Computer Networks Programming Assignment 1 Socket Programming (Python)

Name	**ID**
Rowan Gamal Ahmed Elkhouly	21010539
Ranime Ahmed Elsayed Shehata	21010531

Introduction:

- The assignment aims to develop a **multi-threaded** HTTP Client-Server system using Python.
- The system supports the basic HTTP methods `GET` and `POST` and can handle multiple client connections simultaneously.
- The client can request files from the server using `GET` requests and upload files to the server using `POST` requests.
- The server is designed to handle **multiple requests concurrently** through multithreading, ensuring efficient handling of multiple clients.

System Design:

1. Client:

Client_parser.py:

- This script allows users to manually interact with the server by automating the process of sending multiple requests, either **GET** or **POST** requests.
- The client socket establishes a connection to the server and sends the appropriate HTTP request based on the command read from a text file.
- It reads commands from a text file (`requests.txt`) and processes each line, sending either a `GET` or `POST` request based on the instructions in the file.
- The approach we're following is useful for executing multiple requests without manual intervention.

2. Server:

➔ The server listens for incoming client connections and handles both `GET` and `POST` requests.

➔ The server supports concurrent client connections by creating a new thread for each client that connects.

➔ The key responsibilities of the server include:

- **Handling GET Requests**: When a client requests a file using a `GET` request, the server checks if the file exists in the current directory. If the file is found, the server sends the file back to the client as part of the HTTP response. If the file is not found, the server responds with a `404 NotFound` status.

- **Handling POST Requests**: When a client uploads a file using a P0ST request, the server saves the file to its directory. After successfully saving the file, the server sends a confirmation response back to the client.
- **Multithreading**: The server is designed to handle multiple clients simultaneously by creating a separate thread for each incoming connection. This ensures that multiple clients can interact with the server concurrently without blocking other requests and guarantees consistency.
- **Timeout and Inactivity Handling**: The server includes a timeout mechanism to handle client inactivity. If no requests are received from a client within a specified timeout period, the server closes the connection. Additionally, if the server remains idle for an extended period with no client connections, it shuts down automatically.

Connections:

- Persistent connection is made between server and each client
- Each connection runs on a separate thread.
- Each connection has a dynamic timeout set by server, timeout = max(5 seconds, 20 - current-number-of-connections)
- Server itself has a timeout of 60 seconds of inactivity, meaning if server doesn't get any requests for continuous 60 seconds it will shut down

---

Requests:

Clients can send:

- **GET** requests to get a certain file from the server
- **POST** requests to send a file to the server

For **GET** requests, the server will send the file requested by the client.

For **POST** requests, the server will save the file sent by the client.

Responses:

The server can respond with:

- **200 OK** if the request was successful
- **404 Not Found** if the requested file was not found

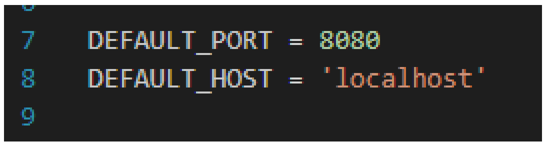
Files Supported:

The server can handle the following files types:

- .txt
- .jpg
- .png
- .html

Implementation Details:

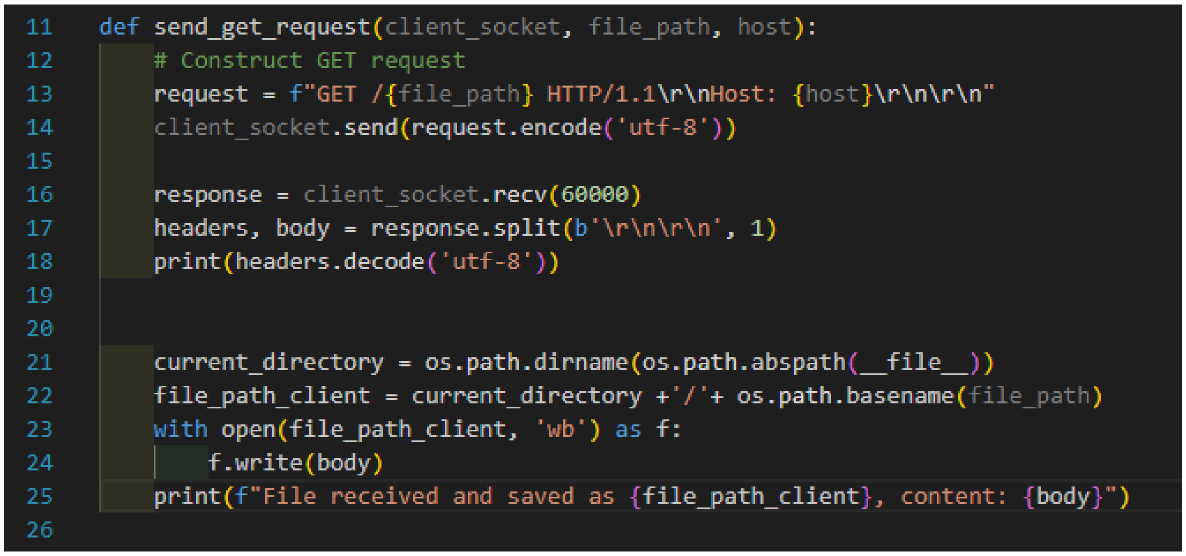
1. client_parser.py:
    - **Default Configurations:**



```
DEFAULT_PORT = 8080
DEFAULT_HOST = 'localhost'
```

These are default values for the port and host used to connect to the server. If not explicitly specified in the requests, the client will use **`localhost`** as the host and port **`8080`**.

- **send_get_request() Function:**



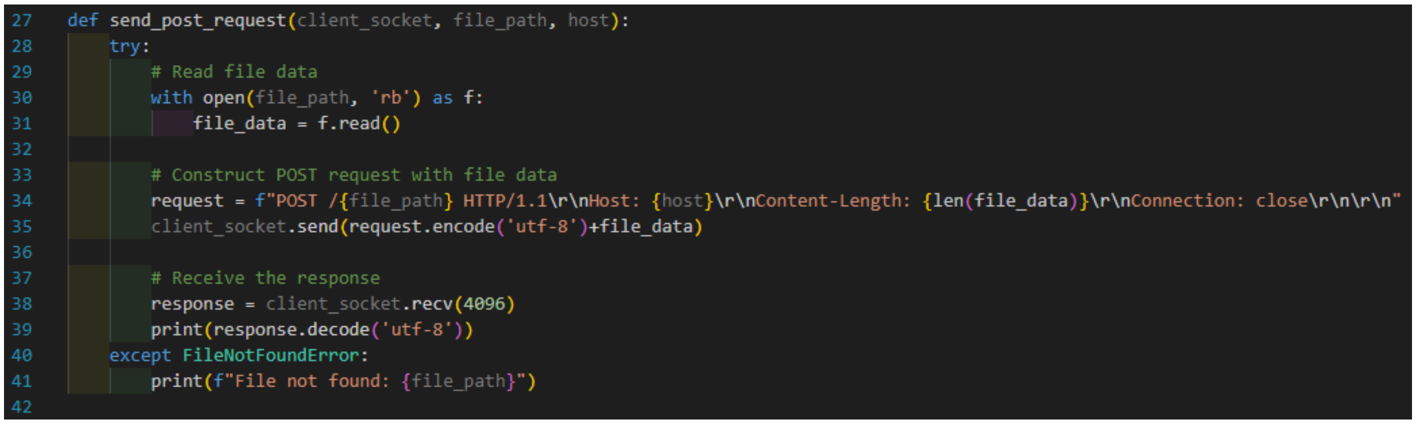
```python
def send_get_request(client_socket, file_path, host):
    # Construct GET request
    request = f"GET /{file_path} HTTP/1.1\r\nHost: {host}\r\n\r\n"
    client_socket.send(request.encode('utf-8'))

    response = client_socket.recv(60000)
    headers, body = response.split(b'\r\n\r\n', 1)
    print(headers.decode('utf-8'))


    current_directory = os.path.dirname(os.path.abspath(__file__))
    file_path_client = current_directory +'/'+ os.path.basename(file_path)
    with open(file_path_client, 'wb') as f:
        f.write(body)
    print(f"File received and saved as {file_path_client}, content: {body}")

```

Purpose: Handles GET requests to download files from the server.

Steps:

- Constructs a GET request using HTTP/1.1 format.
- Sends the request to the server using the socket.
- Receives the response from the server, which contains both the headers and the body (file content).
- The headers and body are separated by a blank line (\r\n\r\n), and the body is written to a file in the client's current directory.
- It saves the file with the same name as requested and prints the saved content.

- **send_post_request() Function:**



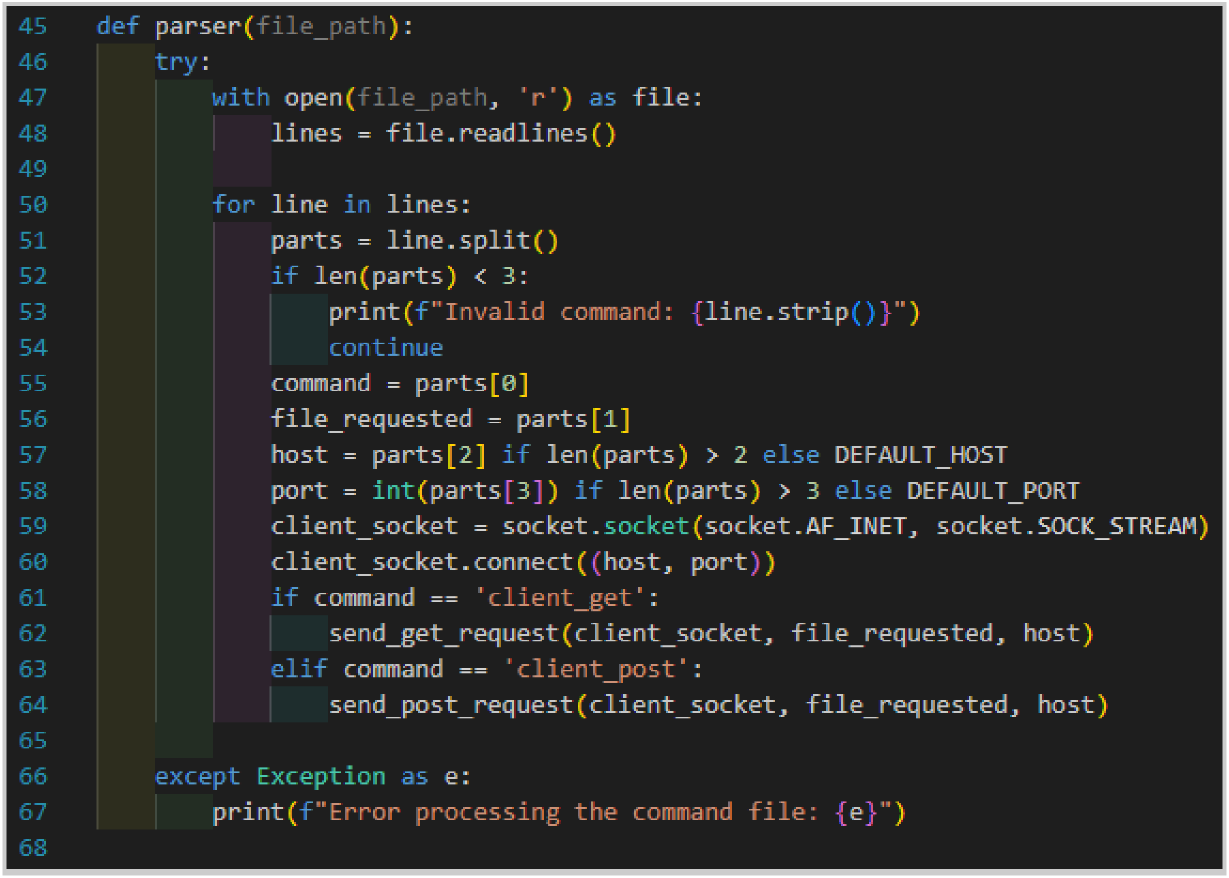
```python
def send_post_request(client_socket, file_path, host):
    try:
        # Read file data
        with open(file_path, 'rb') as f:
            file_data = f.read()

        # Construct POST request with file data
        request = f"POST /{file_path} HTTP/1.1\r\nHost: {host}\r\nContent-Length: {len(file_data)}\r\nConnection: close\r\n\r\n"
        client_socket.send(request.encode('utf-8')+file_data)

        # Receive the response
        response = client_socket.recv(4096)
        print(response.decode('utf-8'))
    except FileNotFoundError:
        print(f"File not found: {file_path}")

```

Purpose: Handles POST requests to upload files to the server.

Steps:

- Reads the file to be uploaded in binary mode (`rb`).
- Constructs a POST request in HTTP/1.1 format, including a `Content-Length` header to specify the size of the file being sent.
- The file data is appended to the request, and the combined message is sent to the server.
- The client then waits for the server's response and prints it out.
- If the file to upload does not exist, a `FileNotFoundError` is caught, and a message is printed.

