MARMARA UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE 4074 Computer Networks – Programming Assignment

Group Members:

Ayşe Sena Aydemir – 150119735

Emre Cihan Varlı – 150119711

Furkan Bozkurt – 150119767

1. DESIGN DOCUMENT

· Project description

The aim of this project is to develop a multi-threaded HTTP server and a proxy server, both of which demonstrate the basics of socket programming and provide key functionalities as specified in the assignment.

The HTTP server is designed to handle client requests and return HTML documents of a specific size based on the requested URI. It supports multi-threading for concurrency and ensures proper error handling for invalid requests or unsupported HTTP methods.

The proxy server acts as an intermediary between the client and the HTTP server. It forwards client requests to the HTTP server and returns the response. The proxy server also includes restrictions, such as rejecting requests with a URI size greater than a specified limit, and handling errors like when the HTTP server is unavailable.

Both servers were tested for performance using the wrk tool, which helped evaluate their handling of concurrency and response speed. These tests provided insights into the efficiency and scalability of the implementations.

This project combines theoretical knowledge with practical programming skills to create robust and efficient network applications. It highlights the importance of concurrency, error handling, and performance testing in real-world server applications.

· System architecture

This section outlines the system architecture of the project, which involves the development of two core components: an HTTP server and a proxy server. Both components are designed to operate concurrently, handle multiple client requests, and ensure efficient request validation and response generation.

Overall Design

The system consists of the following key components:

1. HTTP Server: Handles client requests directly, generates dynamic HTML responses based on URI size, and manages invalid requests with proper error codes.
2. Proxy Server: Acts as an intermediary, forwarding valid client requests to the HTTP server, managing restrictions on URI size, and relaying the HTTP server’s responses back to the clients.

Both components use multi-threading to manage concurrent client connections, ensuring efficient utilization of resources and high responsiveness.

HTTP Server Architecture

1. Server Setup

- Listening for Connections: The server listens on a specified port provided by the user via command-line arguments.
- Concurrency: Each client connection is handled by a new thread, enabling the server to process multiple requests simultaneously.
- Socket Management: The server uses the Python `socket` library with options such as `SO_REUSEADDR` to avoid binding issues.

2. Content Generation

- Dynamic HTML Responses: Based on the URI, the server generates HTML content with a size ranging from 100 to 20,000 bytes. Content is populated with the letter 'a' to meet the required size.
- Title Customization: The `<title>` tag in the HTML response dynamically reflects the content size.

3. Error Handling

- Invalid URI Sizes: Requests with sizes less than 100 bytes or greater than 20,000 bytes result in a 400 Bad Request response.
- Unsupported Methods: Requests using methods other than GET return a 501 Not Implemented response.
- Invalid HTTP Methods: Requests with invalid or unrecognized methods return a 400 Bad Request response.

4. Testing

- Using POSTMAN and CURL: The HTTP server was tested by sending various GET requests and validating its responses. Multithreading capabilities were verified by sending multiple concurrent requests from different terminals.

Proxy Server Architecture

1. Proxy Setup

- Port and Host Configuration: The proxy server listens on port 8888 and forwards requests to the HTTP server running on a specified host and port.
- Concurrency: Similar to the HTTP server, the proxy uses threads to handle multiple client connections simultaneously.

2. Client Request Handling

- Request Parsing: The proxy parses incoming HTTP requests to extract the method and URI.
- Size Restrictions: Requests with a URI size greater than 9,999 bytes are rejected with a 414 URI Too Long error.
- Validation: Only valid HTTP requests are forwarded; invalid requests receive appropriate error responses.

3. Request Forwarding

- Communication with HTTP Server:
    - The proxy establishes a new socket connection to the HTTP server.
    - It sends the parsed HTTP request with proper headers (e.g., Host).
- Response Relaying:
    - The proxy receives the response from the HTTP server, decodes it, and forwards it back to the client.
    - The proxy ensures that the response format adheres to HTTP standards.

4. Error Handling

- Oversized URI Requests: Requests with URI sizes exceeding 9,999 bytes are rejected with an HTTP 414 error.
- HTTP Server Unavailability: If the HTTP server is unreachable, the proxy responds with a 404 Not Found error.

