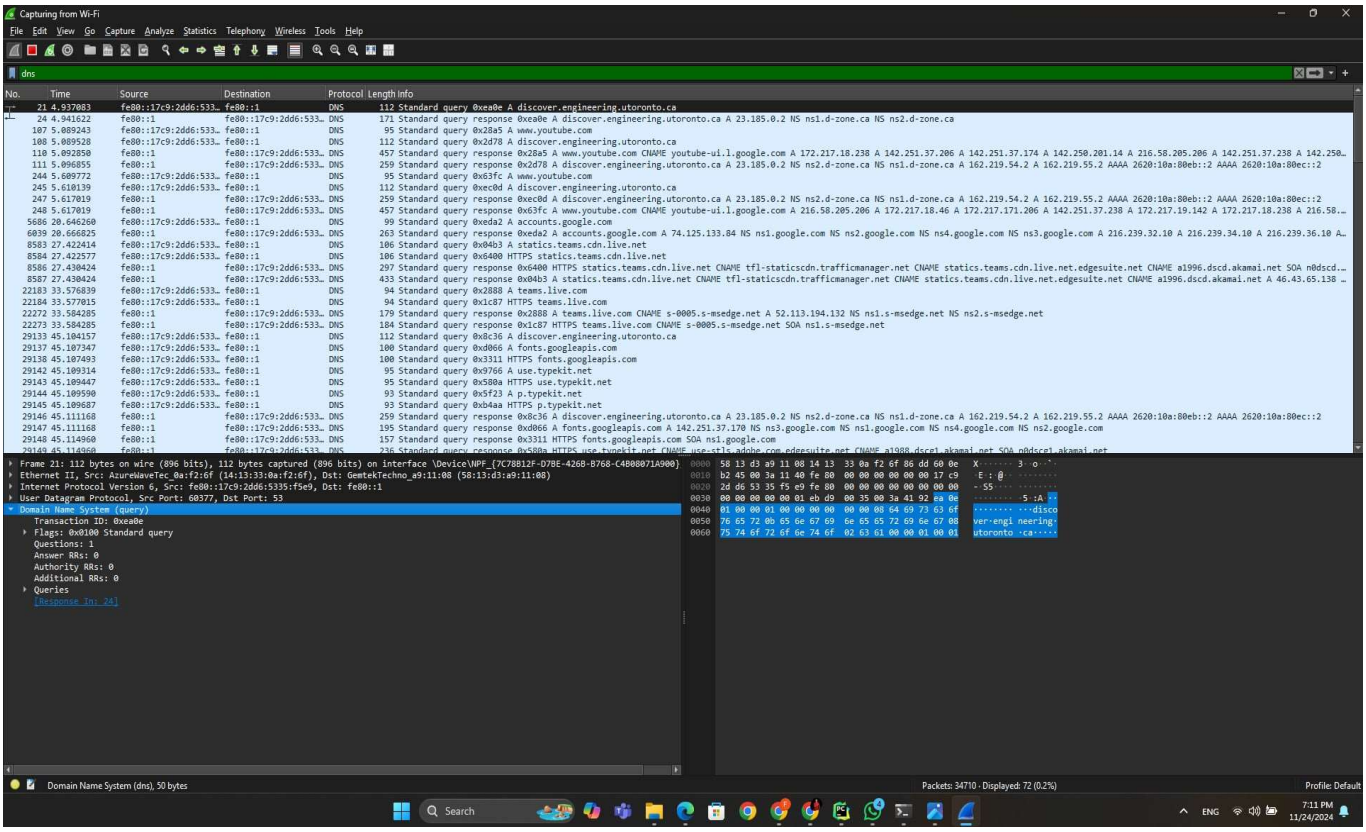


Figure 11: captured DNS query

Figure 7 has a **Wireshark capture of DNS activity** for the query *discover.engineering.utoronto.ca*. In the captured DNS query packet sent to the DNS server, it requests the resolution of this domain name, details can be viewed in the Queries section of the packet analysis. The DNS response packet from the server includes the resolved IP address of **23.185.0.2** and other information, such as authoritative name servers **ns1.d-zone.ca** and **ns2.d-zone.ca**.

Although this analysis confirms that the domain was successfully resolved, it does not immediately indicate the geographic location of the response within the DNS activity. An IP lookup utility showed that the IP address of the response, **23.185.0.2**, is from a server in San Francisco, California.

Combining this with the ping results, the **round-trip time RTT** alone doesn't serve reliably to indicate the physical geographical location of the server, as it depends on many factors such as routing in networks and latency optimizations. This again points out the need for DNS and other external tools for proper network diagnostics with geographical insight.

2.2 Task 2: Web Server

- **Basic HTTP Request Handling:** When a client sends an HTTP GET request for paths such as /, */en*, */index.html*, or */main_en.html*, the server successfully maps these requests to the main_en.html file and serves it with a (200 OK) response.

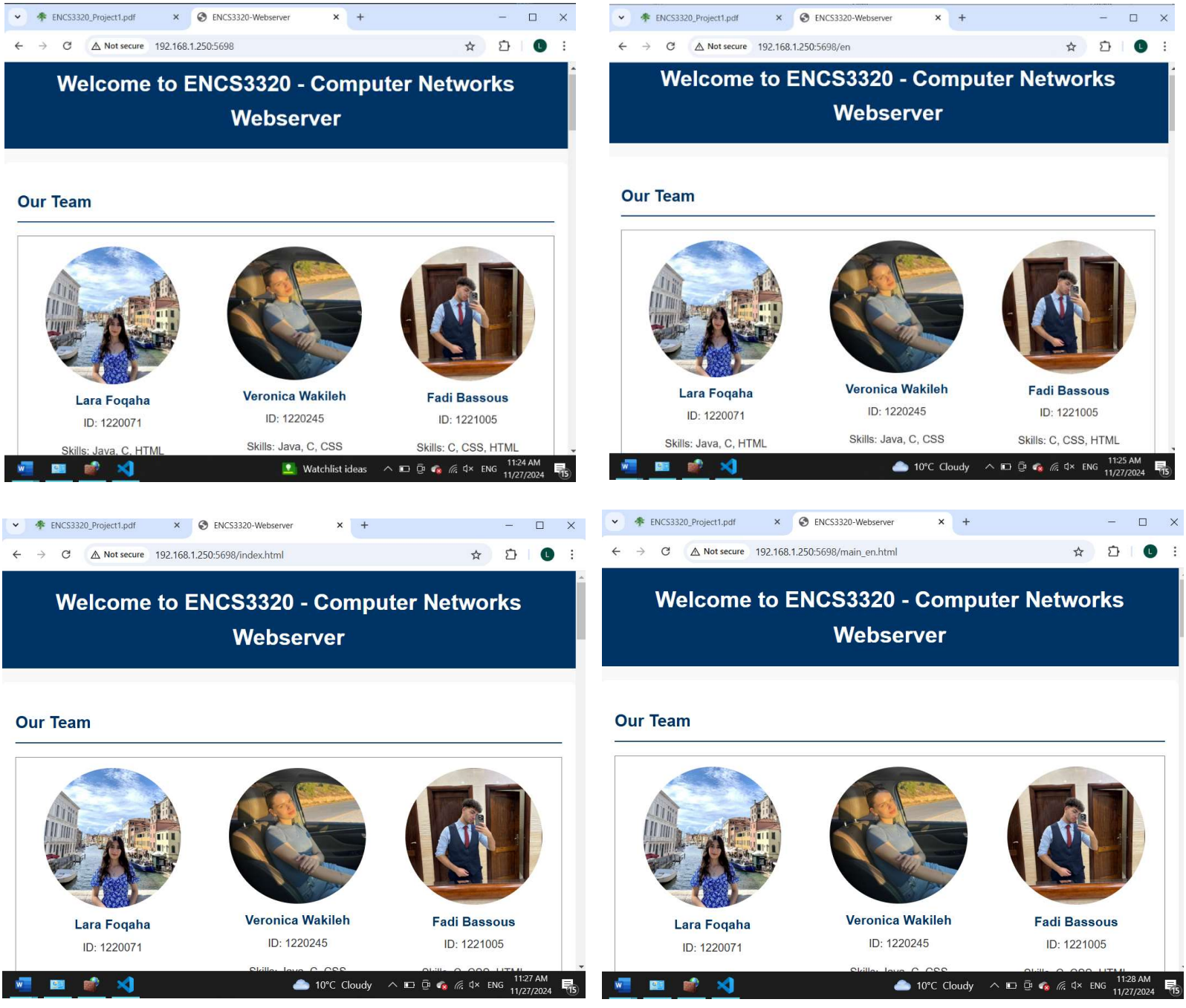
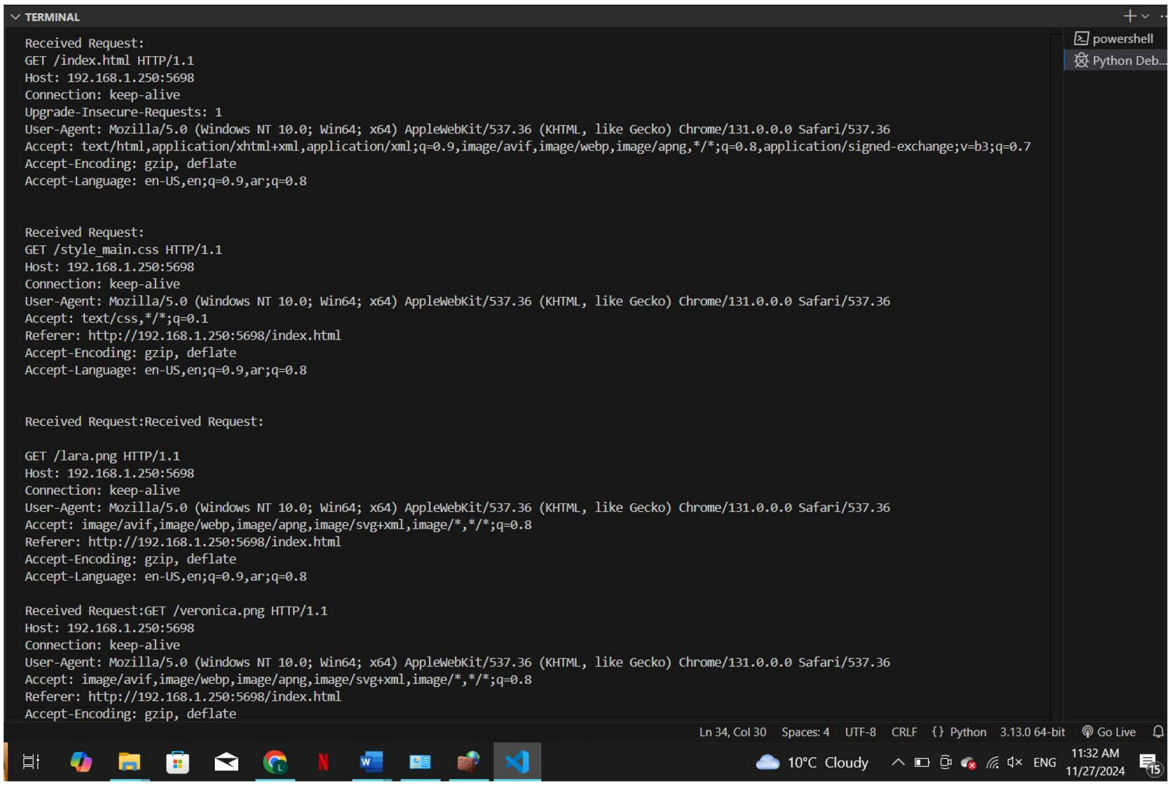


Figure 12: Results of basic HTTP requests (EN) on the same computer

Here is the HTTP request output printed on the command line:



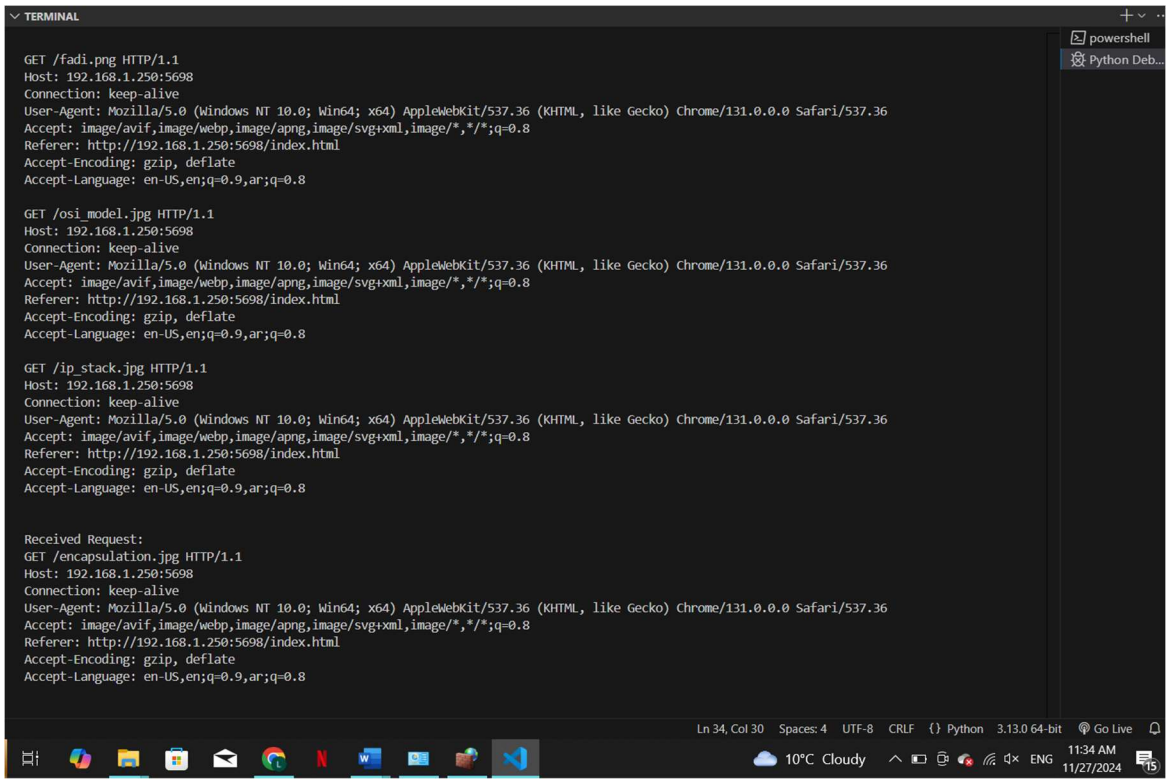


Figure 13: HTTP (EN) request output printed on the command line.