- **parser() Function:**



```python
def parser(file_path):
    try:
        with open(file_path, 'r') as file:
            lines = file.readlines()

        for line in lines:
            parts = line.split()
            if len(parts) < 3:
                print(f"Invalid command: {line.strip()}")
                continue
            command = parts[0]
            file_requested = parts[1]
            host = parts[2] if len(parts) > 2 else DEFAULT_HOST
            port = int(parts[3]) if len(parts) > 3 else DEFAULT_PORT
            client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            client_socket.connect((host, port))
            if command == 'client_get':
                send_get_request(client_socket, file_requested, host)
            elif command == 'client_post':
                send_post_request(client_socket, file_requested, host)

    except Exception as e:
        print(f"Error processing the command file: {e}")
```

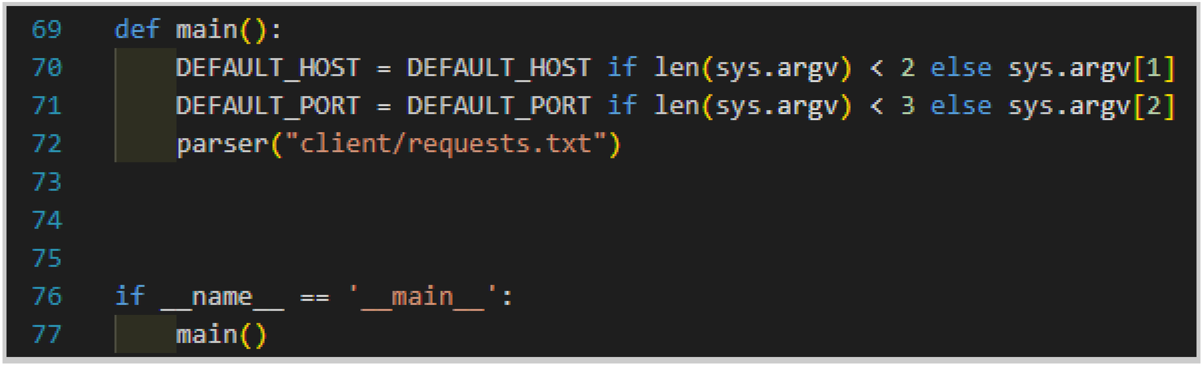
Purpose: This function reads the `requests.txt` file, parses the commands, and determines whether to send a GET or POST request.

Steps:

- Opens the command file and reads each line.
- Splits each line into parts (command, file path, host, and port).

- If the line is valid (i.e., has at least three parts), it establishes a connection to the server using sockets.
- Based on the command (`client_get` or `client_post`), it invokes the appropriate function (`send_get_request()` or `send_post_request()`).
- The function handles missing arguments by using the default host (`localhost`) and port (`8080`).

- **main() Function:**



```python
def main():
    DEFAULT_HOST = DEFAULT_HOST if len(sys.argv) < 2 else sys.argv[1]
    DEFAULT_PORT = DEFAULT_PORT if len(sys.argv) < 3 else sys.argv[2]
    parser("client/requests.txt")



if __name__ == '__main__':
    main()
```

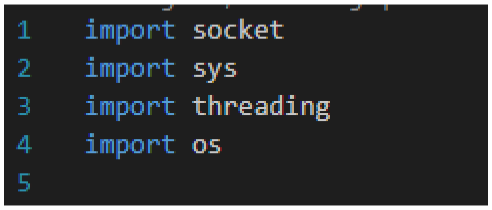
Purpose: Acts as the entry point for the program.

Steps:

- Checks if host and port values are provided via command-line arguments. If not, it falls back to the default values (`localhost` and `8080`).
- Calls the `parser()` function, which processes the `requests.txt` file and executes the necessary commands.
- The `main()` function runs when the script is executed as a standalone program. It processes the command file (`requests.txt`) and handles client-server interactions.

2. server_p2.py:

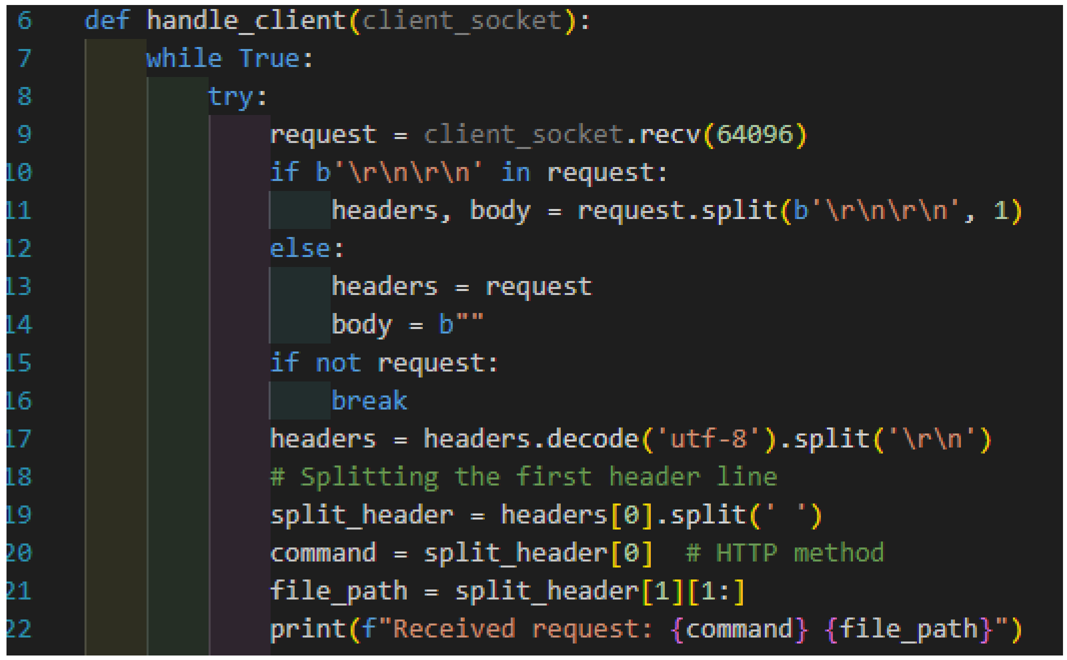
- **Imports:**



```python
import socket
import sys
import threading
import os

```

  - **`socket`**: Provides low-level networking interface for communication between the server and clients.
  - **`sys`**: Enables access to command-line arguments to specify the server's port.
  - **`threading`**: Used to handle multiple client connections concurrently by spawning a new thread for each client.
  - **`os`**: Facilitates file system operations, such as checking if a file exists and determining the file's path.

- **handle_client() Function:**



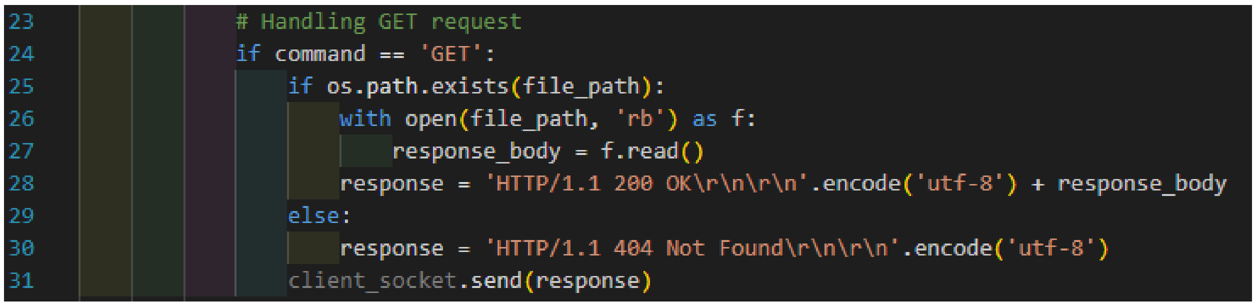
```python
def handle_client(client_socket):
    while True:
        try:
            request = client_socket.recv(64096)
            if b'\r\n\r\n' in request:
                headers, body = request.split(b'\r\n\r\n', 1)
            else:
                headers = request
                body = b""
            if not request:
                break
            headers = headers.decode('utf-8').split('\r\n')
            # Splitting the first header line
            split_header = headers[0].split(' ')
            command = split_header[0]  # HTTP method
            file_path = split_header[1][1:]
            print(f"Received request: {command} {file_path}")
```

Purpose: This function handles individual client requests. It's run in a separate thread for each client to enable simultaneous processing of multiple requests.

Steps:

- **Receiving the request**: The server waits to receive a request from the client using `recv()`. It expects HTTP requests, which consist of headers and (optional) body.
- **Parsing the request**: It splits the received data into HTTP headers and the body using `\r\n\r\n` as the separator.
- **Handling disconnection**: If the client disconnects (empty request), the server breaks out of the loop and closes the connection.
- **Command extraction**: The first line of the headers (e.g., `GET /index.html HTTP/1.1`) is split to extract the HTTP method (`GET`, `POST`) and the requested file path.

- **Handling GET Requests:**



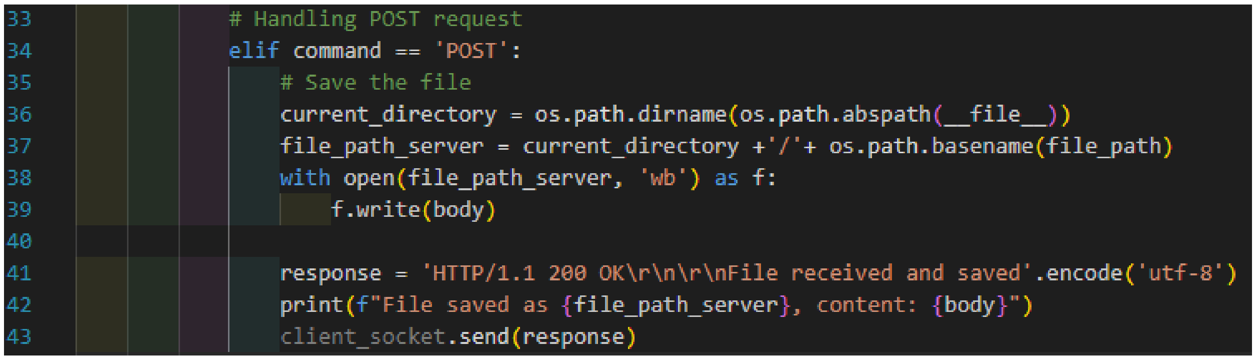
```
            # Handling GET request
            if command == 'GET':
                if os.path.exists(file_path):
                    with open(file_path, 'rb') as f:
                        response_body = f.read()
                    response = 'HTTP/1.1 200 OK\r\n\r\n'.encode('utf-8') + response_body
                else:
                    response = 'HTTP/1.1 404 Not Found\r\n\r\n'.encode('utf-8')
                client_socket.send(response)
```

Purpose: Handles `GET` requests, where the client requests a file from the server.

Steps:

- **File existence check**: The server checks if the requested file exists using `os.path.exists()`.
- **Sending the file**:
  - If the file exists, it's opened in binary mode (`'rb'`), read into memory, and sent back to the client with an **HTTP `200 OK`** status.
  - If the file doesn't exist, a **`404 Not Found`** response is sent.

- **Handling POST Requests:**



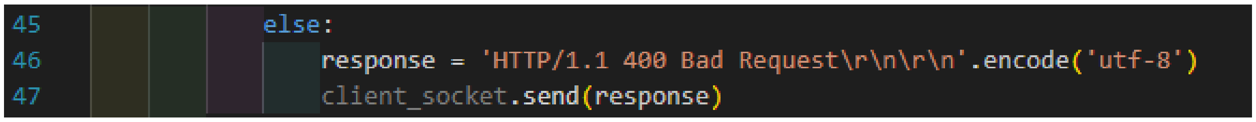
```python
            # Handling POST request
            elif command == 'POST':
                # Save the file
                current_directory = os.path.dirname(os.path.abspath(__file__))
                file_path_server = current_directory +'/'+ os.path.basename(file_path)
                with open(file_path_server, 'wb') as f:
                    f.write(body)

                response = 'HTTP/1.1 200 OK\r\n\r\nFile received and saved'.encode('utf-8')
                print(f"File saved as {file_path_server}, content: {body}")
                client_socket.send(response)
```

Purpose: Handles POST requests, where the client uploads a file to the server.

Steps:

- **File saving**: The server saves the uploaded file to its own directory using the filename from the request. The file's contents are written to disk in binary mode (`'wb'`).
- **Response**: After saving the file, the server sends a `200 OK` response, confirming that the file has been received and saved.

- **Handling Invalid Requests:**

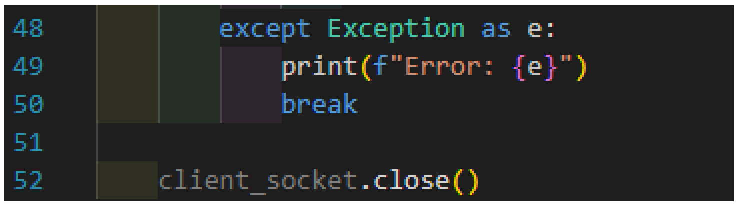


```python
            else:
                response = 'HTTP/1.1 400 Bad Request\r\n\r\n'.encode('utf-8')
                client_socket.send(response)
```

Purpose: Handles cases where the HTTP method is neither `GET` nor `POST`.

Action: The server responds with a **`400 Bad Request`** status to indicate an unrecognized or malformed request.

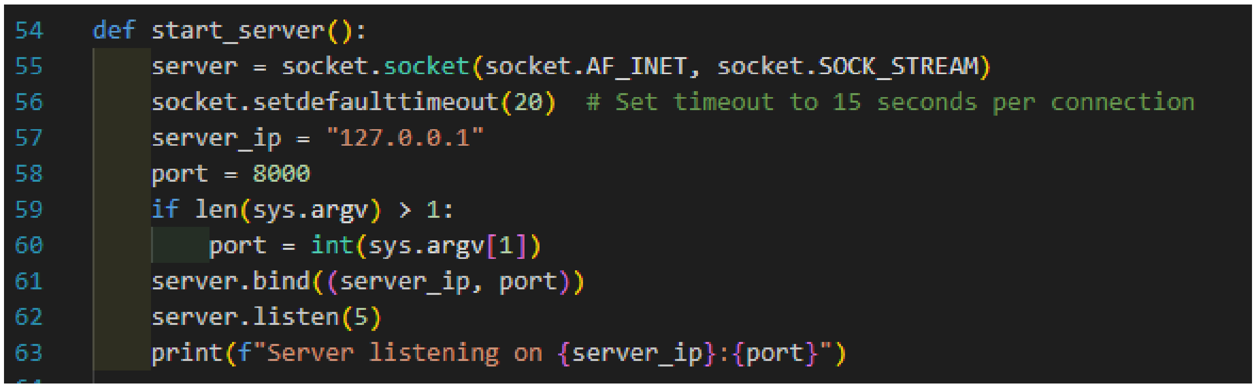
- **Error Handling:**



```python
        except Exception as e:
            print(f"Error: {e}")
            break

    client_socket.close()
```

Purpose: Catches and prints any exceptions that occur during request processing and closes the client connection gracefully.