5. Testing

- Using POSTMAN and CURL: The proxy was tested by sending GET requests to port 8888. Correct forwarding and error handling were verified by observing the responses relayed from the HTTP server.

Component Interaction

1. **Client to Proxy Server:**
    - Clients send HTTP requests to the proxy server on port 8888.
    - The proxy validates the request and forwards it to the HTTP server if valid.
2. **Proxy Server to HTTP Server:**
    - The proxy constructs a valid HTTP request and sends it to the HTTP server.
    - The HTTP server processes the request, generates a response, and sends it back to the proxy.
3. **Proxy Server to Client:**
    - The proxy receives the response from the HTTP server and forwards it to the client.

Concurrency and Performance

- Multi-threading: Both the HTTP server and proxy server utilize threading to handle concurrent client requests efficiently.
- Resource Utilization: By using lightweight threads, the system achieves responsiveness under high load conditions.
- Scalability: The architecture supports multiple simultaneous connections without significant performance degradation.

This architecture ensures a modular, scalable, and efficient design for handling HTTP client-server interactions while adhering to the requirements of the project.

· Functional requirements

HTTP Server

1. The server must handle client requests via multi-threading to allow concurrent connections.
2. It should accept a port number as a command-line argument during execution.
3. The server must generate and return HTML documents of sizes specified by the requested URI, ranging from 100 to 20,000 bytes.
4. If the requested URI is invalid (e.g., not a number, less than 100, or greater than 20,000), the server should respond with a "Bad Request" error (HTTP 400).
5. For unsupported methods, the server should respond with "Not Implemented" (HTTP 501).
6. All responses must include proper HTTP headers such as Content-Type and Content-Length.
7. The server should be compatible with web browsers for requests made via valid URLs.

Proxy Server

1. The proxy server should forward client GET requests to the HTTP server and return the received responses to the client.
2. The proxy server must reject requests with a URI size greater than 9,999 bytes and respond with a "Request-URI Too Long" error (HTTP 414).
3. If the HTTP server is unavailable, the proxy server must return a "Not Found" error (HTTP 404).
4. The proxy server must process both absolute and relative URLs according to the HTTP specifications (RFC 2068).

Error Handling

1. Both servers must handle invalid inputs gracefully and return meaningful error messages with appropriate HTTP status codes.

Testing and Performance

1. The implementation must be tested using the wrk tool to evaluate performance metrics like latency, requests per second, and the effects of concurrency.
2. Both servers must demonstrate stable performance under varying levels of concurrent requests and client connections.

2. IMPLEMENTATION DETAILS

· HTTP server implementation

The HTTP server is designed to accept client connections using sockets. In the project, it is expected to handle multiple client requests concurrently using threading. The server processes HTTP GET requests and responds with dynamically generated HTML content. For requests using methods other than GET, the server is required to reject them and return an HTTP 501 Not Implemented error. Additionally, for requests that use method types not recognized as valid HTTP methods (e.g., methods other than GET, POST, DELETE, PUT, PATCH, HEAD, OPTIONS, or CONNECT), the server must respond with an HTTP 400 Bad Request error. The server operates on a specified port, accepts requests, and returns HTML responses of varying sizes based on the request URI. It also validates and handles incorrect HTTP requests with appropriate error responses.

Implementation Details:

The implementation consisted of four main components: Server Setup, Client Handling, Error Management and Content Generating.

Server Setup:

First, we set up a server that listens on the given port, binds to all network interfaces and reuses the address to avoid binding issues. It creates a new thread for each client connection, delegating the process to handle_client.

```
def start_server(port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    #set socket options to allow reuse and avoid binding issues
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_socket.bind(('0.0.0.0', port))  # Bind to all interfaces
    server_socket.listen(5)
    print(f"[INFO] Server listening on port {port}...")

    try:
        while True:
            try:
                #accept a new client connection
                client_socket, client_address = server_socket.accept()

                #handle the proxy connection in a new thread
                threading.Thread(target=handle_client, args=(client_socket,
client_address)).start()
```

```
            except Exception as e:
                print(f"[ERROR] Error accepting connection: {e}")
    finally:
        server_socket.close()
        print("[INFO] Server has been stopped.")
```

After that step, since we want our server to listen on the given port number, not a specified port; we used the ArgParse library on the main function of the python. We created a new parser as a port number to run the server on. Then, in the main function of the python, we started our server in a separate thread because running the server on its own thread allows the server to manage incoming client connections and delegate request handling to additional threads efficiently. This setup ensures that multiple clients can be served concurrently without blocking each other.
To end the server session, we added a KeyboardInterrupt (CTRL + C) method.