Similarly, requests for */ar* or */main_ar.html* are correctly mapped to the Arabic version of the web page, main_ar.html.

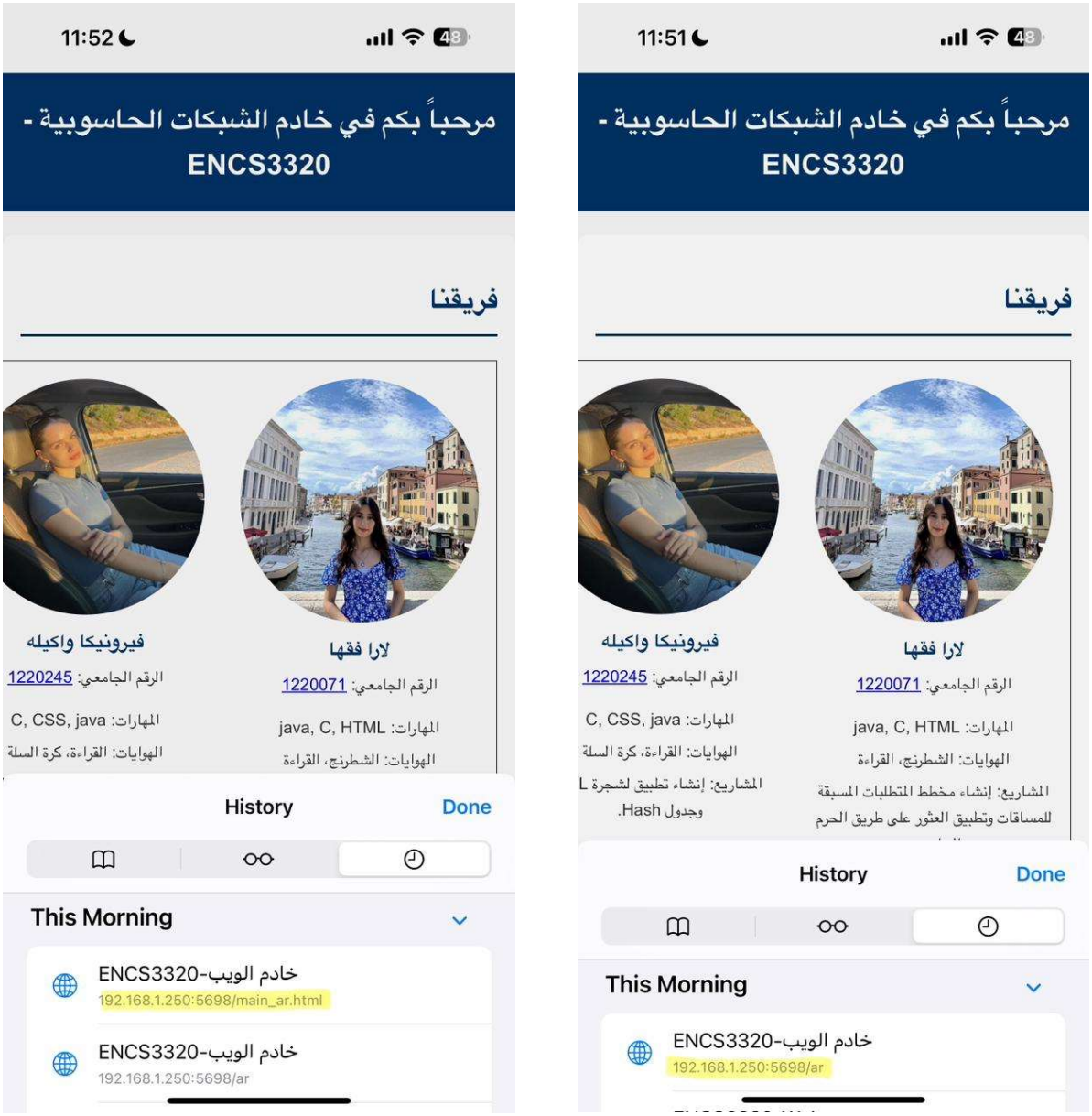


Figure 14: Results of basic HTTP requests (AR) from an external device

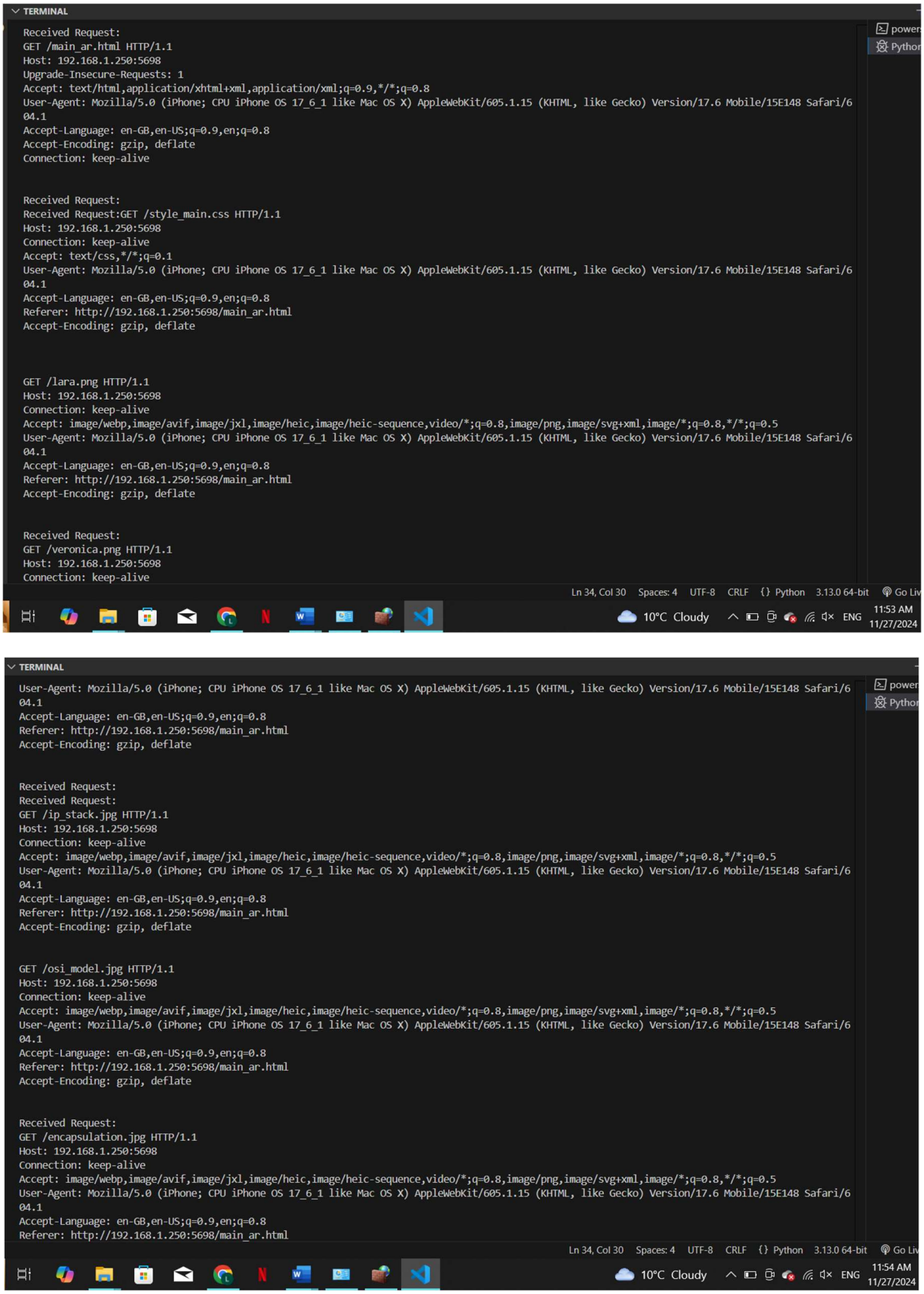


Figure 15: HTTP (AR) request output printed on the command line

The previous results occurred due to the below segment of our code, where each client request was parsed to determine the requested path and file type. Depending on the request, the server performed file serving, error responses, or redirection.

```
request_lines = request.split("\r\n")
request_line = request_lines[0] # the first line contains method and path
method, path, _ = request_line.split(" ",2) # saving the first part in method and the second in path

if (path == "/" or path == "/en" or path == "/main_en.html" or path == "/index.html"):
    path = "main_en.html"
elif (path == "/ar" or path == "/main_ar.html"):
    path = "main_ar.html"
```

Figure 16: Segment 1 of task 2 code

- **File retrieval:** When the client accesses the (supporting material) link, the server retrieves the requested file from the local file system and determines its type using the *get_content_type* function. It then constructs a response header containing the status code, content type, and content length before transmitting the file to the client.

Here is an example of a client requesting a local file (image .jpg or .png) from the English version:

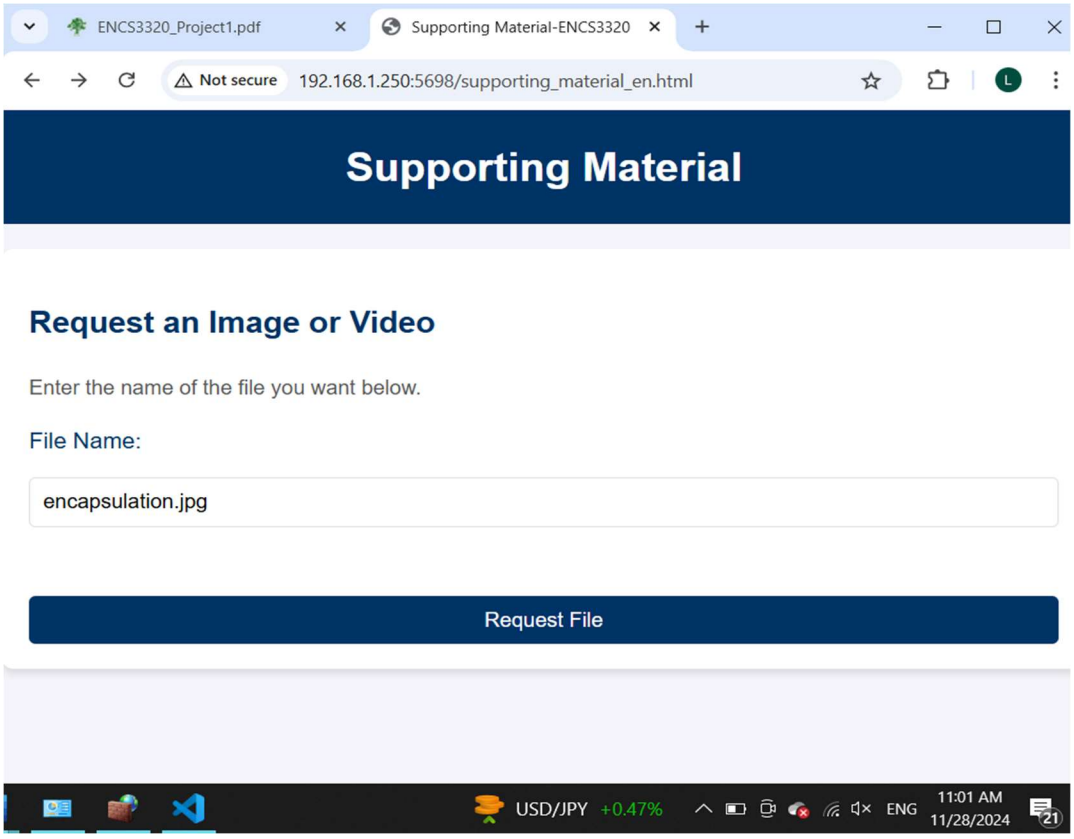


Figure 17: Requesting a local file (image .jpg or .png) (EN)

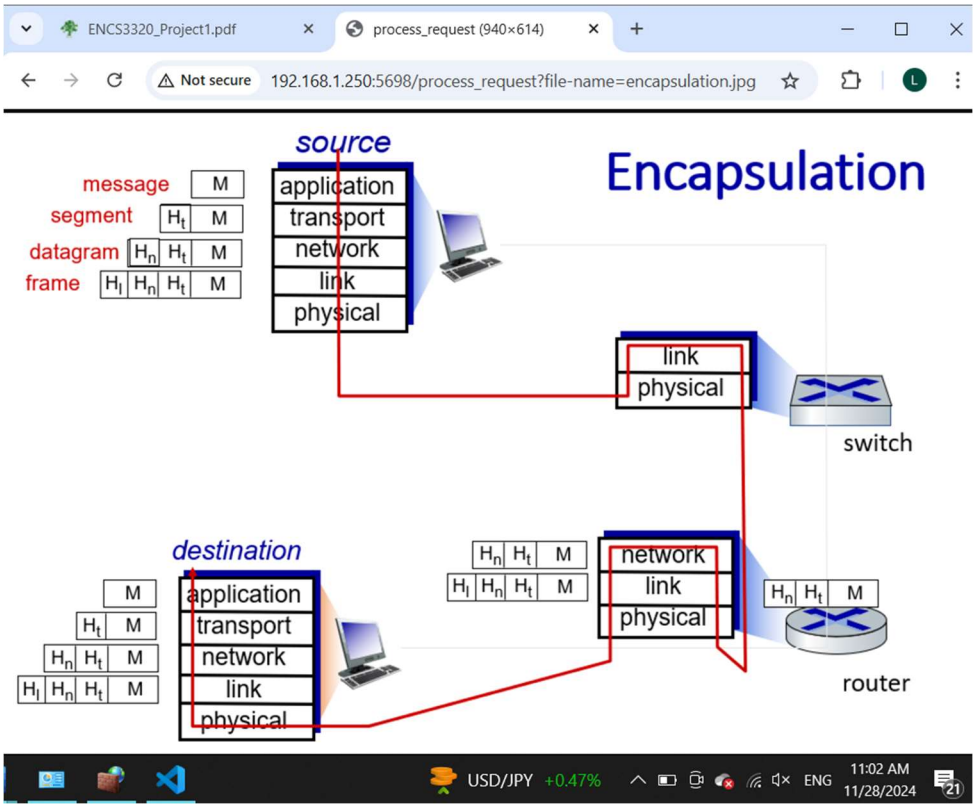


Figure 18: Result of requesting a local file (image .jpg or .png)