- **start_server() Function:**



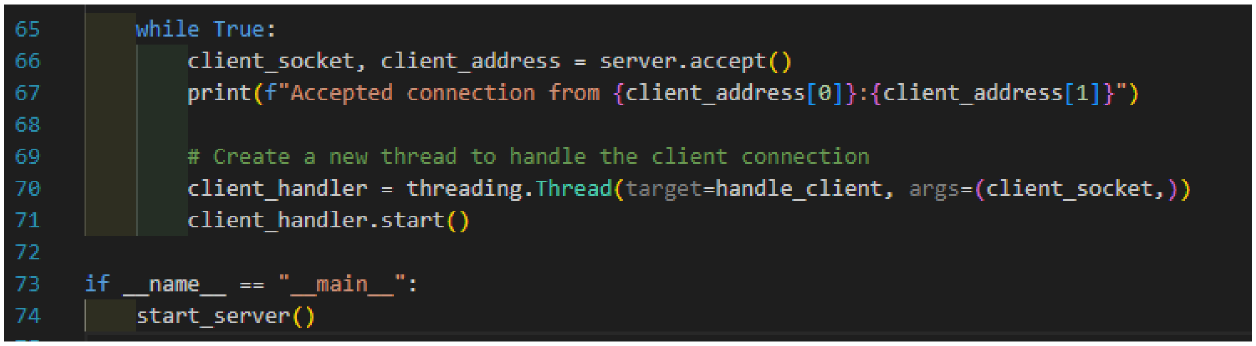
```
def start_server():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    socket.setdefaulttimeout(20)  # Set timeout to 15 seconds per connection
    server_ip = "127.0.0.1"
    port = 8000
    if len(sys.argv) > 1:
        port = int(sys.argv[1])
    server.bind((server_ip, port))
    server.listen(5)
    print(f"Server listening on {server_ip}:{port}")
```

Purpose: Initializes and starts the server to listen for incoming client connections.

Steps:

- **Socket setup**: Creates a new TCP socket using **`AF_INET`** (IPv4) and **`SOCK_STREAM`** (TCP).
- **Timeout**: Sets a timeout for client connections, after which the server stops waiting for data.
- **Port handling**: The server listens on the IP **`127.0.0.1`** (localhost) and port **`8000`** by default. The port can be customized via command-line arguments.
- **Bind and listen**: The server binds to the specified IP and port, and listens for up to **5** simultaneous connection requests.

- **Accepting Client Connections:**



```
    while True:
        client_socket, client_address = server.accept()
        print(f"Accepted connection from {client_address[0]}:{client_address[1]}")

        # Create a new thread to handle the client connection
        client_handler = threading.Thread(target=handle_client, args=(client_socket,))
        client_handler.start()

if __name__ == "__main__":
    start_server()
```

Purpose: Accepts incoming client connections and spawns a new thread for each connection.

Steps:

- **Accepting connections**: The server waits for incoming client connections using `accept()`, which returns a new socket for the connection and the client's address.
- **Threading**: Each client connection is handled in a separate thread (`threading.Thread`) by invoking the `handle_client()` function. This allows the server to process multiple client requests concurrently.

Data Structures:

1. client_parser.py:
    - **HTTP Request String**: The GET and POST requests are constructed as strings using the HTTP/1.1 format.
    - **Socket Object**: Sockets are used to establish a connection between the client and server.

2. server_p2.py:
    - **Socket Object**: The server uses a socket to listen for and accept connections from clients.
    - **Thread Object**: Each client connection is handled in a separate thread to allow concurrent handling of multiple clients.
    - **HTTP Request and Response Strings**: HTTP requests and responses are parsed and constructed as strings. Each request is processed based on its headers and body

---

Major Functions:

1. client_parser.py:

`send_get_request(client_socket, file_path, host)`:

- Constructs an HTTP GET request.
- Sends the request to the server.
- Receives the server's response, including the requested file.
- Saves the file to the client's local file system.

`send_post_request(client_socket, file_path, host)`:

- Constructs an HTTP POST request.
- Reads the file data to be sent to the server.
- Sends the request and file data to the server.
- Receives and displays the server's response.

`parser(file_path)`:

- Reads the commands from a `requests.txt` file, which specifies which files to request via GET or POST.
- For each command, it establishes a socket connection with the server and invokes either `send_get_request()` or `send_post_request()` depending on the type of request.

`main()`:

- Initiates the client by invoking the `parser()` function, which reads commands from the `requests.txt` file.

2. server_p2.py:

`handle_client(client_socket)`:

- Reads the incoming request from the client.
- Parses the request headers and body to determine the HTTP method (GET or POST).

- For GET requests: Attempts to locate the requested file and sends the file data as the HTTP response. If the file does not exist, it responds with a 404 error.
- For POST requests: Saves the file sent by the client and responds with a confirmation message.
- Closes the client connection after handling the request.

`start_server()`:

- Initializes the server socket.
- Binds the server to the specified IP address and port.
- Listens for incoming client connections.
- For each connection, it creates a new thread to handle the client using the `handle_client()` function.

Multithreading:

- The server is multithreaded, allowing it to handle multiple clients simultaneously.
- Each client connection is managed by a separate thread, which processes requests independently.
- This prevents one client from blocking others, improving the server's responsiveness and ability to handle concurrent operations.
- When a client connects, the server spawns a new thread to handle that client's communication, while the main server thread continues to listen for new connections.

Question:

➔ You're free to choose whether to implement a multi-threaded or multi-process approach, justifying your choice.

Answer:

➔ Multithreading allows multiple threads to run within the same process, sharing resources and data, which can lead to more efficient handling of concurrent requests.

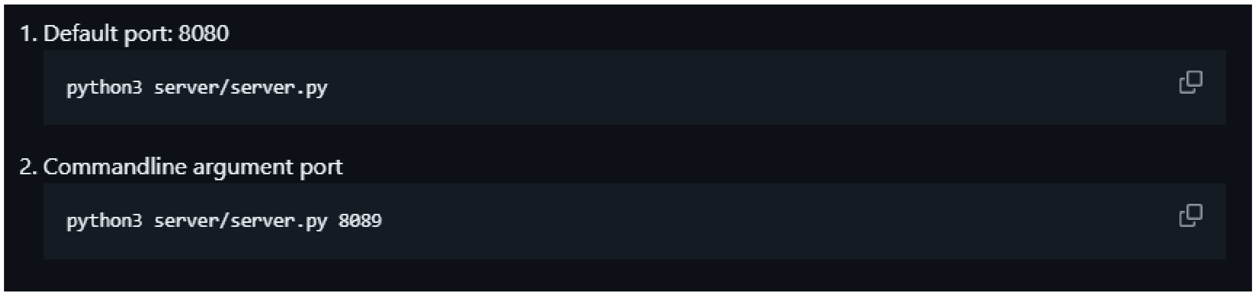
➔ Multithreading is often favored over multiprocessing due to its lower memory overhead and faster instantiation times.

Timeout and Inactivity:

➔ To prevent clients from holding up resources indefinitely, a timeout mechanism is in place. If a client fails to send a request within the specified timeout period, the server closes the connection.

➔ Additionally, the server has an inactivity checker that monitors overall server activity. If no client connections are made within a set period, the server shuts down, conserving resources.

How to run the server?



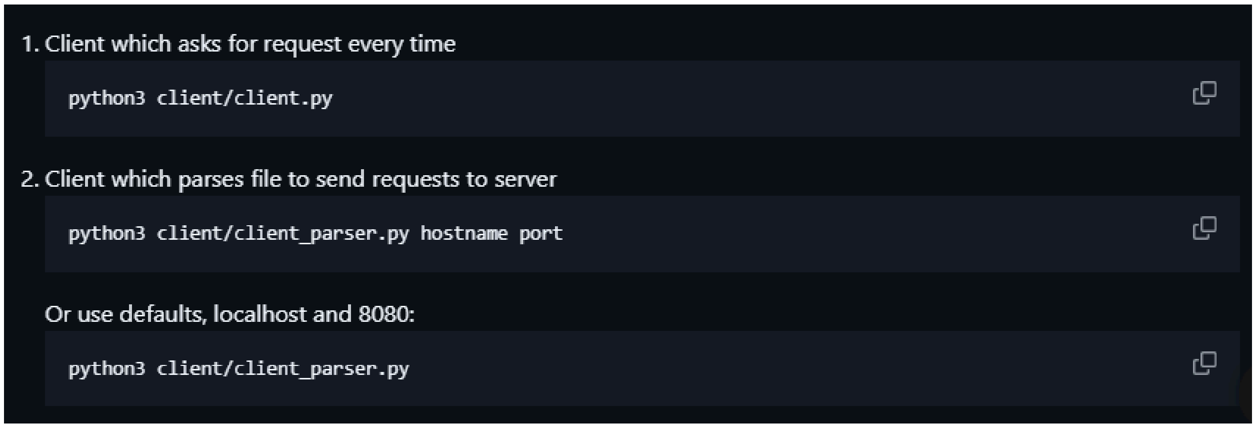
1. Default port: 8080

```
python3 server/server.py
```

2. Commandline argument port

```
python3 server/server.py 8089
```

How to send requests?



1. Client which asks for request every time

```
python3 client/client.py
```

2. Client which parses file to send requests to server

```
python3 client/client_parser.py hostname port
```

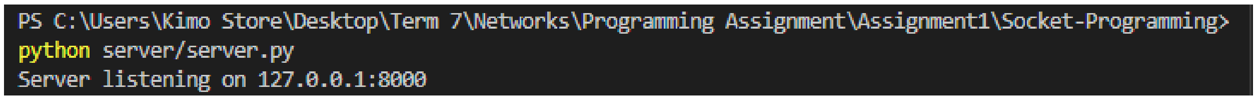
Or use defaults, localhost and 8080:

```
python3 client/client_parser.py
```

Sample Runs:

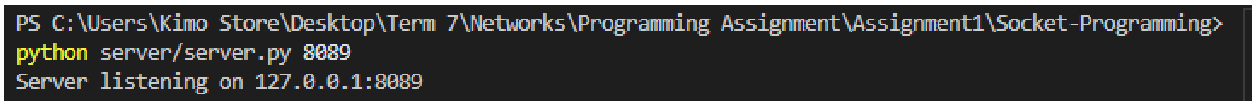
1. Starting the server:

- Default Port:



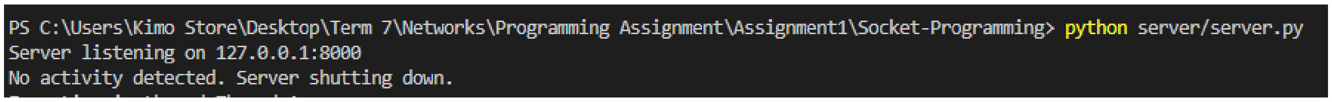
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python server/server.py
Server listening on 127.0.0.1:8000
```

- Custom Port:



```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python server/server.py 8089
Server listening on 127.0.0.1:8089
```

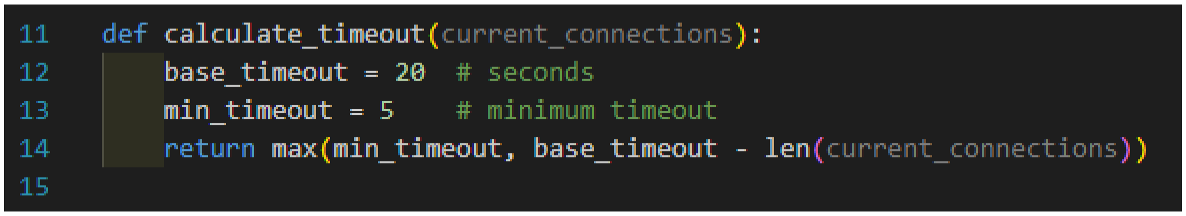
2. Server shuts down after 60 seconds of inactivity