Now, in order to start our server on terminal, here are the steps to follow:
-Be sure that the mainsocket.py file is under the Users-User document in your computer.
Open a powershell or bash terminal:

```bash
    PS C:\Users\emre> python mainsocket.py 8080
    ->[INFO] Server listening on port 8080… #terminal response
```

Give the port number for the server to listen on.

Client Setup:

Most functions and error handling that need to be added are done on the client side. However, we will first send a request to the server and try to get a response.

```
def handle_client(client_socket, client_address):

    print(f"[INFO] Connected to client: {client_address}")

    try:
        #receive the HTTP request from client
         request = client_socket.recv(2048).decode('utf-8')
         print(f"[REQUEST] From {client_address}:\n{request}")

         client_socket.sendall(response.encode('utf-8'))
    except Exception as e:
        print(f"[ERROR] Error handling client {client_address}: {e}")
    finally:
        client_socket.close()
        print(f"[INFO] Connection to {client_address} closed.")
```

Send Connection Request from POSTMAN:

```
curl (GET method)
     http://localhost:8000/2000 #HTTP Request
```

Now our server receives the incoming request, accepts this request and sends a response. In order to test multithreading, you can open new terminals and send different requests progressively.

Generate a Content:

Now, the server must prepare an HTML file to display in the interface to the client. The server should generate an HTML file with a size equal to the number entered to the right of the port number when the request is made. If we follow the example given in the project file, let's first divide the incoming request into parts. Then, in order to use it as an integer, let's split the slashes (/) and take the entered value as size to generate the letter 'a' in that size.

```
#to add a letter with given size in the uri.
def generate_content(size, title="response"):
    http_content = f'<html><head><title>{title}</title></head><body>'
    while len(http_content) < size:
        http_content += 'a '
    http_content += '</body></html>'
    return http_content[:size]  # trim to the exact size
```

Get the uri size in handle_client by splitting /:

```
request_line = request.splitlines()[0]
method, uri, _ = request_line.split(" ")
```

Call the generate_content function to return the response to the client:

```
sized_title = f"I am {size} bytes long"
        content = generate_content(size, title = sized_title)
        response = (
            f"HTTP/1.1 200 OK\r\n"
            f"Content-Type: text/html\r\n"
            f"Content-Length: {len(content)}\r\n"
            f"\r\n"
            f"{content}"
        )
```

Error Handling for Invalid Requests:

The server shouldn’t accept requests with the size greater than 20000 bytes, or less than 100 bytes, exactly. Again, in the handle_client function, add a condition to accept our request in the main try.

```
      try:
            size = int(uri.lstrip('/'))
            if size < 100 or size > 20000: #size limitation
                raise ValueError

        except ValueError:
            response = "HTTP/1.1 400 Bad Request\r\n\r\n<h1>400 Bad
Request</h1>"
            client_socket.sendall(response.encode('utf-8'))
            return
```

The server shouldn’t accept any request type rather than GET, for other requests it should send an “501 Not Implemented” error.

```
        if method != "GET":
            response = "HTTP/1.1 501 Not Implemented\r\n\r\n<h1>501 Not
Implemented</h1>"
            client_socket.sendall(response.encode('utf-8'))
            return
```