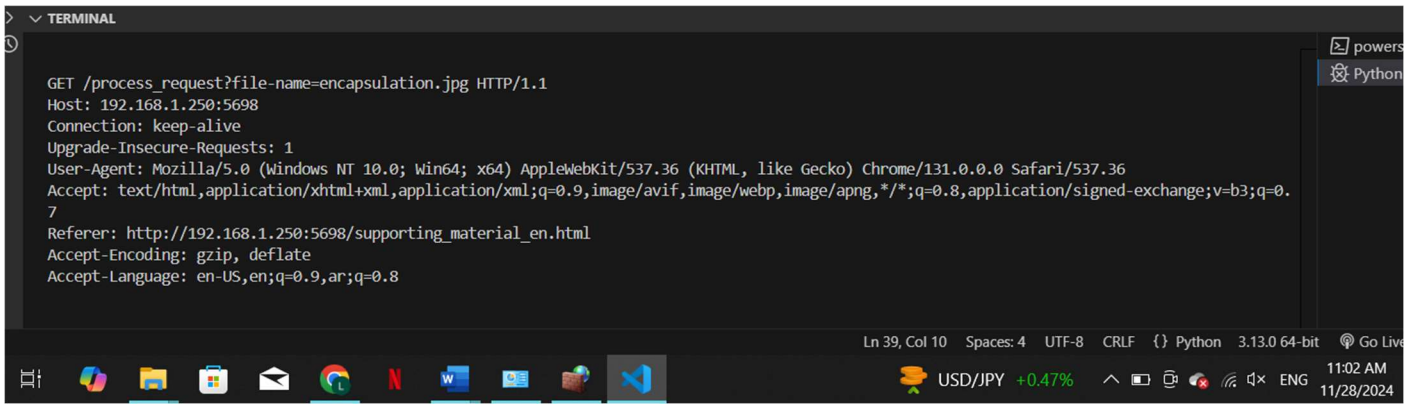


Figure 19: local file (image .jpg or .png) request printed on the command line

Here is an example of a client requesting a local file (image jpg or .png) from the Arabic version from an external device:

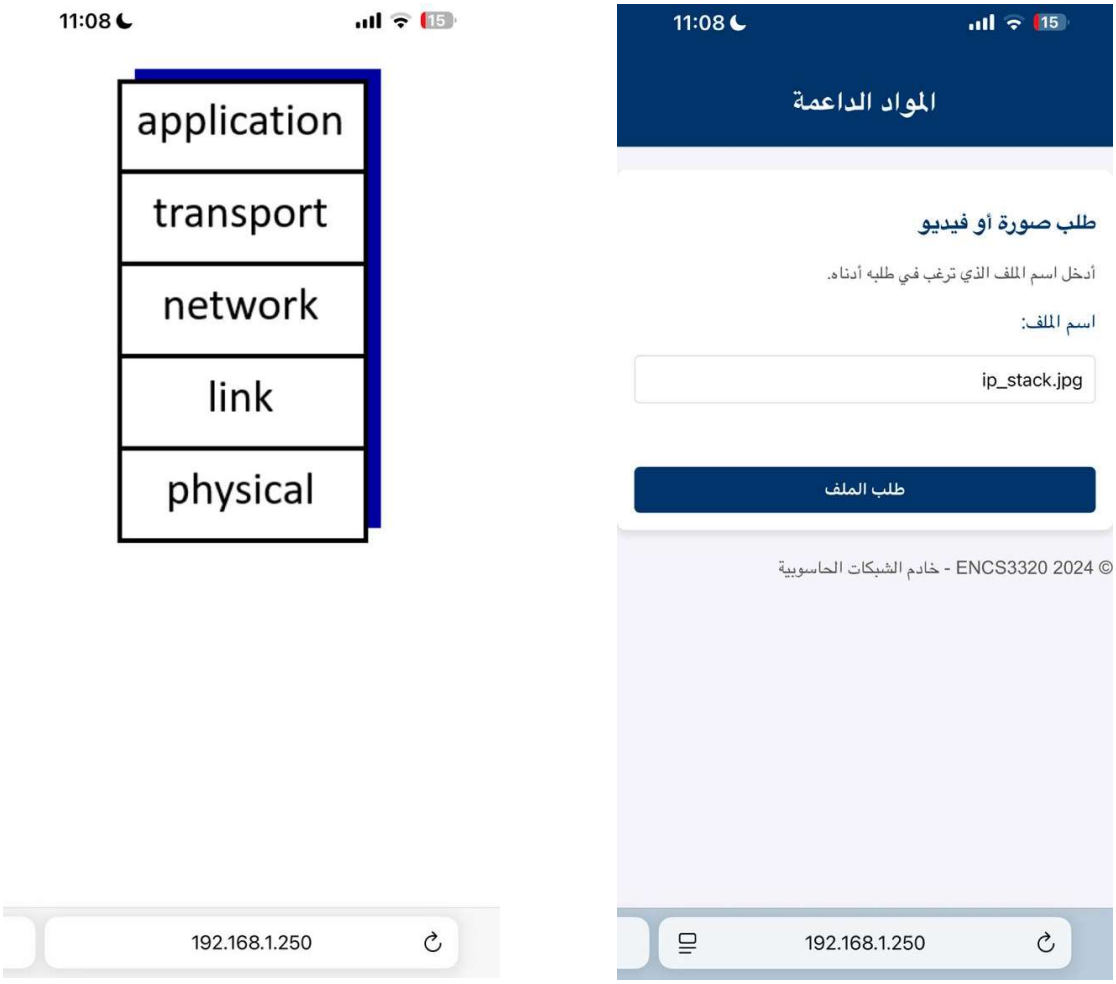


Figure 20: Requesting a local file (image .jpg or .png) (AR)

Here is an example of a client requesting a local file (video .mp4) from the English version:

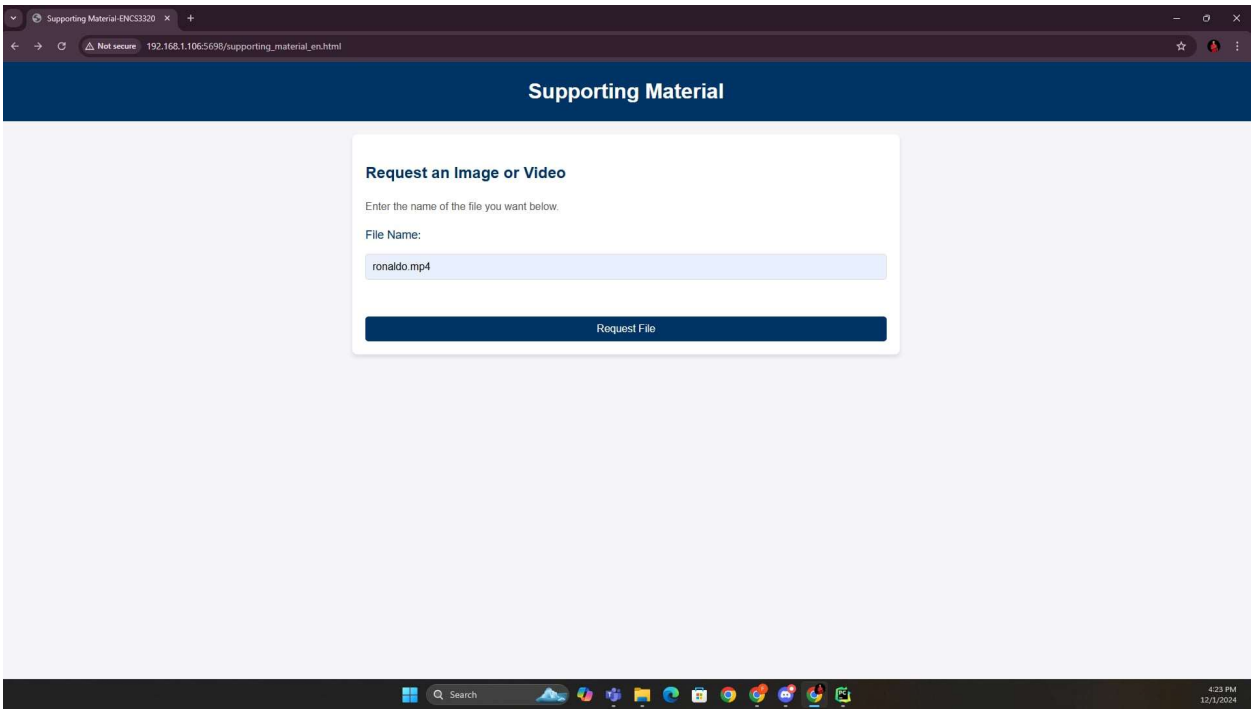


Figure 24: Requesting a local file (Video .mp4) (EN)

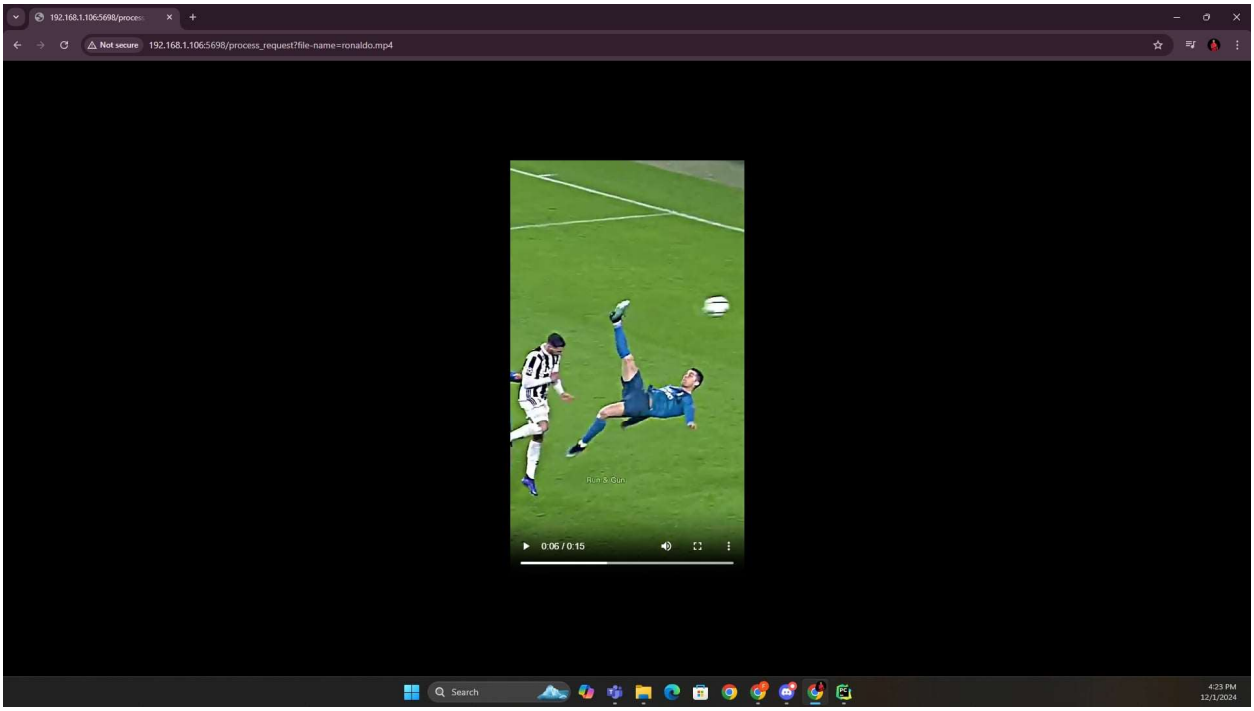


Figure 52: Result of requesting a local file (video .mp4)

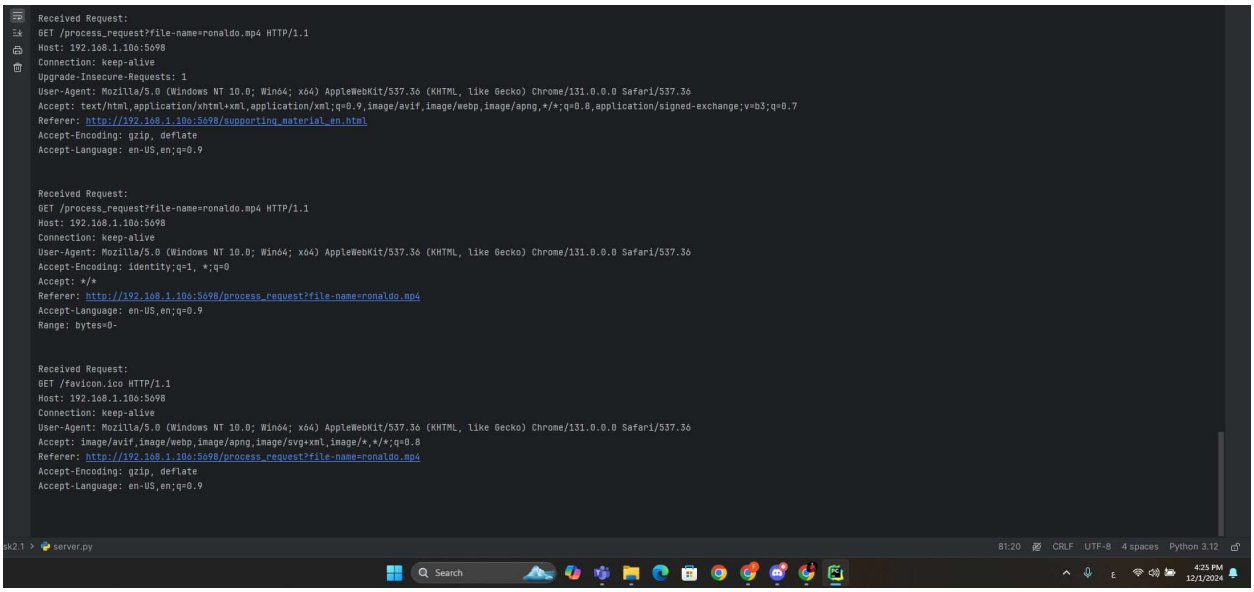


Figure 26: local file (video .mp4) request printed on the command line

The previous results are obtained due to this part of our code:

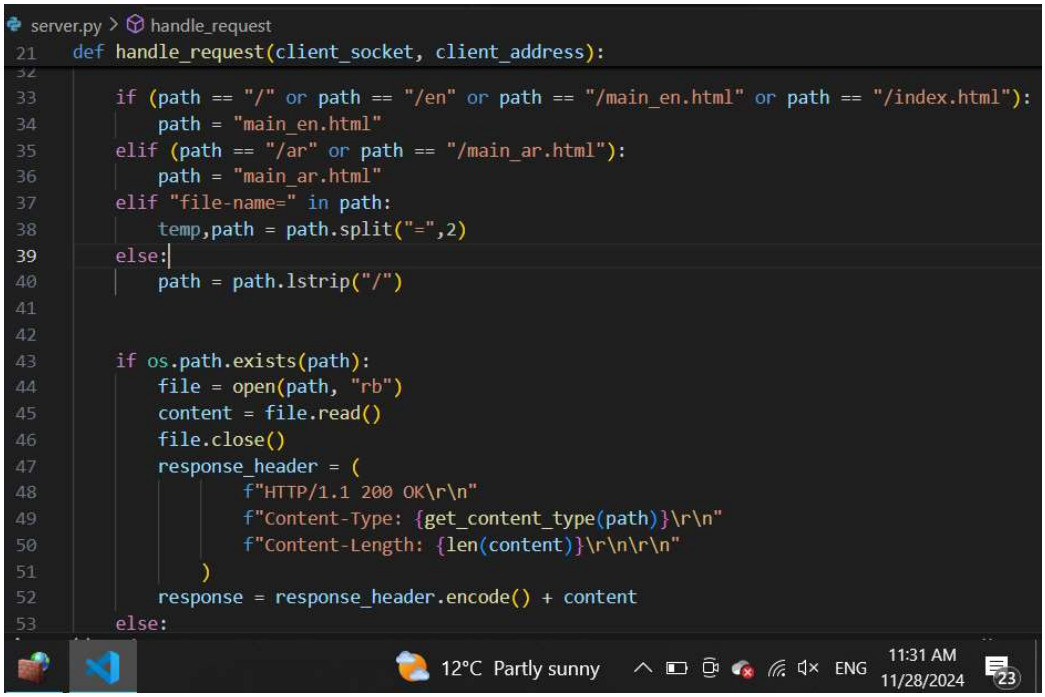


Figure 24: Segment 2 of task 2 code

In Figure 20, the variable “temp” in line 38 was declared but not used. This was done for splitting and extracting the path reasons. For example, if this was the requested URL *http://192.168.1.250:5698/process_request?file-name=encapsulation.jpg* , lines 37 and 38 are executed. The code then checks if the specified file exists in the extracted path.

And this is our *get_content_type* function:

```
6  def get_content_type(file_name):
7      #determine the content type based on the file extension
8      if file_name.endswith(".html"):
9          return "text/html"
10     elif file_name.endswith(".css"):
11         return "text/css"
12     elif file_name.endswith(".png"):
13         return "image/png"
14     elif file_name.endswith(".jpg"):
15         return "image/jpeg"
16     elif file_name.endswith(".mp4"):
17         return "video/mp4"
18     return "application/octet-stream" #default
```

Figure 25: Segment 3 of task 2 code

- **Error handling:** For invalid or unavailable file paths, the server returns a (404 Not Found) error page. This page is dynamically updated to include the client's IP address and port, providing a personalized error response. A browser test confirmed that the error page is displayed correctly when requesting nonexistent files.

Here is an example from an external device:

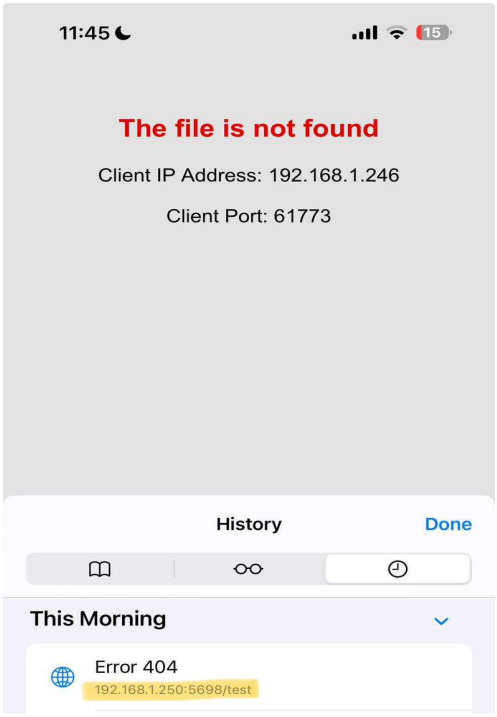


Figure 26: Error handling result on an external device

The output on the command line was:

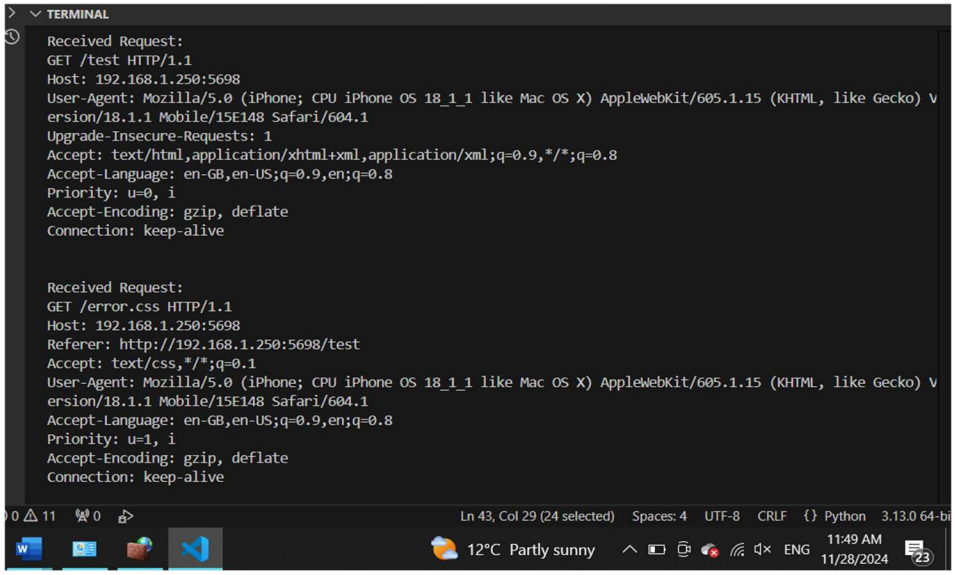


Figure 27: Output of error handling on command line

Another example for requesting invalid material from the *supporting material* page

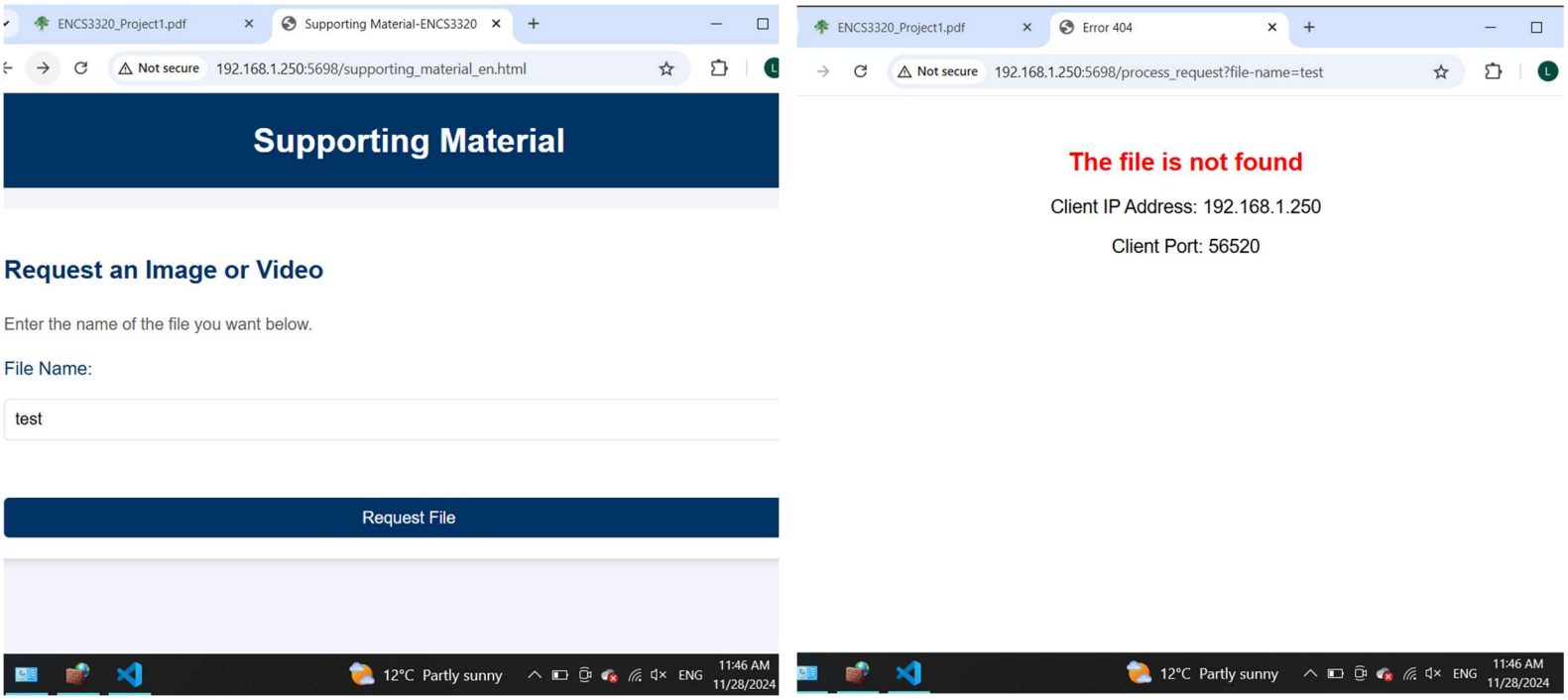


Figure 28: Error handling result for the supporting material page

And this is the part of our code that handles errors:

```
server.py > handle_request
21  def handle_request(client_socket, client_address):
41
42      
43      if os.path.exists(path): ···
53      else:
54          if path.endswith((".jpg",".png")): ···
62
63          elif path.endswith(".mp4"): ···
71
72          else:
73              path = "Error_404.html"
74              file = open(path,"rb")
75              content = file.read()
76              file.close()
77              content = content.replace(b"{CLIENT_IP}", client_ip.encode())
78              content = content.replace(b"{CLIENT_PORT}", str(client_port).encode())
79              response_header = (
80                      "HTTP/1.1 404 Not Found\r\n"
81                      f"Content-Type: {get_content_type(path)}\r\n"
82                      f"Content-Length: {len(content)}\r\n\r\n"
83                  )
84              response = response_header.encode() + content
```


Figure 29: Segment 4 of task 2 code

When the requested path is invalid or unavailable, the server displays the *Error_404.html* page, but replaces the "CLIENT_IP" and "CLIENT_PORT" in the html page with the actual encoded client IP address and port number. Then, the response header is formed with a status code 404, encoded, and sent to the client with the content page.

The client IP address and port number are passed as an argument to the *handle_request* function as below:

```
21  def handle_request(client_socket, client_address):
22      client_ip = client_address[0]  # client IP
23      client_port = client_address[1]  # client port
```

Video highlight ENG 12:26 PM 11/28/2024

Figure 30: Segment 5 of task 2 code

- **Redirection for images and videos:**

- When clients request unavailable image files (.jpg or .png), the server redirects them to a Google Images search URL with a (307 Temporary Redirect) response. For example, a request for *http protocol.png* resulted in a redirect to *https://www.google.com/search?q=http+protocol&udm=2*.