```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> python server/server.py
Server listening on 127.0.0.1:8000
No activity detected. Server shutting down.
```

3. Client shuts down after a certain time of inactivity.

➔ The clients are specifically set to shut down after a dynamically calculated timeout.



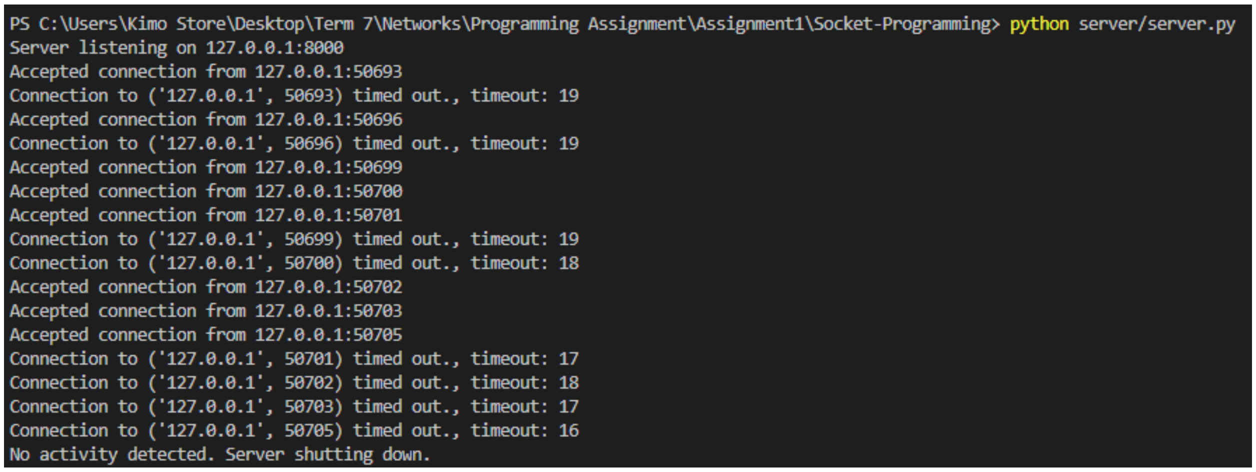
```python
def calculate_timeout(current_connections):
    base_timeout = 20  # seconds
    min_timeout = 5    # minimum timeout
    return max(min_timeout, base_timeout - len(current_connections))

```

➔ The `calculate_timeout` function defines a dynamic timeout mechanism that adjusts the timeout for each client based on the number of current active connections.

1. **`base_timeout = 20`**:
   - This sets the base timeout period to 20 seconds. If there are no connections, each client will have a 20-second timeout by default.
2. **`min_timeout = 5`**:
   - The minimum allowable timeout is set to 5 seconds. This ensures that even when the number of active connections is high, no client will have a timeout shorter than 5 seconds. This guarantees that every client gets at

least a 5-second window to send or receive data before the server considers the connection timed out.

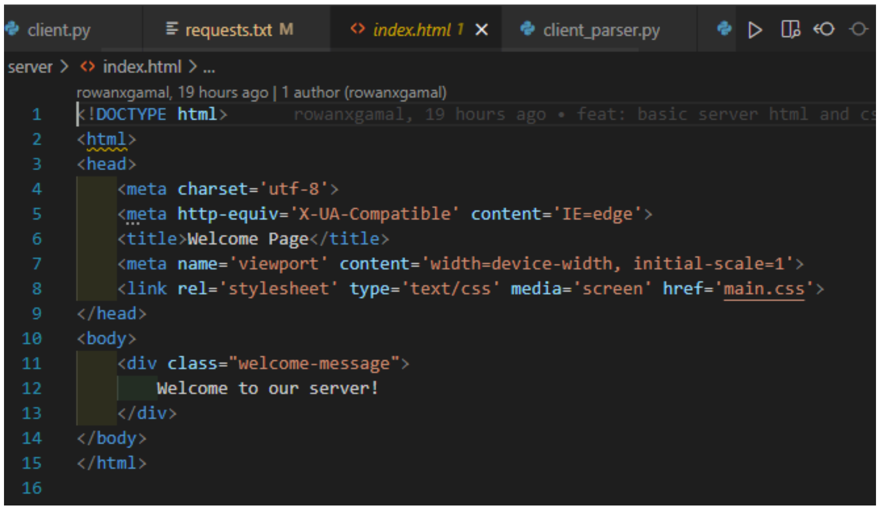
3. **`max(min_timeout, base_timeout - len(current_connections))`:**
    - **`base_timeout - len(current_connections)`**: As the number of active connections increases, the timeout for each new client decreases. For example, if there are 5 active connections, the timeout becomes `20 - 5 = 15 seconds`.
    - **`max(min_timeout, ...)`**: This ensures that the timeout never falls below the `min_timeout`. For instance, if there are more than 15 connections (i.e., `20 - 15 = 5`), the timeout will remain at the minimum value of 5 seconds, regardless of the number of connections beyond that.



```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> python server/server.py
Server listening on 127.0.0.1:8000
Accepted connection from 127.0.0.1:50693
Connection to ('127.0.0.1', 50693) timed out., timeout: 19
Accepted connection from 127.0.0.1:50696
Connection to ('127.0.0.1', 50696) timed out., timeout: 19
Accepted connection from 127.0.0.1:50699
Accepted connection from 127.0.0.1:50700
Accepted connection from 127.0.0.1:50701
Connection to ('127.0.0.1', 50699) timed out., timeout: 19
Connection to ('127.0.0.1', 50700) timed out., timeout: 18
Accepted connection from 127.0.0.1:50702
Accepted connection from 127.0.0.1:50703
Accepted connection from 127.0.0.1:50705
Connection to ('127.0.0.1', 50701) timed out., timeout: 17
Connection to ('127.0.0.1', 50702) timed out., timeout: 18
Connection to ('127.0.0.1', 50703) timed out., timeout: 17
Connection to ('127.0.0.1', 50705) timed out., timeout: 16
No activity detected. Server shutting down.
```

4. **Test GET Requests:**

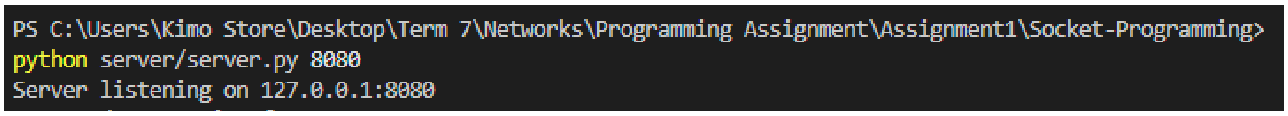
- Request a text file from the server:
    1. There's a file server/index.html in the server's directory.



client.py
requests.txt M
index.html 1
client_parser.py
server > index.html > ...
rowanxgamal, 19 hours ago | 1 author (rowanxgamal)

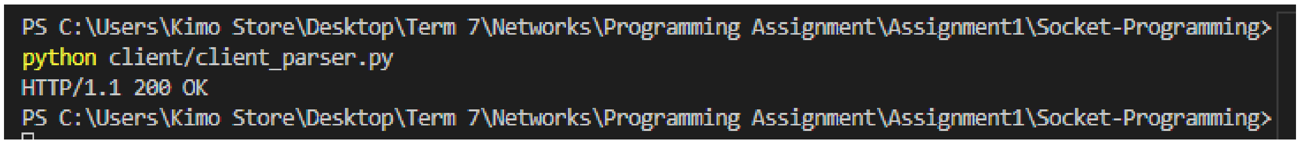
```html
<!DOCTYPE html>       rowanxgamal, 19 hours ago • feat: basic server html and c
<html>
<head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Welcome Page</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
    <link rel='stylesheet' type='text/css' media='screen' href='main.css'>
</head>
<body>
    <div class="welcome-message">
        Welcome to our server!
    </div>
</body>
</html>
```

    2. Run the server on port 8080



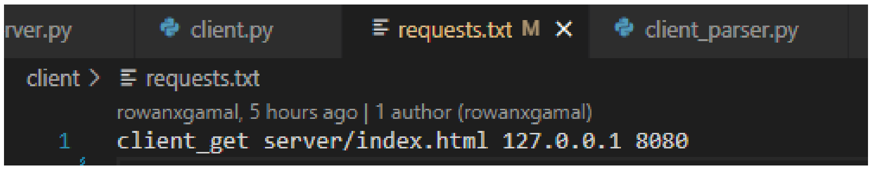
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python server/server.py 8080
Server listening on 127.0.0.1:8080
```

    3. Run the client to send a GET Request:



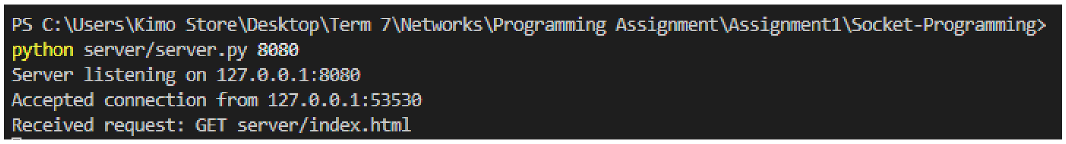
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python client/client_parser.py
HTTP/1.1 200 OK
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
```

    4. Contents of request.txt file:



```
client_get server/index.html 127.0.0.1 8080
```

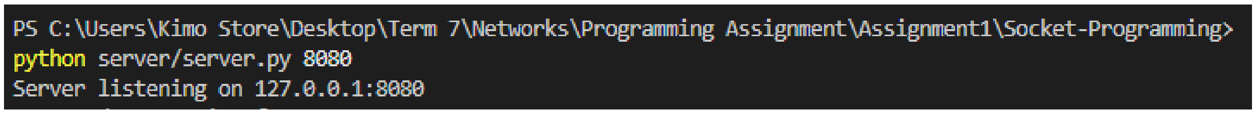
5. Expected Result:



```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python server/server.py 8080
Server listening on 127.0.0.1:8080
Accepted connection from 127.0.0.1:53530
Received request: GET server/index.html
```

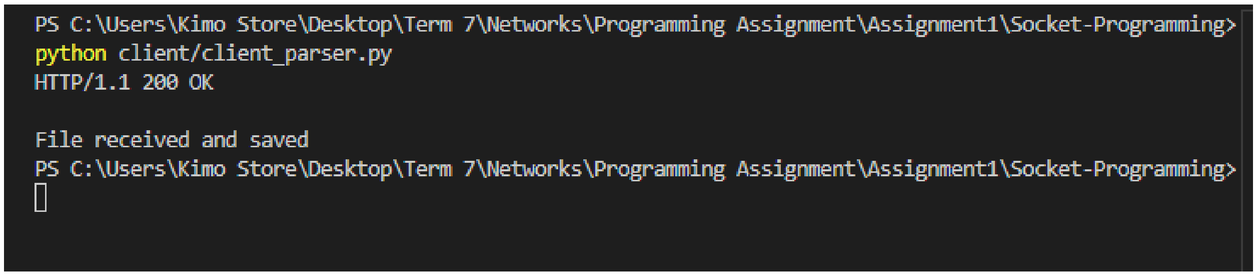
3. Test POST Requests:

- Send a text file to the server:
    1. Run the server on port 8080



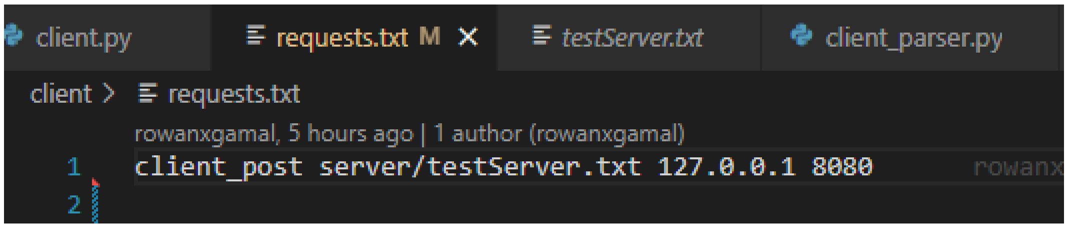
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python server/server.py 8080
Server listening on 127.0.0.1:8080
```

    2. Run the client to send a POST Request:



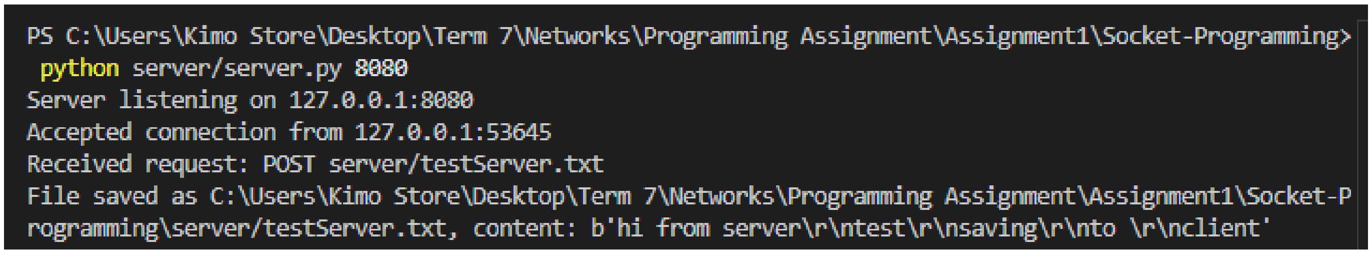
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python client/client_parser.py
HTTP/1.1 200 OK

File received and saved
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
```

    3. Contents of request.txt file:



```
client_post server/testServer.txt 127.0.0.1 8080
```

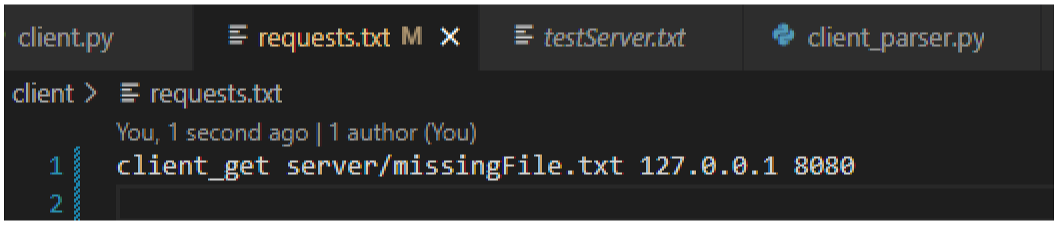
    4. Result:



```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
 python server/server.py 8080
Server listening on 127.0.0.1:8080
Accepted connection from 127.0.0.1:53645
Received request: POST server/testServer.txt
File saved as C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-P
rogramming\server/testServer.txt, content: b'hi from server\r\ntest\r\nsaving\r\nto \r\nclient'
```

4. Request a non-existent file (should return 404)

1. Modify client/requests.txt:



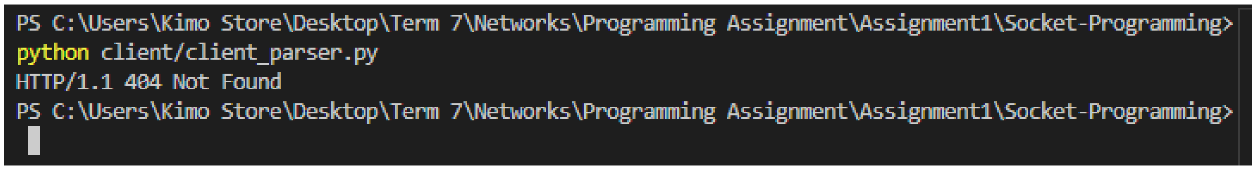
client.py | requests.txt M | testServer.txt | client_parser.py

client > requests.txt

You, 1 second ago | 1 author (You)

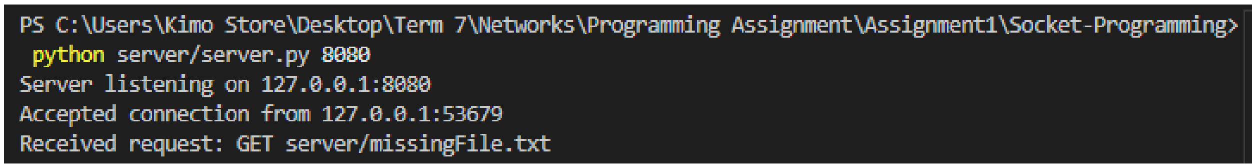
```
client_get server/missingFile.txt 127.0.0.1 8080
```

2. Run the client again:



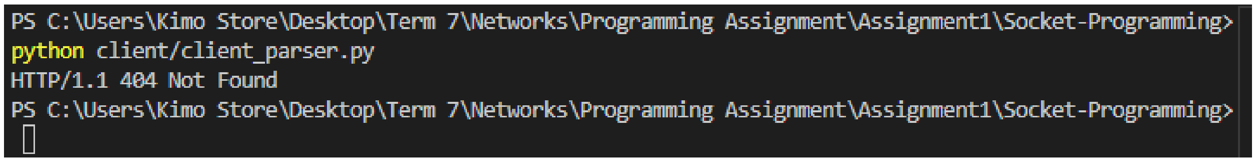
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python client/client_parser.py
HTTP/1.1 404 Not Found
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
```

3. Server console:



```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
 python server/server.py 8080
Server listening on 127.0.0.1:8080
Accepted connection from 127.0.0.1:53679
Received request: GET server/missingFile.txt
```

4. Result:

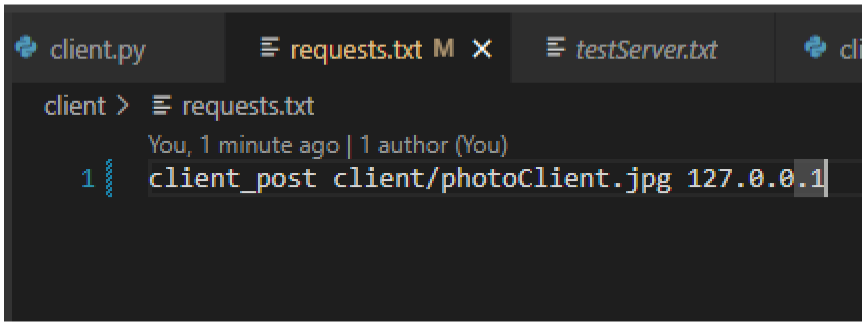


```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python client/client_parser.py
HTTP/1.1 404 Not Found
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
```

5. Sending an image to the server:

1. Added an image client/photoClient.jpg in the client directory.

2. Modified client/requests.txt:



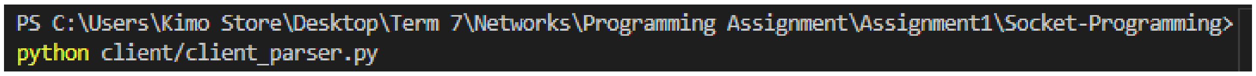
client.py | requests.txt M | testServer.txt | cl

client > requests.txt

You, 1 minute ago | 1 author (You)

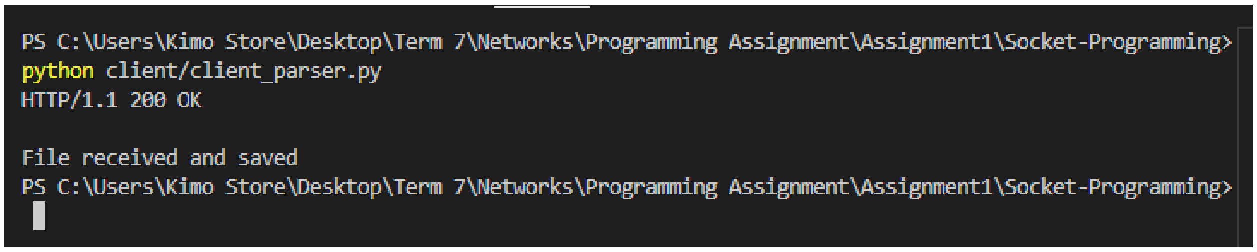
```
client_post client/photoClient.jpg 127.0.0.1
```

3. Run the client:



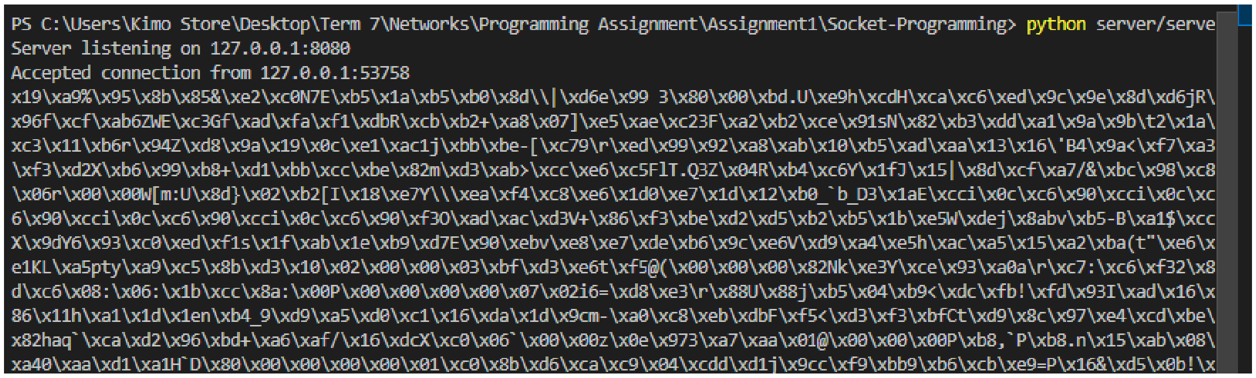
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python client/client_parser.py
```

4. Result:



```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python client/client_parser.py
HTTP/1.1 200 OK

File received and saved
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
```

5. The image is saved on server side:

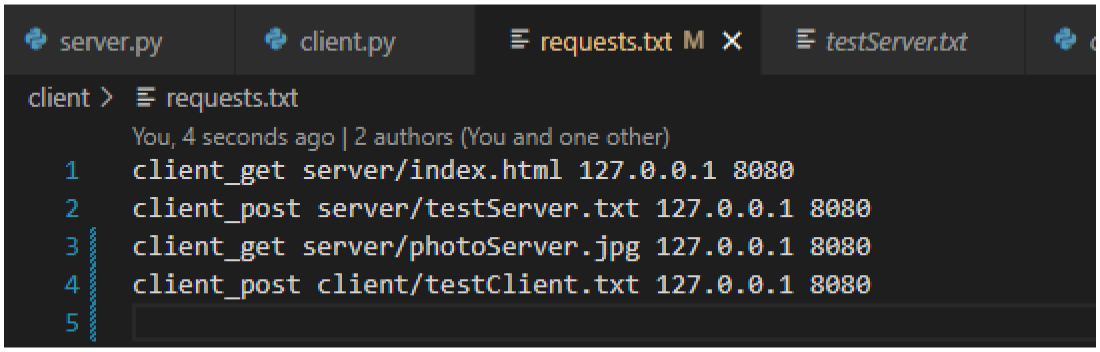


```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> python server/serve
Server listening on 127.0.0.1:8080
Accepted connection from 127.0.0.1:53758
x19\xa9%\x95\x8b\x85&\xe2\xc0N7E\xb5\x1a\xb5\xb0\x8d\\|\xd6e\x99 3\x80\x00\xbd.U\xe9h\xcdH\xca\xc6\xed\x9c\x9e\x8d\xd6jR\
x96f\xcf\xab6ZWE\xc3Gf\xad\xfa\xf1\xdbR\xcb\xb2+\xa8\x07]\xe5\xae\xc23F\xa2\xb2\xce\x91sN\x82\xb3\xdd\xa1\x9a\x9b\t2\x1a\
xc3\x11\xb6r\x94Z\xd8\x9a\x19\x0c\xe1\xac1j\xbb\xbe-[\xc79\r\xed\x99\x92\xa8\xab\x10\xb5\xad\xaa\x13\x16\'B4\x9a<\xf7\xa3
\xf3\xd2X\xb6\x99\xb8+\xd1\xbb\xcc\xbe\x82m\xd3\xab>\xcc\xe6\xc5FlT.Q3Z\x04R\xb4\xc6Y\x1fJ\x15|\x8d\xcf\xa7/&\xbc\x98\xc8
\x06r\x00\x00W[m:U\x8d}\x02\xb2[I\x18\xe7Y\\\xea\xf4\xc8\xe6\x1d0\xe7\x1d\x12\xb0_`b_D3\x1aE\xcci\x0c\xc6\x90\xcci\x0c\xc
6\x90\xcci\x0c\xc6\x90\xcci\x0c\xc6\x90\xf3O\xad\xac\xd3V+\x86\xf3\xbe\xd2\xd5\xb2\xb5\x1b\xe5W\xdej\x8abv\xb5-B\xa1$\xcc
X\x9dY6\x93\xc0\xed\xf1s\x1f\xab\x1e\xb9\xd7E\x90\xebv\xe8\xe7\xde\xb6\x9c\xe6V\xd9\xa4\xe5h\xac\xa5\x15\xa2\xba(t"\xe6\x
e1KL\xa5pty\xa9\xc5\x8b\xd3\x10\x02\x00\x00\x03\xbf\xd3\xe6t\xf5@(\x00\x00\x00\x82Nk\xe3Y\xce\x93\xa0a\r\xc7:\xc6\xf32\x8
d\xc6\x08:\x06:\x1b\xcc\x8a:\x00P\x00\x00\x00\x00\x07\x02i6=\xd8\xe3\r\x88U\x88j\xb5\x04\xb9<\xdc\xfb!\xfd\x93I\xad\x16\x
86\x11h\xa1\x1d\x1en\xb4_9\xd9\xa5\xd0\xc1\x16\xda\x1d\x9cm-\xa0\xc8\xeb\xdbF\xf5<\xd3\xf3\xbfCt\xd9\x8c\x97\xe4\xcd\xbe\
x82haq`\xca\xd2\x96\xbd+\xa6\xaf/\x16\xdcX\xc0\x06`\x00\x00z\x0e\x973\xa7\xaa\x01@\x00\x00\x00P\xb8,`P\xb8.n\x15\xab\x08\
xa40\xaa\xd1\xa1H`D\x80\x00\x00\x00\x00\x01\xc0\x8b\xd6\xca\xc9\x04\xcdd\xd1j\x9cc\xf9\xbb9\xb6\xcb\xe9=P\x16&\xd5\x0b!\x
```

6. Handling Multiple Requests (Concurrent Clients):

- Simulate multiple clients sending concurrent requests.

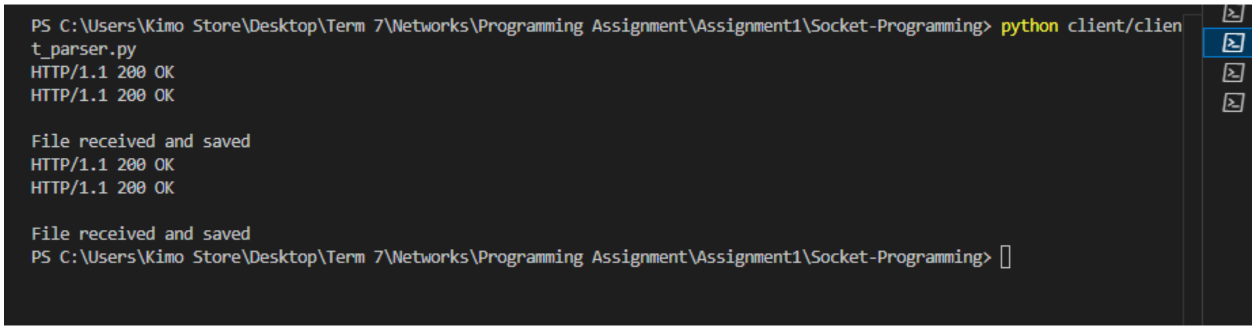
1. Modify `client/requests.txt` to include multiple GET and POST requests:



```
client_get server/index.html 127.0.0.1 8080
client_post server/testServer.txt 127.0.0.1 8080
client_get server/photoServer.jpg 127.0.0.1 8080
client_post client/testClient.txt 127.0.0.1 8080
```

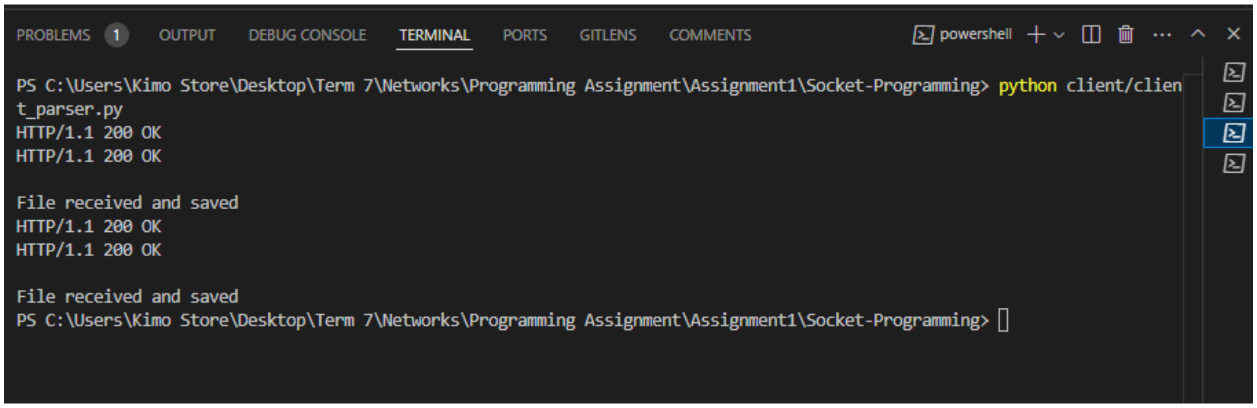
2. Run the client multiple times in separate terminals (tried on 3 clients).
3. Result:

   Client 1:



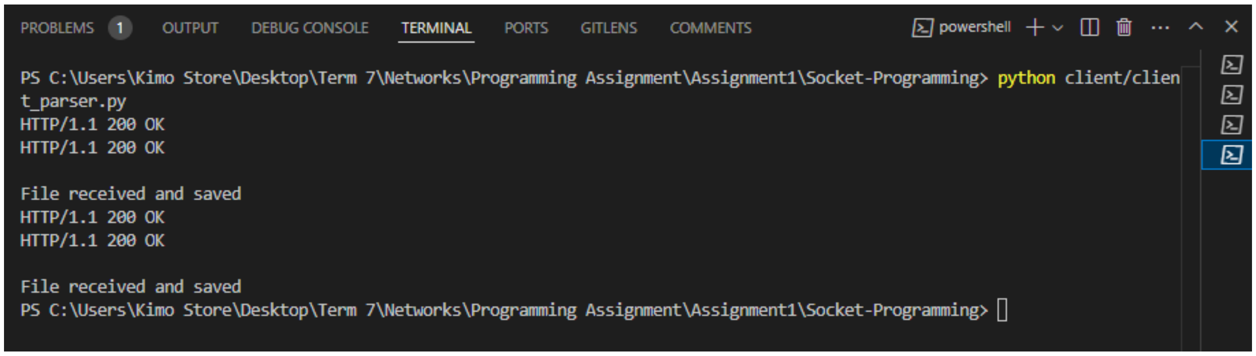
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> python client/clien
t_parser.py
HTTP/1.1 200 OK
HTTP/1.1 200 OK

File received and saved
HTTP/1.1 200 OK
HTTP/1.1 200 OK

File received and saved
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> 
```

   Client 2:



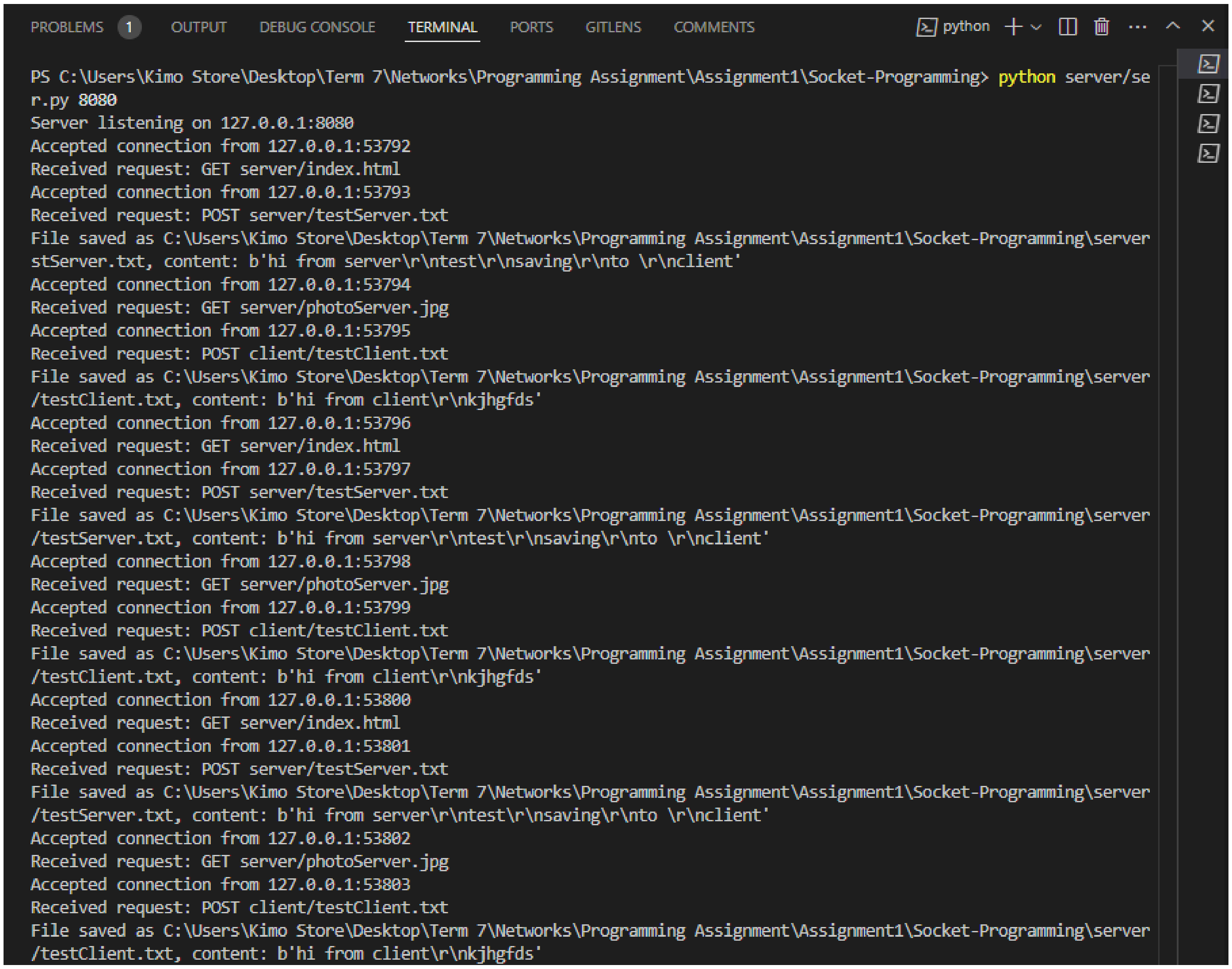
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> python client/clien
t_parser.py
HTTP/1.1 200 OK
HTTP/1.1 200 OK

File received and saved
HTTP/1.1 200 OK
HTTP/1.1 200 OK

File received and saved
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> 
```

   Client 3:



```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> python client/clien
t_parser.py
HTTP/1.1 200 OK
HTTP/1.1 200 OK

File received and saved
HTTP/1.1 200 OK
HTTP/1.1 200 OK

File received and saved
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> 
```

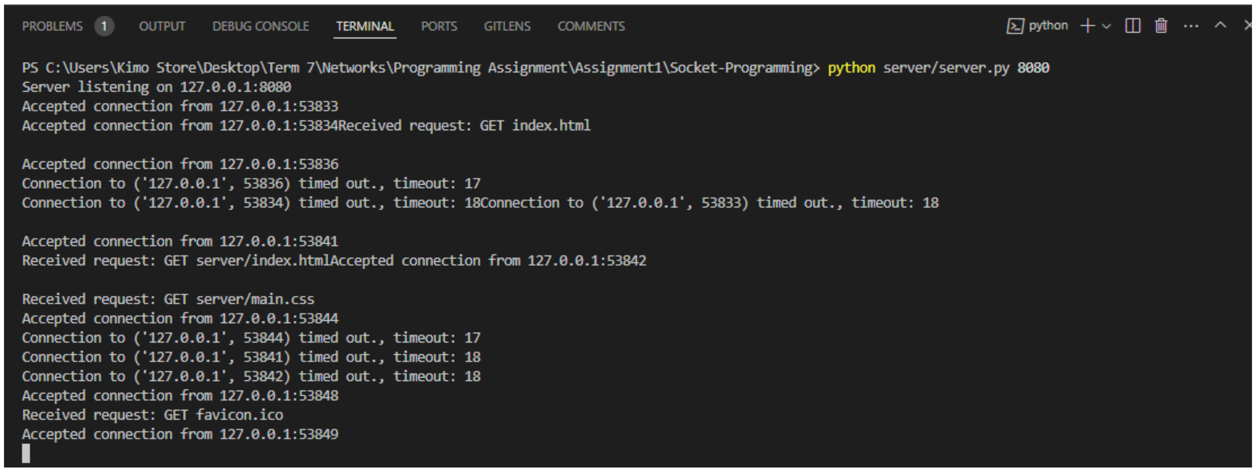
4. Result on server side:



```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> python server/se
r.py 8080
Server listening on 127.0.0.1:8080
Accepted connection from 127.0.0.1:53792
Received request: GET server/index.html
Accepted connection from 127.0.0.1:53793
Received request: POST server/testServer.txt
File saved as C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming\server
stServer.txt, content: b'hi from server\r\ntest\r\nsaving\r\nto \r\nclient'
Accepted connection from 127.0.0.1:53794
Received request: GET server/photoServer.jpg
Accepted connection from 127.0.0.1:53795
Received request: POST client/testClient.txt
File saved as C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming\server
/testClient.txt, content: b'hi from client\r\nkjhgfds'
Accepted connection from 127.0.0.1:53796
Received request: GET server/index.html
Accepted connection from 127.0.0.1:53797
Received request: POST server/testServer.txt
File saved as C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming\server
/testServer.txt, content: b'hi from server\r\ntest\r\nsaving\r\nto \r\nclient'
Accepted connection from 127.0.0.1:53798
Received request: GET server/photoServer.jpg
Accepted connection from 127.0.0.1:53799
Received request: POST client/testClient.txt
File saved as C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming\server
/testClient.txt, content: b'hi from client\r\nkjhgfds'
Accepted connection from 127.0.0.1:53800
Received request: GET server/index.html
Accepted connection from 127.0.0.1:53801
Received request: POST server/testServer.txt
File saved as C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming\server
/testServer.txt, content: b'hi from server\r\ntest\r\nsaving\r\nto \r\nclient'
Accepted connection from 127.0.0.1:53802
Received request: GET server/photoServer.jpg
Accepted connection from 127.0.0.1:53803
Received request: POST client/testClient.txt
File saved as C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming\server
/testClient.txt, content: b'hi from client\r\nkjhgfds'
```

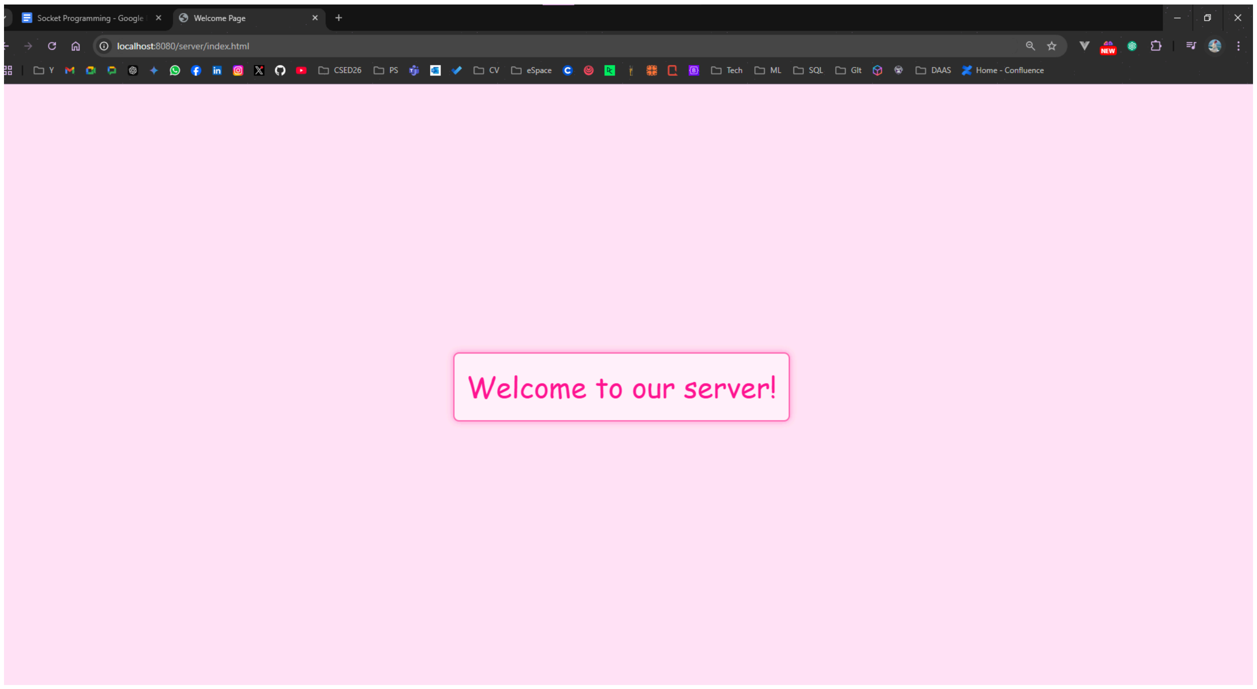
Bonus: Test your server with a real web browser:

1. Run the server:



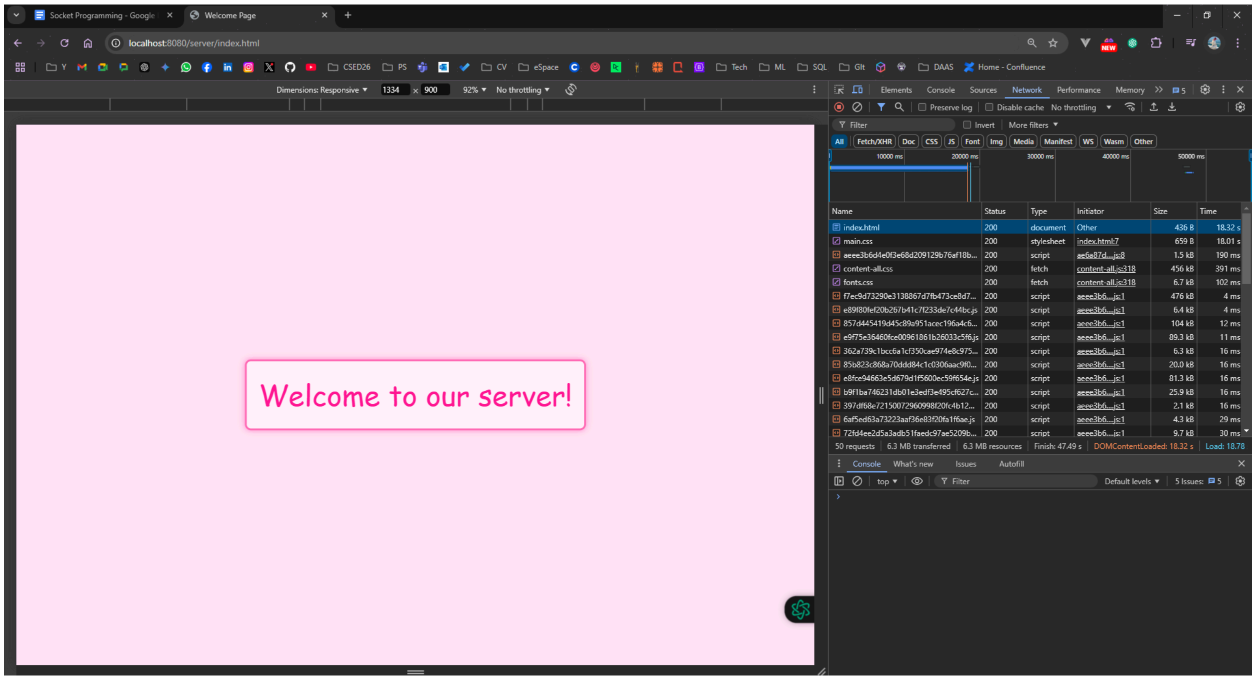
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming> python server/server.py 8080
Server listening on 127.0.0.1:8080
Accepted connection from 127.0.0.1:53833
Accepted connection from 127.0.0.1:53834Received request: GET index.html

Accepted connection from 127.0.0.1:53836
Connection to ('127.0.0.1', 53836) timed out., timeout: 17
Connection to ('127.0.0.1', 53834) timed out., timeout: 18Connection to ('127.0.0.1', 53833) timed out., timeout: 18

Accepted connection from 127.0.0.1:53841
Received request: GET server/index.htmlAccepted connection from 127.0.0.1:53842

Received request: GET server/main.css
Accepted connection from 127.0.0.1:53844
Connection to ('127.0.0.1', 53844) timed out., timeout: 17
Connection to ('127.0.0.1', 53841) timed out., timeout: 18
Connection to ('127.0.0.1', 53842) timed out., timeout: 18
Accepted connection from 127.0.0.1:53848
Received request: GET favicon.ico
Accepted connection from 127.0.0.1:53849
```