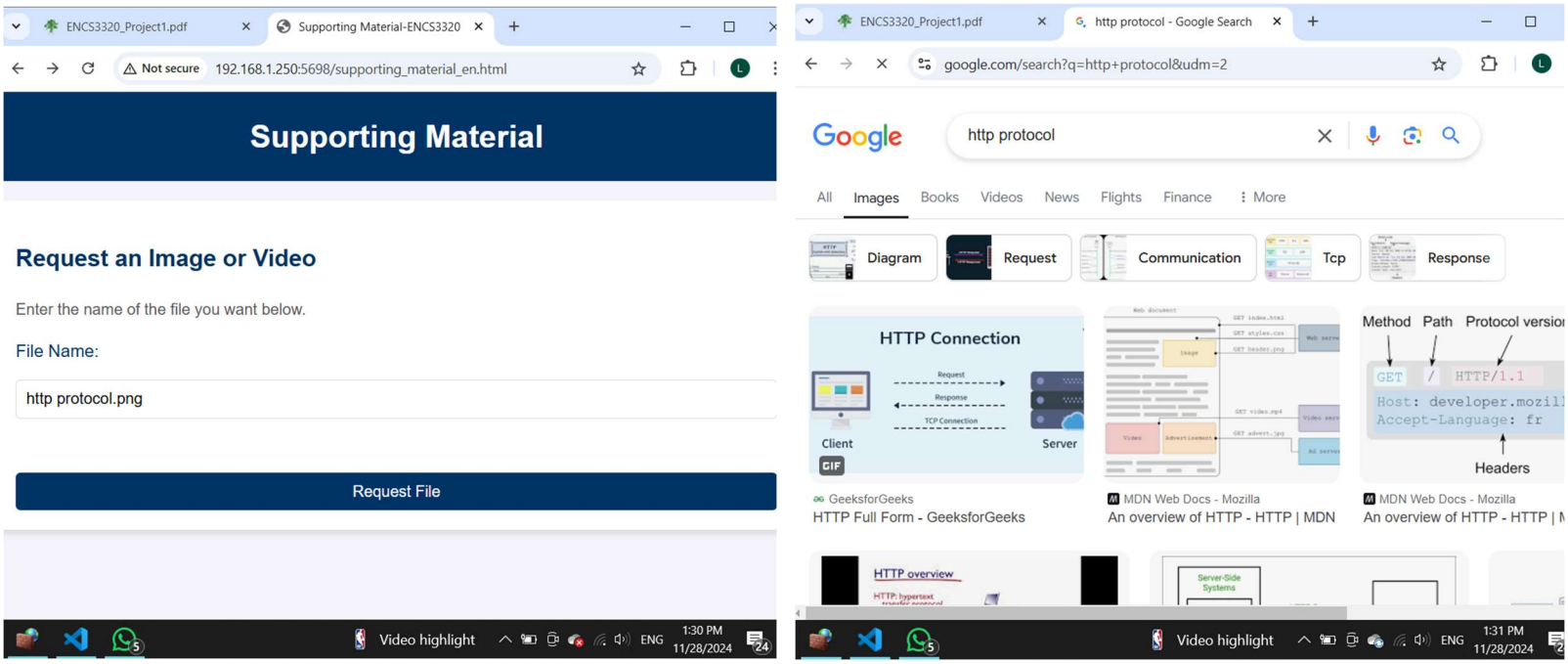


Figure 31: Image redirection

The output on the terminal is:

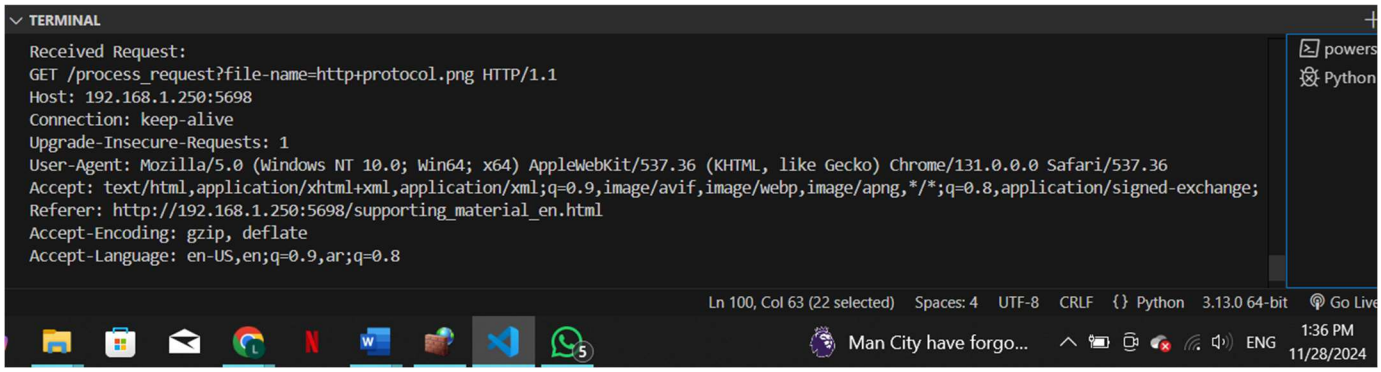
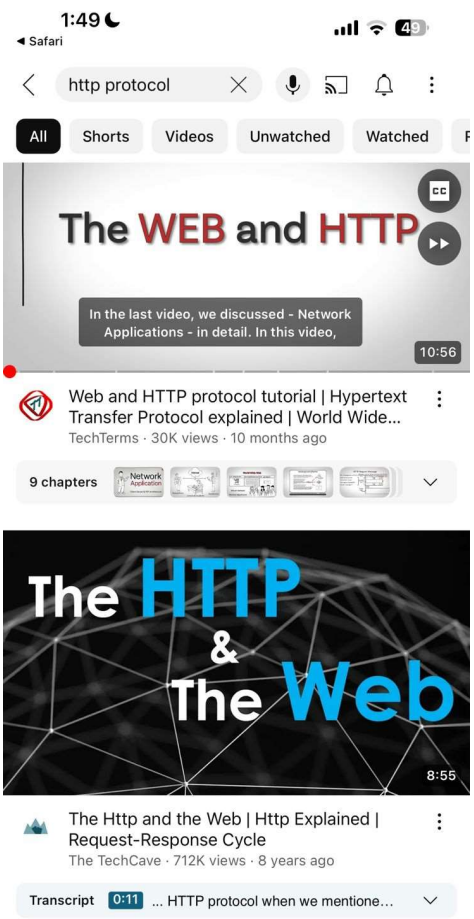


Figure 32: Image redirection output on the command line

- For unavailable video files (.mp4), the server redirects clients to a YouTube search URL with the query based on the file name. For example, a request for *http protocol.mp4* resulted in a redirect to *https://www.youtube.com/results?search_query=http+protocol*+



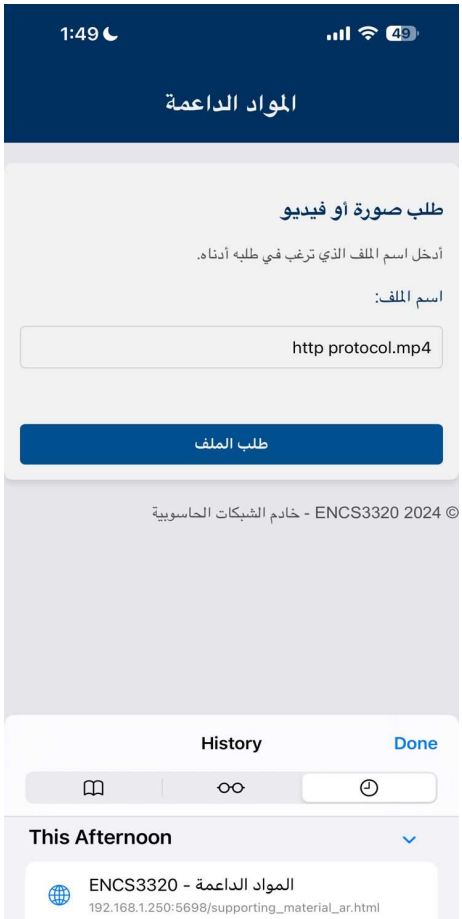


Figure 33: Video redirection

The output on the command line:

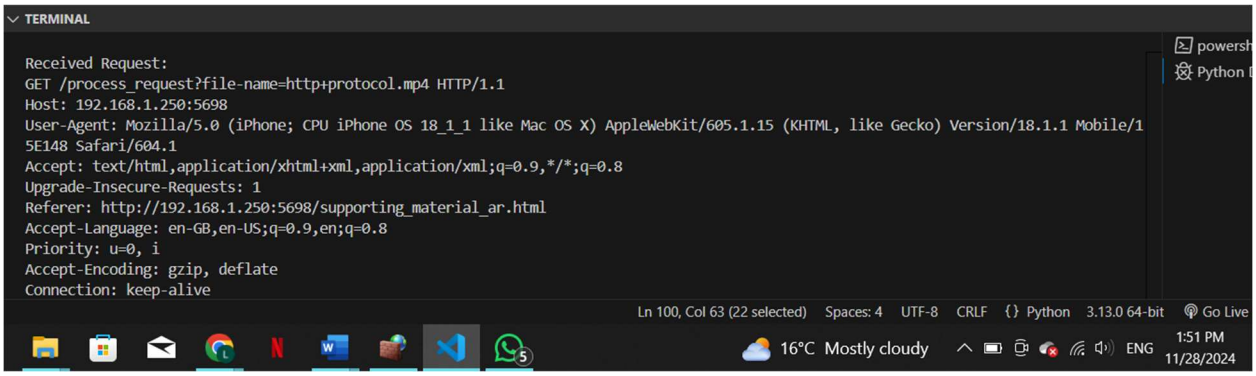


Figure 34: Video redirection output on the command line

This temporary redirection happens as a result of this part of our code:

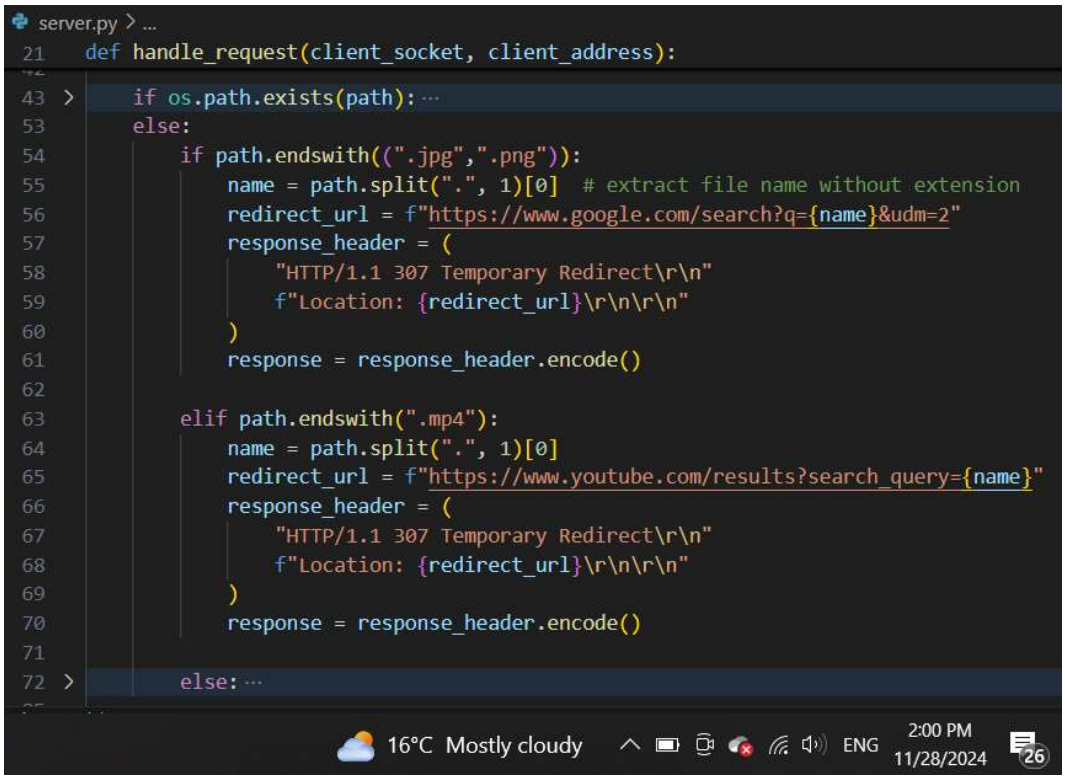


Figure 35: Segment 6 of task 2 code

If the requested file is an image (.jpg or .png) that is unavailable on the server, the server extracts the base name (query) of the file (excluding the extension) and redirects the client to a Google Images search for that query. The Google redirection URL has the following format: *https://www.google.com/search?q=<query>&udm=2* . Where the base URL *https://www.google.com/search* specifies the Google search endpoint, which processes queries for various types of content, including images. The q parameter is assigned the name of the missing file. For example, a request for *test.png* generates the query q=test, which ensures that Google searches for content related to the file name. the udm=2 parameter filters Google search results to images. Including this parameter ensures that clients are directed to pictures.

Similarly, if the requested file is a video (.mp4), the server redirects the client to a YouTube search for the video's name. The YouTube redirection URL has the following format: *https://www.youtube.com/results?search_query=<query>* . Where the base URL *https://www.youtube.com/results* specifies the YouTube search endpoint, which returns results based on the provided search query. The file name is included in the *search_query* parameter. For example, a request for test.mp4 generates the query *search_query=test*, ensuring that the client is redirected to relevant video content.
These redirections are achieved by constructing an HTTP 307 (Temporary Redirect) response with a Location header pointing to the respective search URL. Then this response header is encoded and sent to the client as the response.

- **Concurrent connections:** The multithreaded server handled multiple simultaneous requests from different clients without blocking. Each client connection was managed in a separate thread, ensuring responsiveness and reliability. The server allows a maximum number of 20 connections (can be changed based on the defined variable) at the same time.

  The multithreaded architecture allowed the server to manage concurrent requests effectively. This is essential in real-world scenarios where multiple clients might request resources simultaneously. The use of unique source ports ensured proper identification and handling of individual client sessions.

Below is an example of 3 different devices using the server at the same time. One is browsing the main page, while the other two are requesting files.

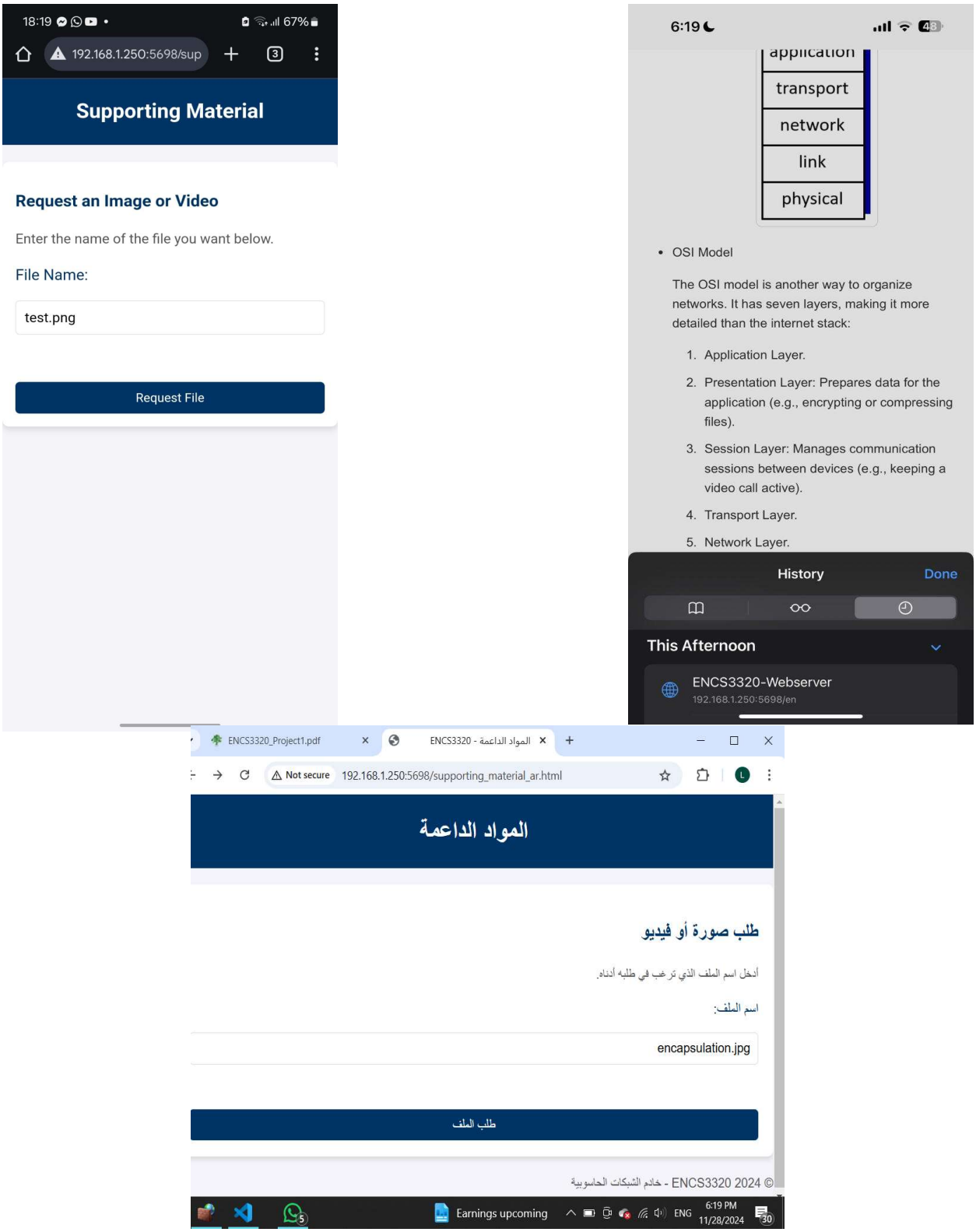


Figure 36: Multiple devices connecting to the server at the same time

This part of the code handles different clients at the same time:

```
try:
    while True:
        # Handle the TCP connection request from Upcoming Clients
        client_socket, client_address = server_socket.accept()
        # each client has a different thread
        client_thread = threading.Thread(target=handle_request, args=(client_socket, client_address))
        client_thread.start()
except KeyboardInterrupt:
    print("Shutting down the server...")
    server_socket.close()
```

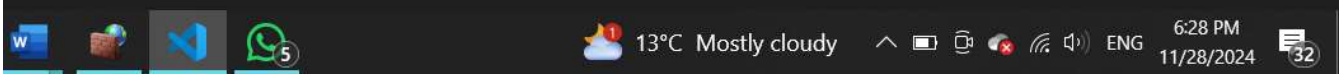


Figure 37: Segment 7 of task 2 code

Figure 37 shows how the server can always receive incoming client connections since it is in an infinite loop. We used *server_socket.accept()* to accept a connection from a client, getting the client_socket (for communication) and the client_address (IP and port). For each new connection, it creates a new thread using *threading.Thread*, with the *handle_request* function assigned to process the client's request, passing the client's socket and address as arguments. This allows the server to handle multiple clients simultaneously by running each client's interaction in a separate thread. The try block ensures the server runs until interrupted. When a KeyboardInterrupt (like Ctrl+C) is detected, the except block shuts down the server by closing the socket and printing a shutdown message.

2.3 Task 3: UDP Client-Server Trivia Game Using Socket Programming

- **Client – server communication**: The fundamental functionality of the client-server communication was successfully achieved. The client was able to connect to the server and receive game messages, such as questions and notifications about the game status. The server broadcasted the questions to all connected clients and processed their responses in real-time.

The figures below show the successful client – server communication:

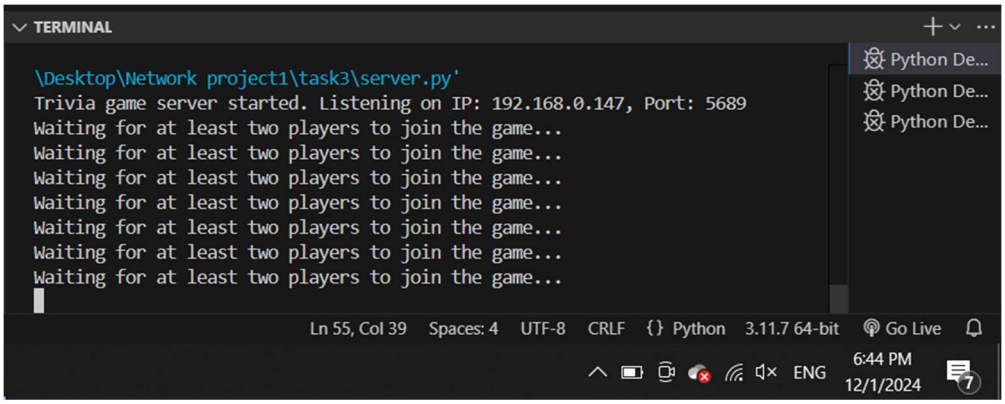


Figure 38: Server's terminal (1)

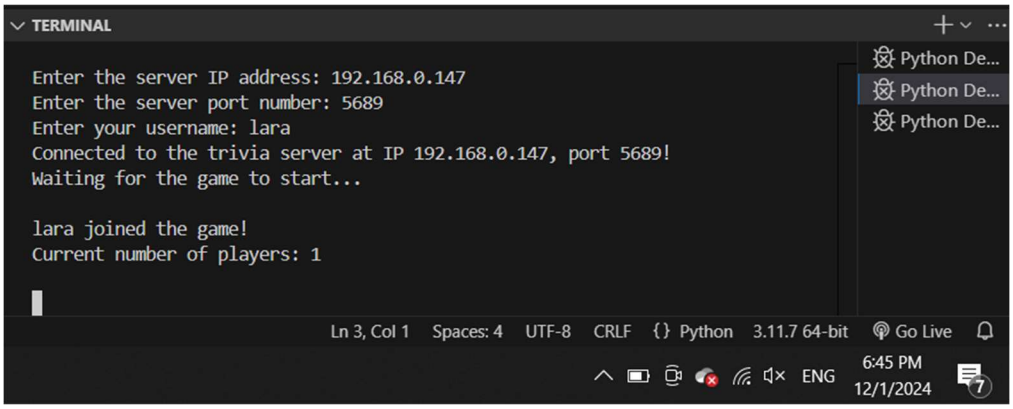


Figure 39: first client joining the game

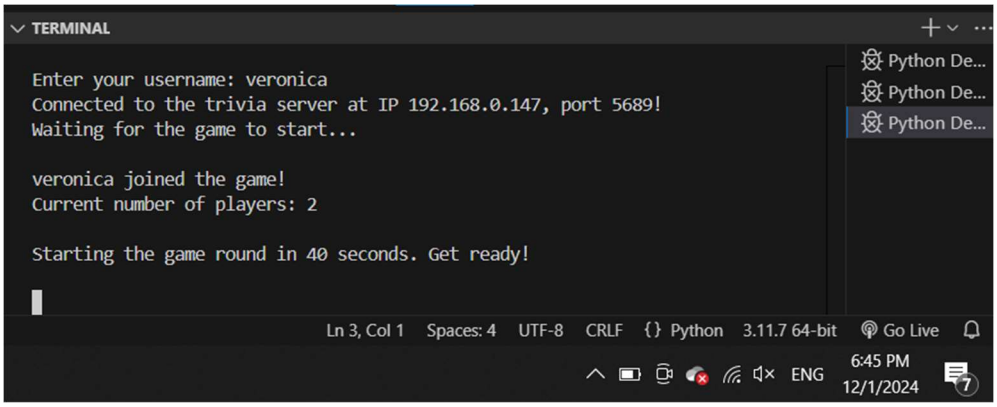


Figure 40: second client joining the game

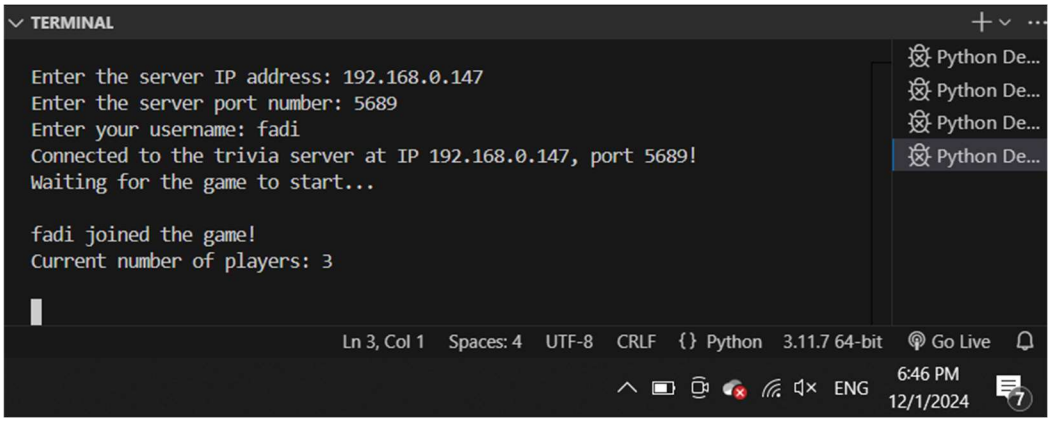


Figure 41: Third player joining the game

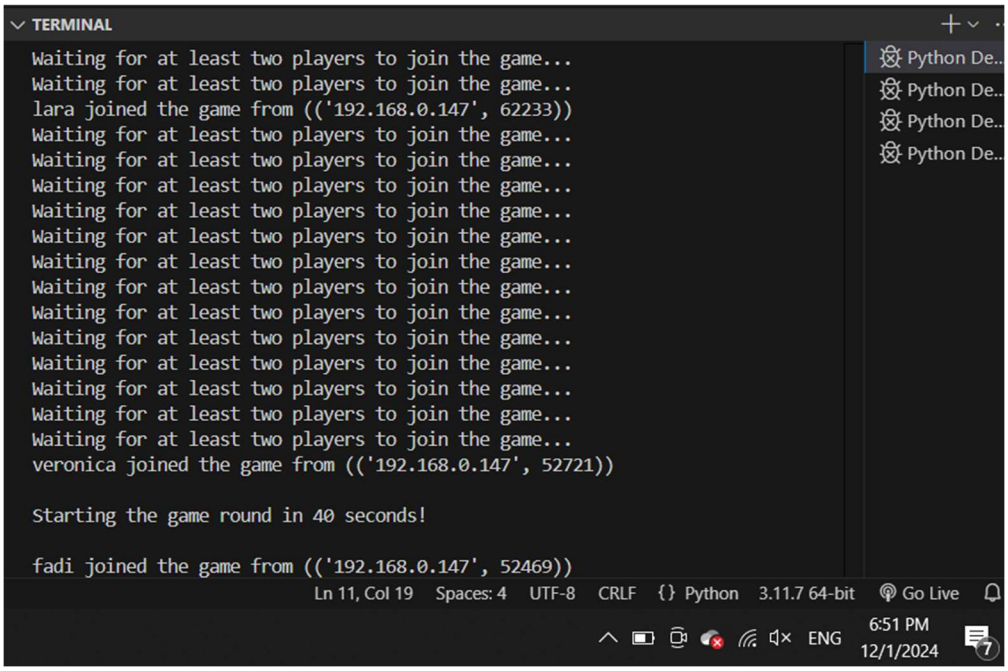


Figure 42: Server's terminal (2)

- **Client response handling**: The client could input answers correctly when the question was presented, and responses were transmitted back to the server without issues. This allowed the game to progress smoothly with players answering questions.

This figure shows the smooth progress of the game with players answering questions:

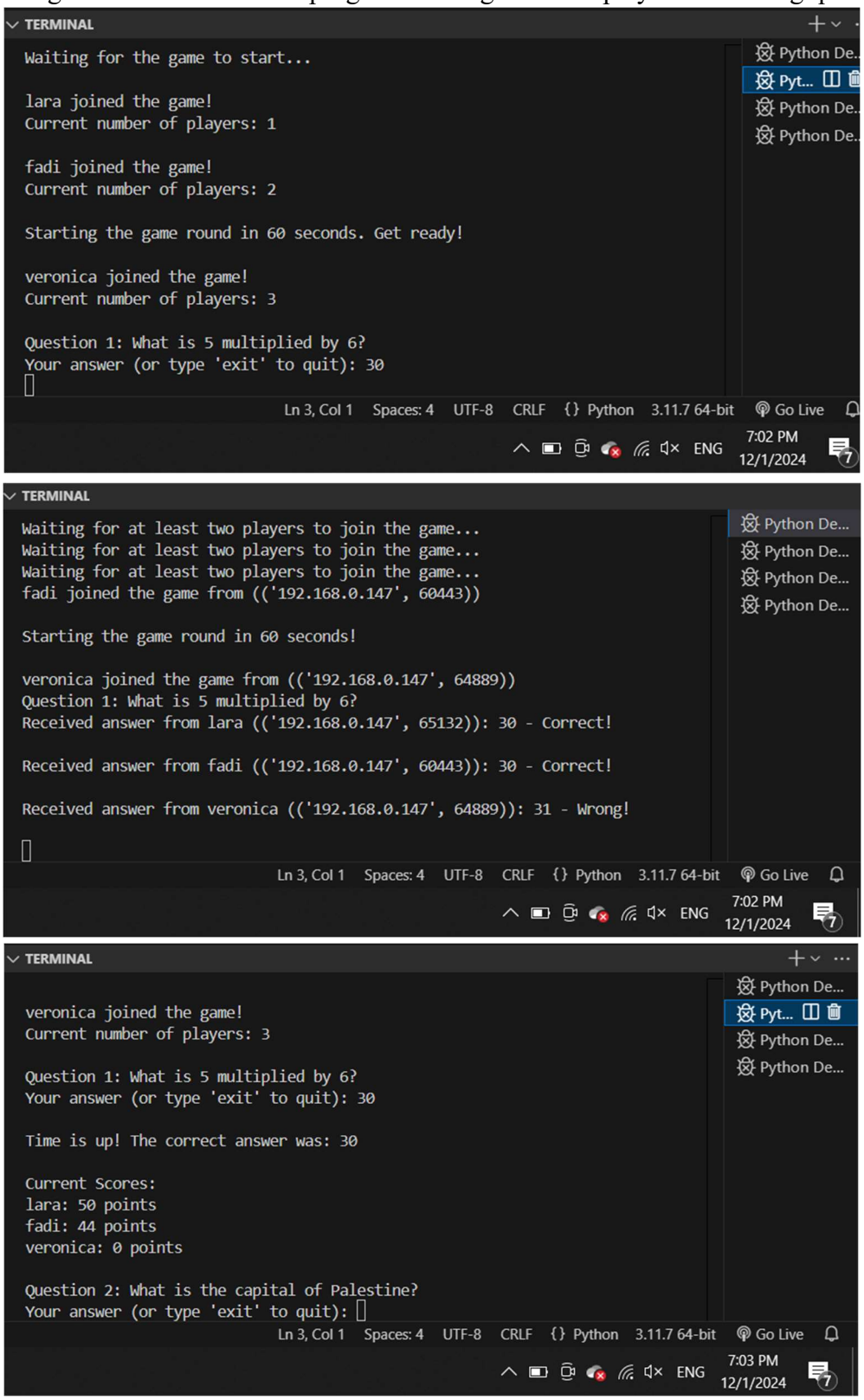


Figure 43: progress of the game with players answering questions

After each round (3 questions) end, the winner with the most achieved points is announced and another round of 3 questions starts after a specified amount of time (here it is 60 seconds)