2. Browser View for a simple index.html file:



3. Right click and select inspect.

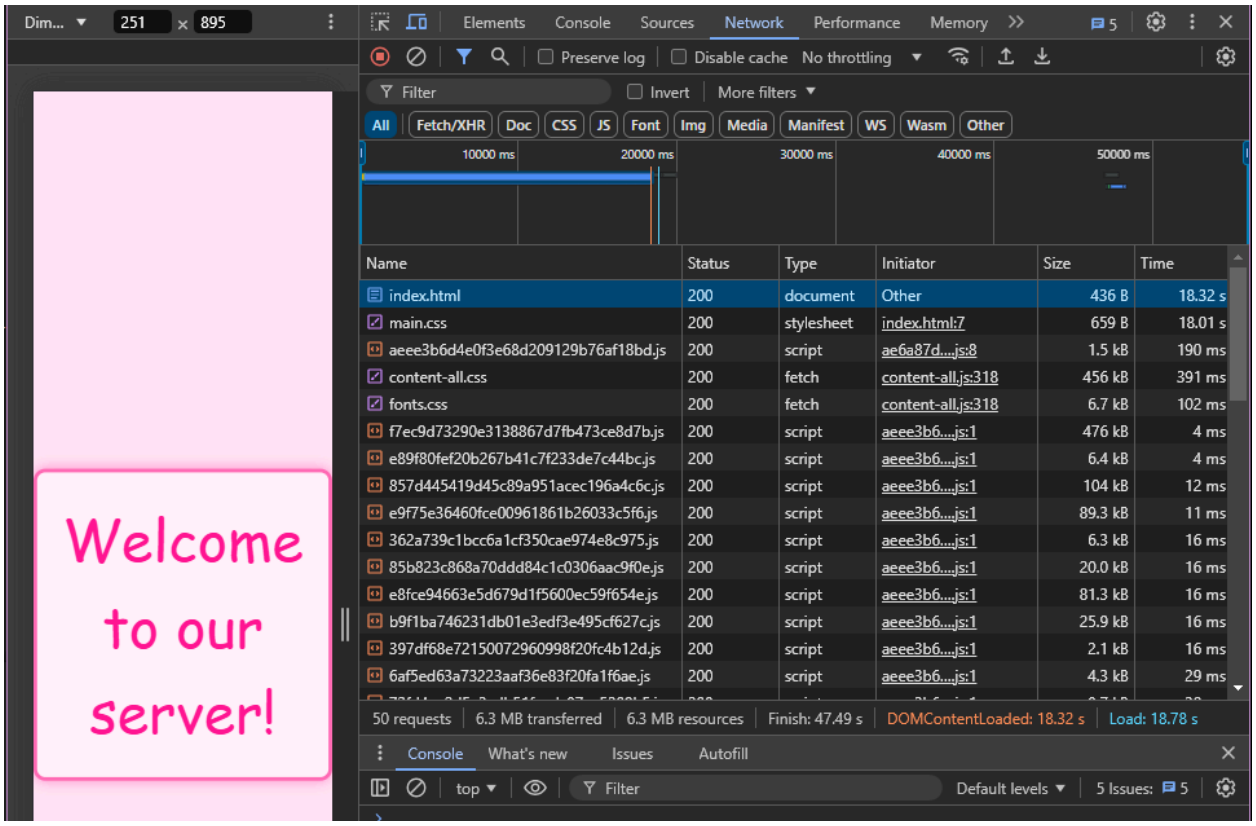
4. From the networks tab:



Name	Status	Type	Initiator	Size	Time
index.html	200	document	Other	436 B	18.32 s
main.css	200	stylesheet	index.html:7	659 B	18.01 s
aeee3b6d4e0f3e68d209129b76af18b...	200	script	ae6a87d....js:8	1.5 kB	190 ms
content-all.css	200	fetch	content-all.js:318	456 kB	391 ms
fonts.css	200	fetch	content-all.js:318	6.7 kB	102 ms
f7ec9d73290e3138867d7fb473ce8d7...	200	script	aeee3b6....js:1	476 kB	4 ms
e89f80fef20b267b41c7f233de7c44bc.js	200	script	aeee3b6....js:1	6.4 kB	4 ms
857d445419d45c89a951acec196a4c6...	200	script	aeee3b6....js:1	104 kB	12 ms
e9f75e36460fce00961861b26033c5f6.js	200	script	aeee3b6....js:1	89.3 kB	11 ms
362a739c1bcc6a1cf350cae974e8c975...	200	script	aeee3b6....js:1	6.3 kB	16 ms
85b823c868a70ddd84c1c0306aac9f0...	200	script	aeee3b6....js:1	20.0 kB	16 ms
e8fce94663e5d679d1f5600ec59f654e.js	200	script	aeee3b6....js:1	81.3 kB	16 ms
b9f1ba746231db01e3edf3e495cf627c...	200	script	aeee3b6....js:1	25.9 kB	16 ms
397df68e72150072960998f20fc4b12...	200	script	aeee3b6....js:1	2.1 kB	16 ms
6af5ed63a73223aaf36e83f20fa1f6ae.js	200	script	aeee3b6....js:1	4.3 kB	29 ms
72fd4ee2d5a3adb51faedc97ae5209b...	200	script	aeee3b6....js:1	9.7 kB	30 ms

50 requests | 6.3 MB transferred | 6.3 MB resources | Finish: 47.49 s | DOMContentLoaded: 18.32 s | Load: 18.78

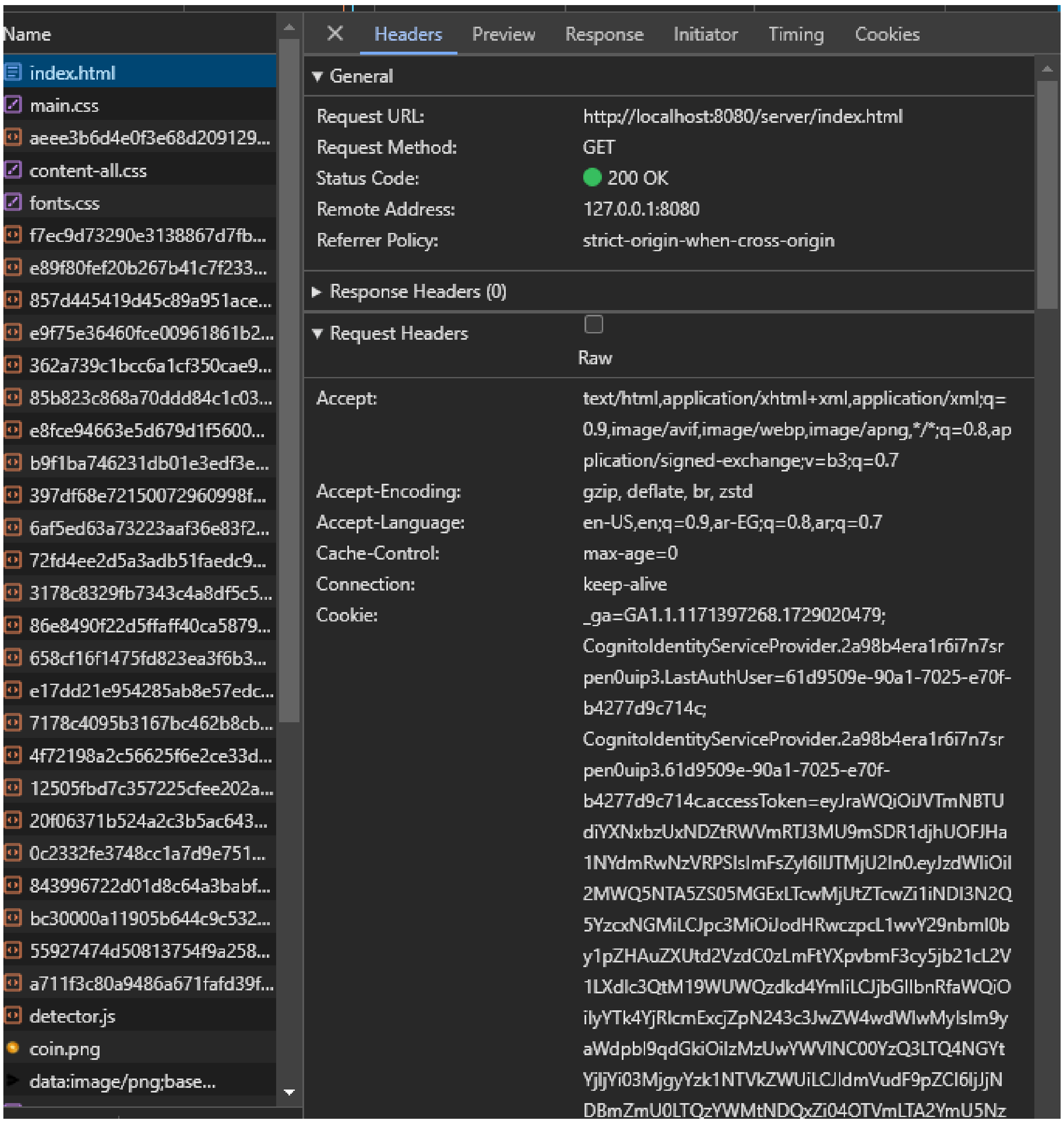
5. You can see index.html



Name	Status	Type	Initiator	Size	Time
index.html	200	document	Other	436 B	18.32 s
main.css	200	stylesheet	index.html:7	659 B	18.01 s
aeee3b6d4e0f3e68d209129b76af18bd.js	200	script	ae6a87d....js:8	1.5 kB	190 ms
content-all.css	200	fetch	content-all.js:318	456 kB	391 ms
fonts.css	200	fetch	content-all.js:318	6.7 kB	102 ms
f7ec9d73290e3138867d7fb473ce8d7b.js	200	script	aeee3b6....js:1	476 kB	4 ms
e89f80fef20b267b41c7f233de7c44bc.js	200	script	aeee3b6....js:1	6.4 kB	4 ms
857d445419d45c89a951acec196a4c6c.js	200	script	aeee3b6....js:1	104 kB	12 ms
e9f75e36460fce00961861b26033c5f6.js	200	script	aeee3b6....js:1	89.3 kB	11 ms
362a739c1bcc6a1cf350cae974e8c975.js	200	script	aeee3b6....js:1	6.3 kB	16 ms
85b823c868a70ddd84c1c0306aac9f0e.js	200	script	aeee3b6....js:1	20.0 kB	16 ms
e8fce94663e5d679d1f5600ec59f654e.js	200	script	aeee3b6....js:1	81.3 kB	16 ms
b9f1ba746231db01e3edf3e495cf627c.js	200	script	aeee3b6....js:1	25.9 kB	16 ms
397df68e72150072960998f20fc4b12d.js	200	script	aeee3b6....js:1	2.1 kB	16 ms
6af5ed63a73223aaf36e83f20fa1f6ae.js	200	script	aeee3b6....js:1	4.3 kB	29 ms

50 requests | 6.3 MB transferred | 6.3 MB resources | Finish: 47.49 s | DOMContentLoaded: 18.32 s | Load: 18.78 s

6. Notice the headers tab:



```
index.html
main.css
aeee3b6d4e0f3e68d209129...
content-all.css
fonts.css
f7ec9d73290e3138867d7fb...
e89f80fef20b267b41c7f233...
857d445419d45c89a951ace...
e9f75e36460fce00961861b2...
362a739c1bcc6a1cf350cae9...
85b823c868a70ddd84c1c03...
e8fce94663e5d679d1f5600...
b9f1ba746231db01e3edf3e...
397df68e72150072960998f...
6af5ed63a73223aaf36e83f2...
72fd4ee2d5a3adb51faedc9...
3178c8329fb7343c4a8df5c5...
86e8490f22d5ffaff40ca5879...
658cf16f1475fd823ea3f6b3...
e17dd21e954285ab8e57edc...
7178c4095b3167bc462b8cb...
4f72198a2c56625f6e2ce33d...
12505fbd7c357225cfee202a...
20f06371b524a2c3b5ac643...
0c2332fe3748cc1a7d9e751...
843996722d01d8c64a3babf...
bc30000a11905b644c9c532...
55927474d50813754f9a258...
a711f3c80a9486a671fafd39f...
detector.js
coin.png
data:image/png;base...
```

```
▼ General
Request URL:          http://localhost:8080/server/index.html
Request Method:       GET
Status Code:          ● 200 OK
Remote Address:       127.0.0.1:8080
Referrer Policy:      strict-origin-when-cross-origin

▶ Response Headers (0)

▼ Request Headers
Accept:               text/html,application/xhtml+xml,application/xml;q=
                      0.9,image/avif,image/webp,image/apng,*/*;q=0.8,ap
                      plication/signed-exchange;v=b3;q=0.7
Accept-Encoding:      gzip, deflate, br, zstd
Accept-Language:      en-US,en;q=0.9,ar-EG;q=0.8,ar;q=0.7
Cache-Control:        max-age=0
Connection:           keep-alive
Cookie:               _ga=GA1.1.1171397268.1729020479;
                      CognitoIdentityServiceProvider.2a98b4era1r6i7n7sr
                      pen0uip3.LastAuthUser=61d9509e-90a1-7025-e70f-
                      b4277d9c714c;
                      CognitoIdentityServiceProvider.2a98b4era1r6i7n7sr
                      pen0uip3.61d9509e-90a1-7025-e70f-
                      b4277d9c714c.accessToken=eyJraWQiOiJVTmNBTU
                      diYXNxbzUxNDZtRWVmRTJ3MU9mSDR1djhUOFJHa
                      1NYdmRwNzVRPSIsImFsZyI6IlJTMjU2In0.eyJzdWIiOiI
                      2MWQ5NTA5ZS05MGExLTcwMjUtZTcwZi1iNDI3N2Q
                      5YzcxNGMiLCJpc3MiOiJodHRwczpcL1wvY29nbml0b
                      y1pZHAuZXUtd2VzdC0zLmFtYXpvbmF3cy5jb21cL2V
                      1LXdlc3QtM19WUWQzdkd4YmliLCJjbGllbnRfaWQiO
                      iIyYTk4YjRlcmExcjZpN243c3JwZW4wdWlwMyIsIm9y
                      aWdpbl9qdGkiOiIzMzUwYWVlNC00YzQ3LTQ4NGYt
                      YjljYi03MjgyYzk1NTVkZWUiLCJldmVudF9pZCI6IjJjN
                      DBmZmU0LTQzYWMtNDQxZi04OTVmLTA2YmU5Nz
```

Bonus: Performance Evaluation:

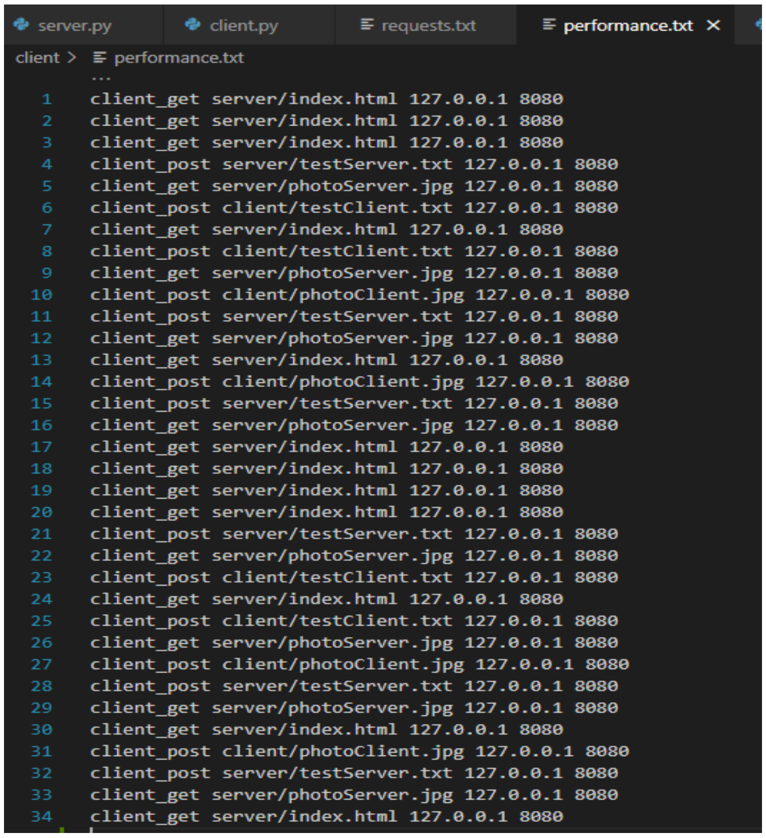
- We wrote a script "**performance.py**" to simulate multiple GET and POST requests to the server and measure the delay for each request, plotting a chart of the average delay as the number of requests increases.

- The script sends GET or POST requests to the server based on commands in client/performance.txt. It measures the delay for each request and calculates the average delay over time.

- Performance Metrics:
  - ➔ **Total Delay**: The total time taken by all the requests.
  - ➔ **Average Delay**: The cumulative average time for all requests.
  - ➔ **Throughput**: The rate of requests per second.
  - ➔ **Chart**: It uses Matplotlib to plot the number of requests against the average delay.

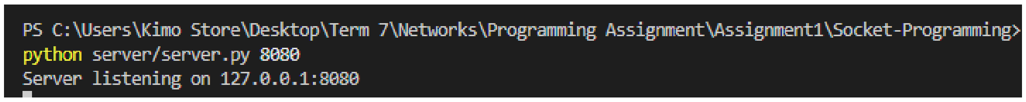
- For every request we measure the delay time and get its average by dividing total delay by the number of requests so far.

- **Steps for Performance Evaluation:**
  1. Create performance.txt file for testing in client directory and add multiple client_get and client_post commands for testing:



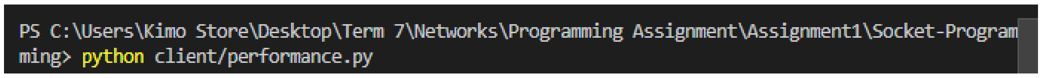
```
client_get server/index.html 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_post server/testServer.txt 127.0.0.1 8080
client_get server/photoServer.jpg 127.0.0.1 8080
client_post client/testClient.txt 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_post client/testClient.txt 127.0.0.1 8080
client_get server/photoServer.jpg 127.0.0.1 8080
client_post client/photoClient.jpg 127.0.0.1 8080
client_post server/testServer.txt 127.0.0.1 8080
client_get server/photoServer.jpg 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_post client/photoClient.jpg 127.0.0.1 8080
client_post server/testServer.txt 127.0.0.1 8080
client_get server/photoServer.jpg 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_post server/testServer.txt 127.0.0.1 8080
client_get server/photoServer.jpg 127.0.0.1 8080
client_post client/testClient.txt 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_post client/testClient.txt 127.0.0.1 8080
client_get server/photoServer.jpg 127.0.0.1 8080
client_post client/photoClient.jpg 127.0.0.1 8080
client_post server/testServer.txt 127.0.0.1 8080
client_get server/photoServer.jpg 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
client_post client/photoClient.jpg 127.0.0.1 8080
client_post server/testServer.txt 127.0.0.1 8080
client_get server/photoServer.jpg 127.0.0.1 8080
client_get server/index.html 127.0.0.1 8080
```

2. Run the server:



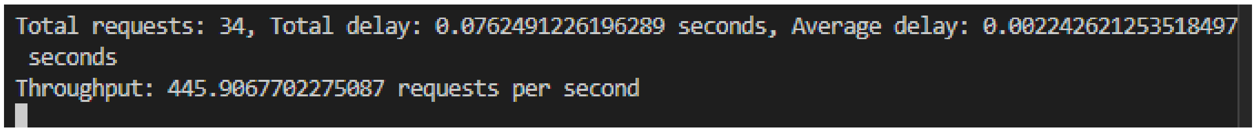
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Programming>
python server/server.py 8080
Server listening on 127.0.0.1:8080
```

3. Run the performance.py:



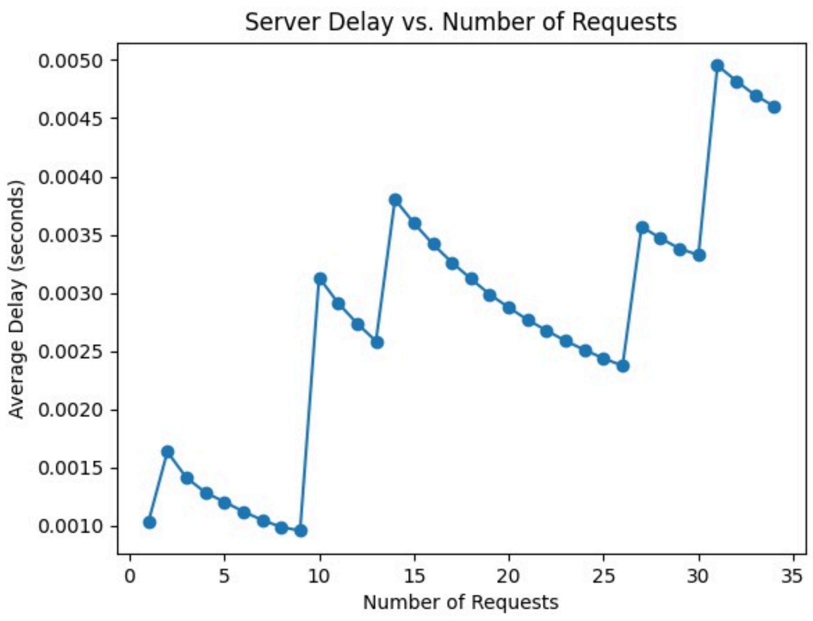
```
PS C:\Users\Kimo Store\Desktop\Term 7\Networks\Programming Assignment\Assignment1\Socket-Program
ming> python client/performance.py
```

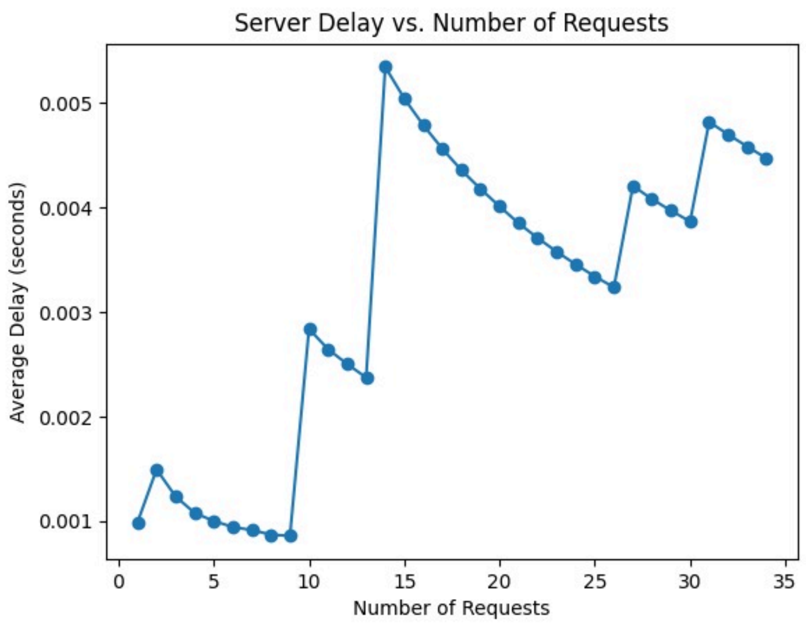
4. Output:



```
Total requests: 34, Total delay: 0.0762491226196289 seconds, Average delay: 0.002242621253518497
 seconds
Throughput: 445.9067702275087 requests per second
```

5. **Chart:**





- **Performance Analysis:**

  Peak delay time is in the middle because it's the point of congestion which has the most requests still being processed by the server.

➔ Key Observations:

1. **Initial Performance**:

- For the first few requests (up to around 10), there is a relatively low average delay (around 0.001 - 0.002 seconds). This indicates that the server is able to handle a small number of requests efficiently without much delay.

2. **Spike in Delay**:

- Between the 10th and 12th request, there is a noticeable spike in average delay, which increases to around 0.0035 seconds. This could indicate the server is starting to experience higher load, leading to increased processing time.

3. **Fluctuations**:

- After the initial spike, the average delay fluctuates between around 0.0025 and 0.003 seconds until the 25th request. This could indicate that the server is sometimes able to catch up and reduce delay, potentially due to short periods of lower traffic or successful handling of queued requests.

4. **Consistent Increase After 25 Requests**:

- After 25 requests, the delay begins to increase more steadily, with the average delay reaching around 0.005 seconds by the 35th request. This likely reflects the server becoming more overwhelmed as the number of requests continues to rise.

5. **Server Load**:

- As the number of requests increases, the server may need to handle more concurrent operations, leading to longer response times and higher average delay.

6. **Resource Bottlenecks**:

- The spikes in delay could be caused by server resource limitations, such as CPU or memory usage, where the server experiences moments of increased load, leading to temporary increases in delay.

7. **Thread Scheduling Overhead**:

- Since the server handles requests using multithreading, the overhead associated with thread management may contribute to increased delays as more threads are created to handle the rising number of requests.