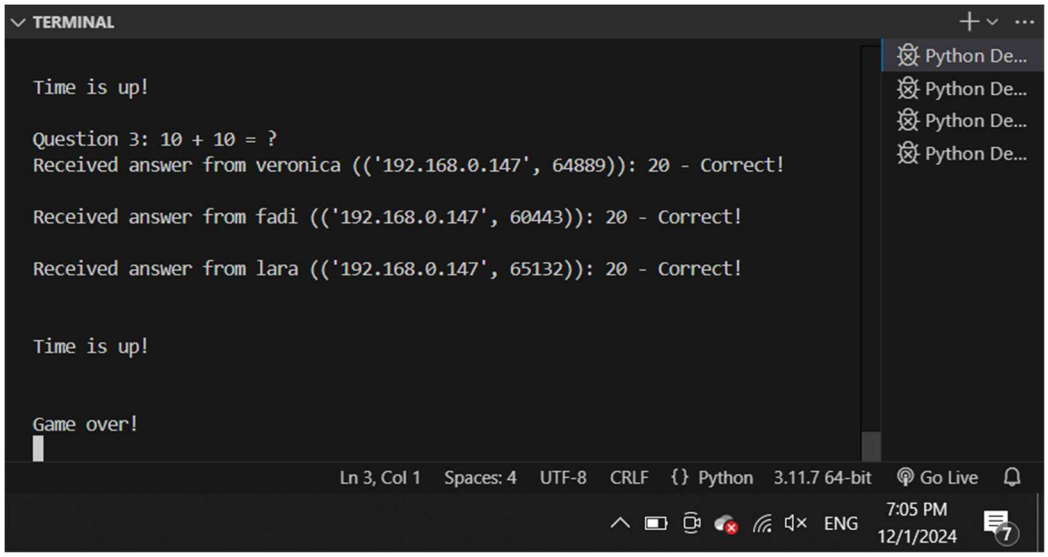


Figure 44: Server's terminal after each round

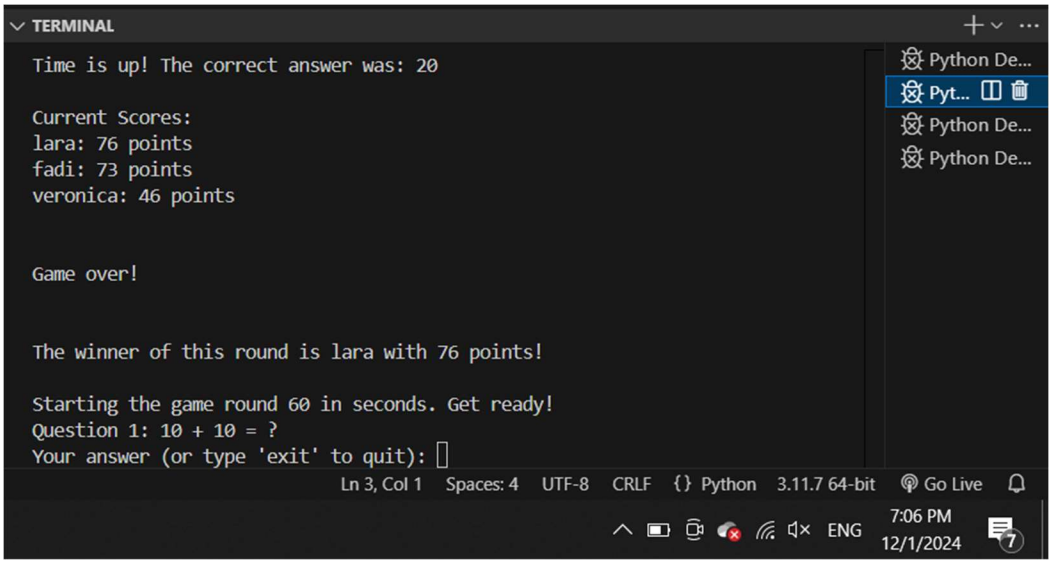


Figure 45: Announcing the round winner to the players

- **Handling of 'exit' command**: When a player typed the "exit" command, the server correctly removed the client from the active list. However, the handling of this event caused an issue with the *ConnectionResetError* on the server side, which occurs when a client disconnects abruptly. This indicates that the server wasn't properly handling the closure of connections or cleaning up resources when a client exited the game.

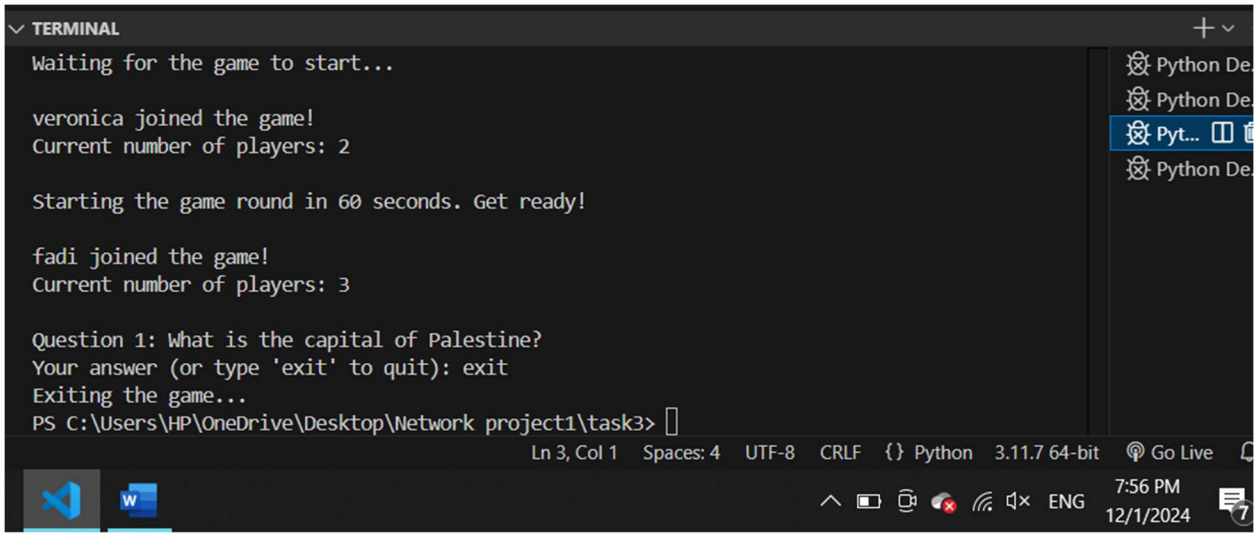


Figure 46: A client exiting the game

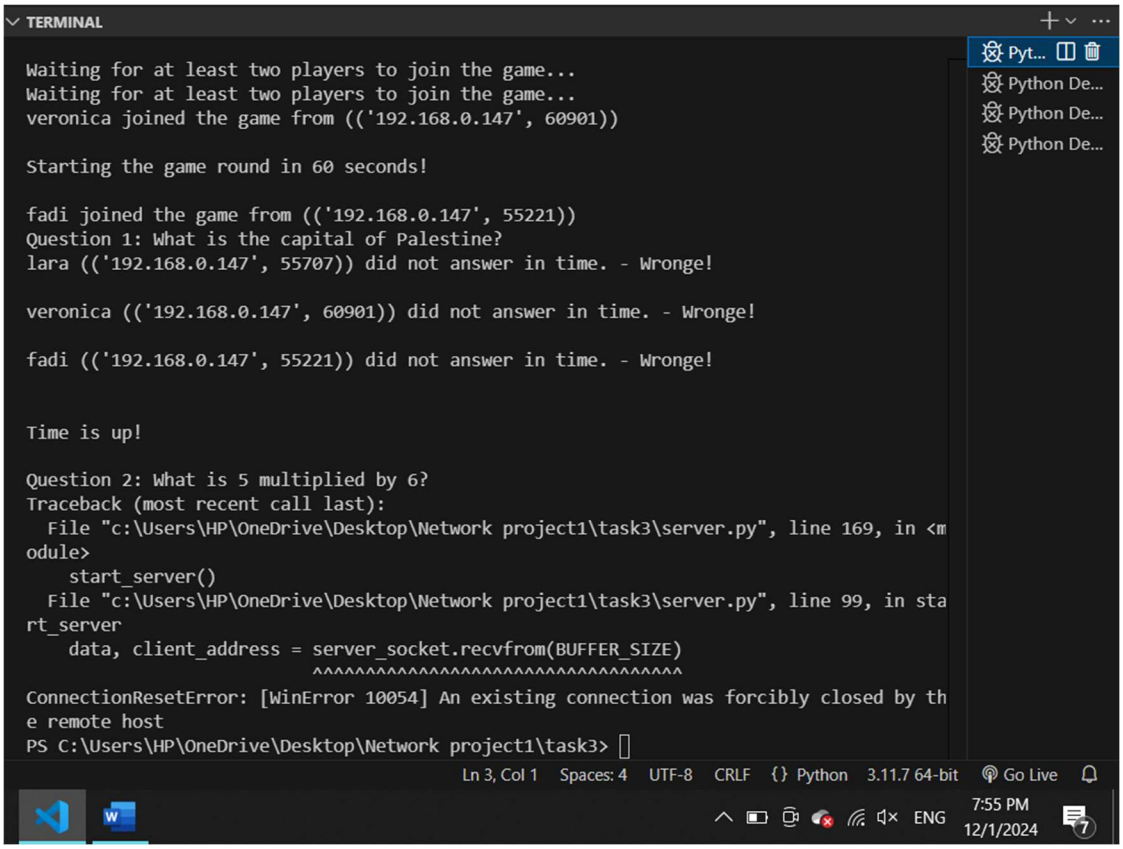


Figure 47: The server's terminal after a client exits the game

    - **Game flow**: The game flow, including the handling of rounds, questions, and client answers, worked as expected
- **Time management**: The server properly handled question time limits, ensuring that each question was answered within the specified time (QUESTION_TIME). The “Time is up!” message was broadcasted when the time limit was reached.
- **Inactive clients**: One significant issue identified was how the server handled inactive clients who failed to answer a question within the allotted time. While the server correctly marks these clients as inactive and removes them after a round, their terminals remain stuck waiting for input, causing confusion for the player. This results from blocking input logic on the client side, which does not account for the server broadcasting "time is up" messages.

In this example, a client is active while the other in inactive:

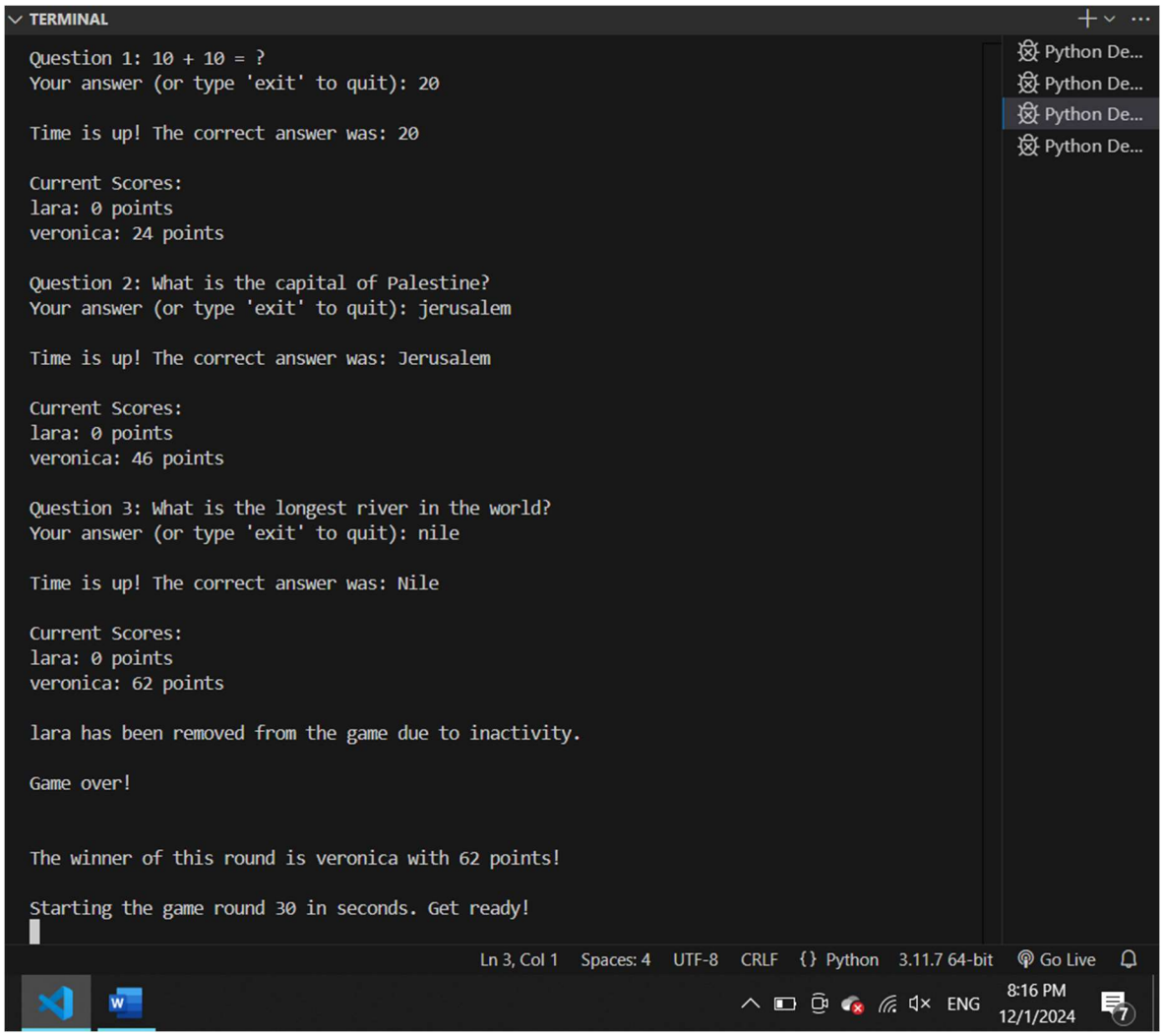


Figure 48: Active client’s terminal

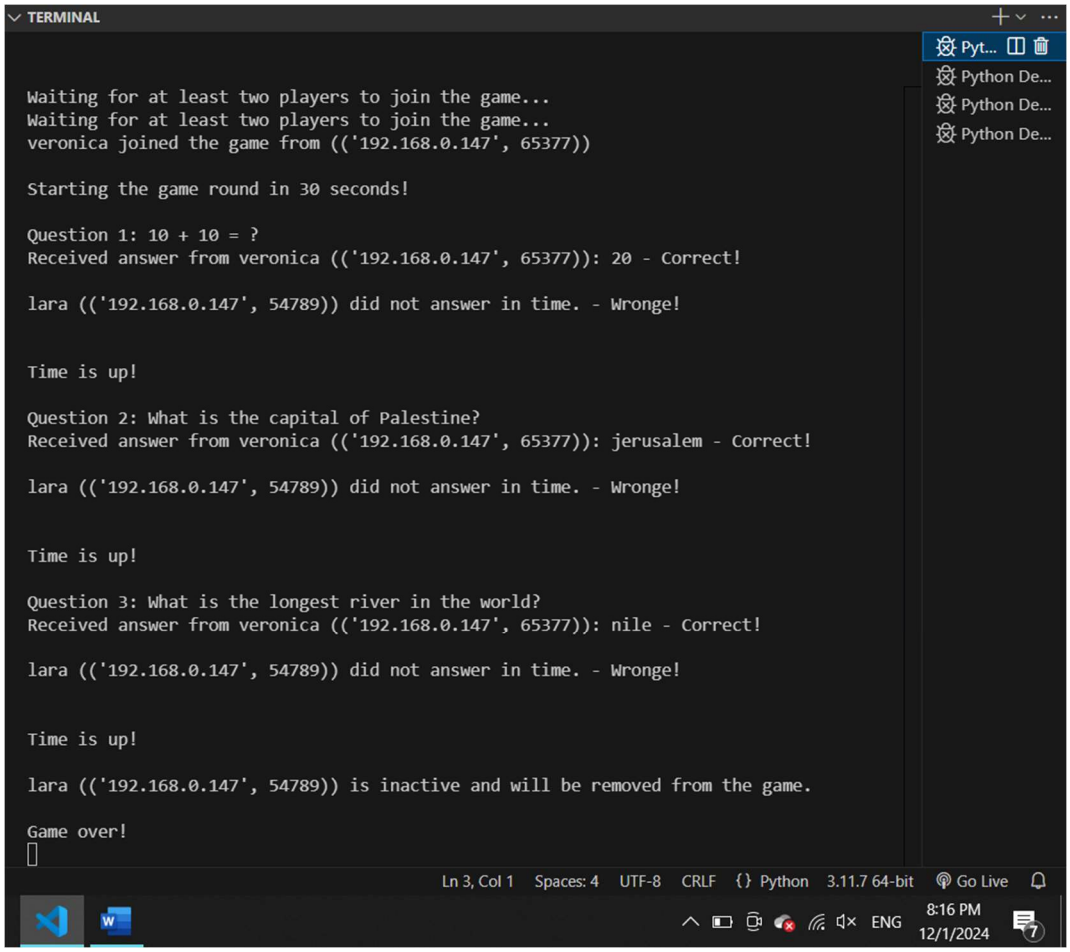


Figure 49: Server's terminal when a client is inactive

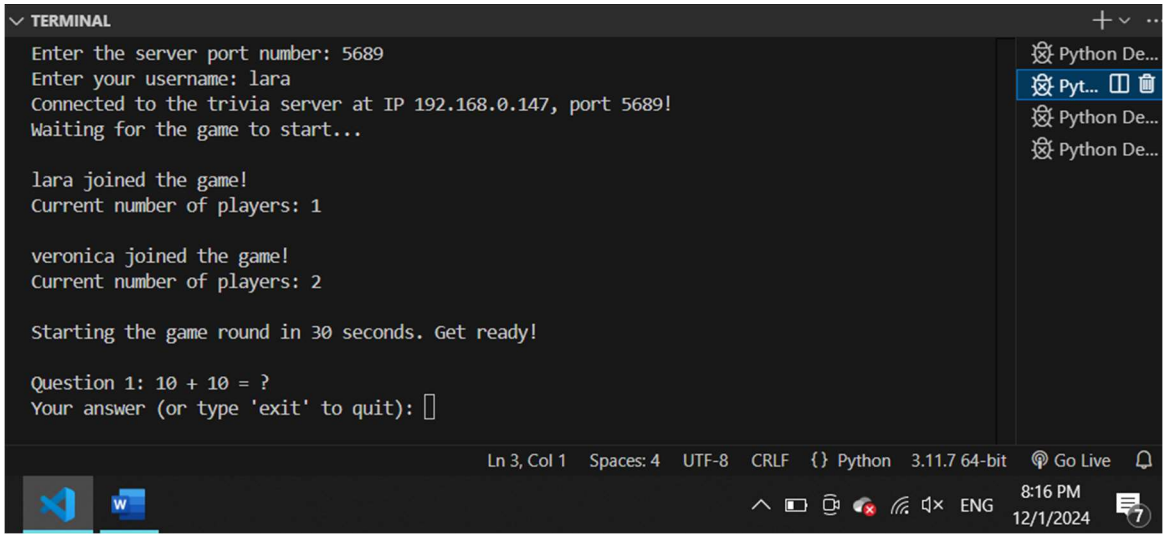


Figure 50: Inactive client's terminal

- **Scalability and performance**:

- **Multiple clients**: The system functions well for a small number of players but encounters challenges as client activity increases. The client-side blocking input and server-side waiting for inactive clients result in delays during gameplay transitions.
- **Time synchronization**: The server's ability to wait for all players or move to the next question after the timer expires is critical for a smooth experience. Current issues with inactive clients disrupt the game flow, as the server must account for both active and inactive players, leading to inefficiencies.

3. Alternative Solutions, Issues, and Limitations

3.1 Task 1: Network Commands and Wireshark

While Wireshark is powerful, its complexity and volume of captured data can overwhelm beginners. Additionally, these tools are limited in scalability and often require administrative privileges, which may not always be available. Another limitation is the inability to analyze encrypted traffic effectively, restricting insights into some network activities. Despite these challenges, the tools provide foundational capabilities for network troubleshooting and analysis.

3.2 Task2: Web Server

- One observed limitation was the inability to serve files larger than the maximum socket buffer size in a single read operation. While this was not critical for this task, it highlights the need for chunked data transfer in more advanced implementations.
- Dynamically generating the error page with client IP and port required careful handling of string replacements.
- Managing threads was challenging, especially ensuring proper cleanup of sockets.
- The server's multithreading approach may not handle very high traffic efficiently.
- Lack of HTTPS support makes communication insecure.
- External device testing was sometimes blocked by network firewalls.

3.3 Task 3: UDP Client-Server Trivia Game Using Socket Programming

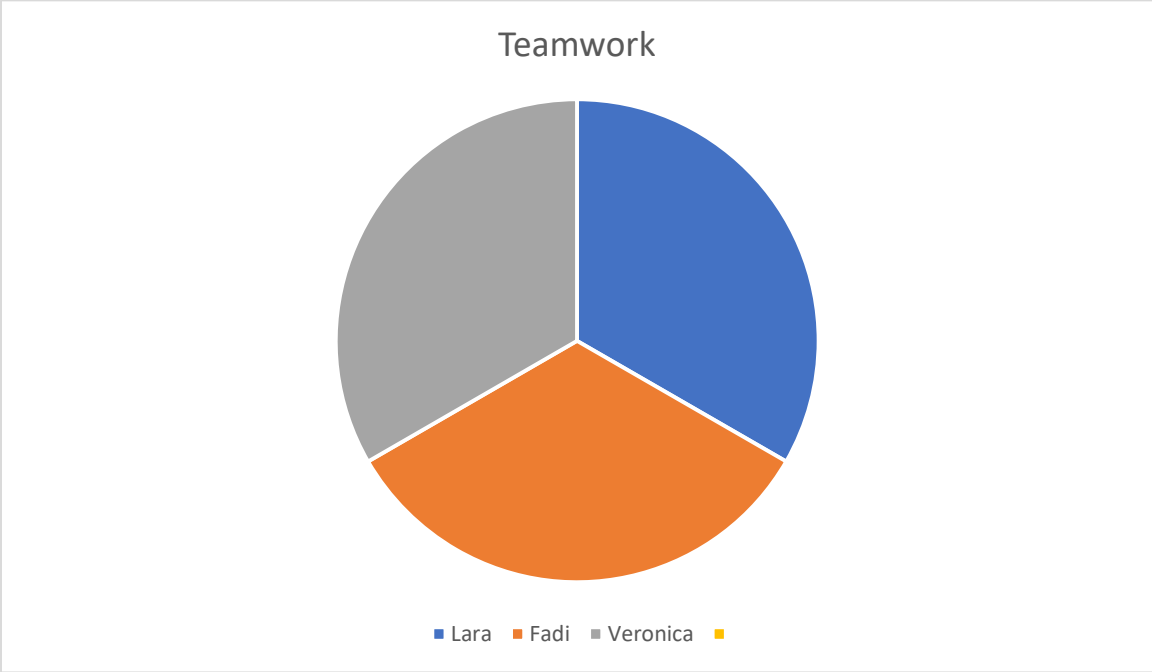
- **UDP Protocol Limitations:** The use of UDP ensures fast communication but does not guarantee message delivery or order. When packets are dropped, it affects game synchronization. In scenarios where clients fail to respond or leave abruptly, game progression can become inconsistent.
- **Blocking Input:** On the client side, the use of blocking input prevents the game from progressing smoothly. When clients do not answer, their terminals remain stuck until an input is provided, even if the server has moved on to the next question.
- The trivia game project successfully demonstrates client-server communication with UDP sockets, allowing multiple users to participate in a shared game environment. However, improvements are needed for better synchronization, user experience, and scalability:

1) **Handling inactive clients:** Implementing a non-blocking input on the client side to ensure the game proceeds seamlessly, even if a client fails to answer within the allotted time.
2) **Improving disconnection handling:** Enhancing server-side logic to manage unexpected client exits, ensuring the game flow remains uninterrupted for active players.
3) **Error resilience:** Improving error handling for scenarios involving packet loss or unexpected client behavior.

These adjustments might refine the trivia game, addressing the current challenges and enhancing it. If our time wasn't limited, maybe we could have succeeded at fixing these issues.

In general, coding with python was a bit challenging at the beginning since we were not familiar with the language's syntax and exception handlings. But we tried to learn it and gradually became more comfortable with it through practice and research. Initially, understanding how to manage sockets, handle network protocols, and debug the server-client communication was a bit tricky. However, we leveraged resources such as online tutorials, Python documentation, and forums to help troubleshoot and refine our approach. As we worked through the process, we also encountered issues like handling timeouts, ensuring proper synchronization between the server and multiple clients, and addressing exceptions properly. Despite these challenges, we learned a lot about both Python and networking, and in the end, the experience helped us improve our problem-solving and coding skills significantly.

Teamwork



Each team member contributed equally to all three tasks of the project. We collaboratively researched and implemented network commands and analyzed traffic using Wireshark. For the web server task, we worked together on building and testing the server-client model. In the final task, the trivia game, we equally shared responsibilities in coding the server-client communication and handling user interactions. Our collaboration ensured that each member was involved in every step, from planning and coding to testing and debugging, allowing us to successfully complete all aspects of the project.

Conclusion

this project offered valuable experience in networking, including the use of network commands, Wireshark, and web server creation. The first task taught us how to analyze network traffic and troubleshoot using tools like ping and traceroute, while Wireshark allowed us to capture and inspect packets. The second task involved building a basic web server in Python, enhancing our understanding of server-client communication. Finally, the trivia game server-client system reinforced concepts of UDP communication, client management, and handling user input. Overall, this project helped develop foundational networking skills essential for future applications in network administration and development.

References:

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

[2]: https://www.dell.com/support/kbdoc/en-us/000130703/the-command-prompt-what-it-is-and-how-to-use-it-on-a-dell-system

[3]: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/ipconfig

[4]: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/ping

[5]: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/nslookup

[6]: https://gaia.cs.umass.edu/kurose_ross/index.php

[7]: https://www.haproxy.com/blog/http-keep-alive-pipelining-multiplexing-and-connection-pooling

[8]: https://www.fastly.com/learning/what-is-a-cdn