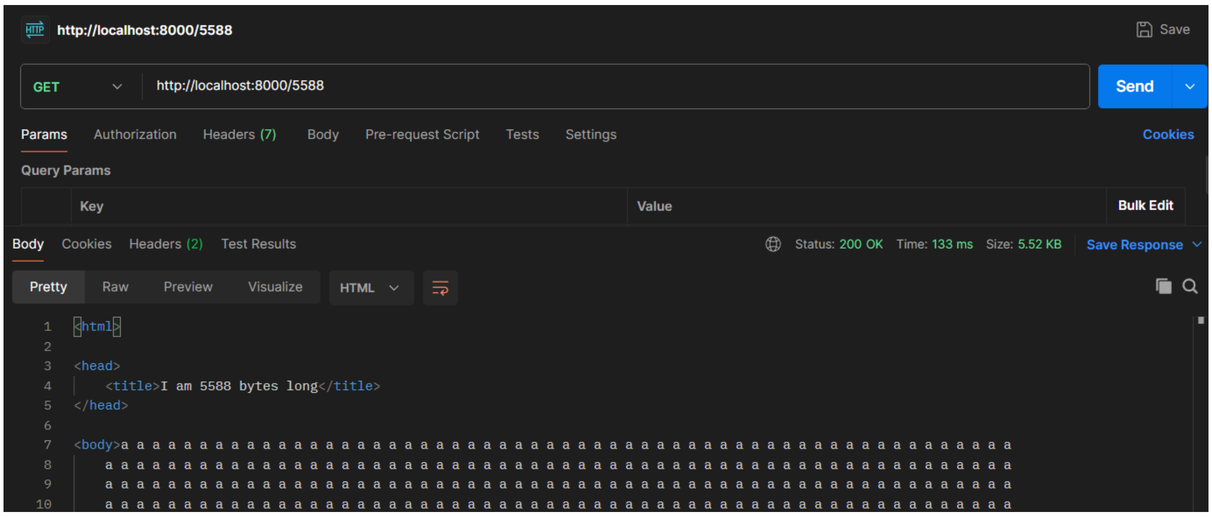
However, if the method type is an invalid method type that includes valid methods other than GET, the error message it sends should be a 400 Bad Request error, not a 501 error. To distinguish this, we do not need to send 501 to valid methods one by one, because the system automatically sends 501 errors to invalid methods. In addition, we will send 400 errors to methods that are not valid.

```
      valid_method = {"POST", "GET", "DELETE", "PUT", "PATCH", "HEAD",
"OPTIONS"} #postman valid methods.
              if method not in valid_method:
                  response = "HTTP/1.1 400 Bad Request\r\n\r\n<h1>400
Bad Request</h1>"
```

```
            client_socket.sendall(response.encode('utf-8'))
            return
```

OUTPUTS:

- A valid GET request with proper size is sent to the server.



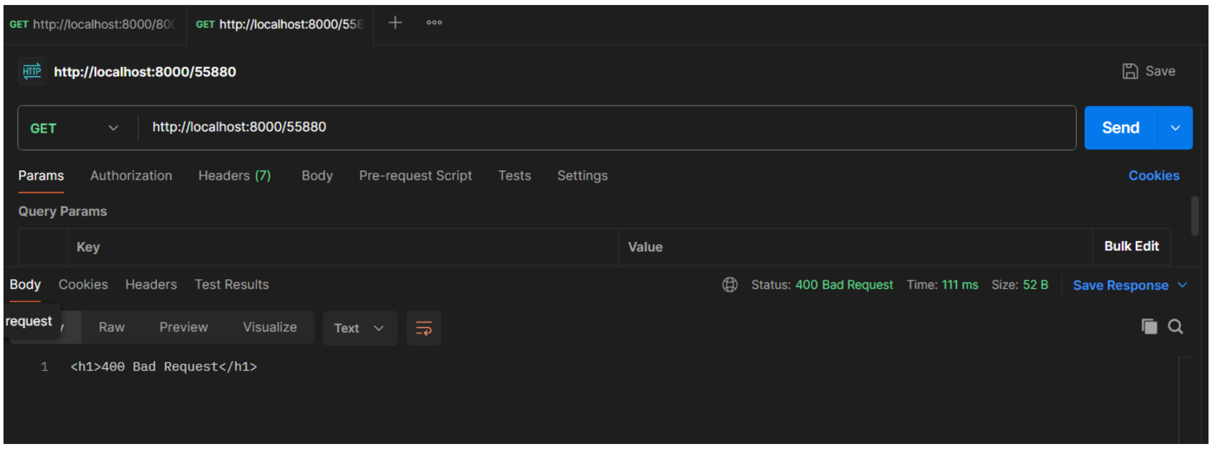
http://localhost:8000/5588

GET http://localhost:8000/5588

Status: 200 OK Time: 133 ms Size: 5.52 KB

```
<html>

<head>
    <title>I am 5588 bytes long</title>
</head>

<body>a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
    a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
    a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
    a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
```

- A valid GET request with excessive size is sent to the server.



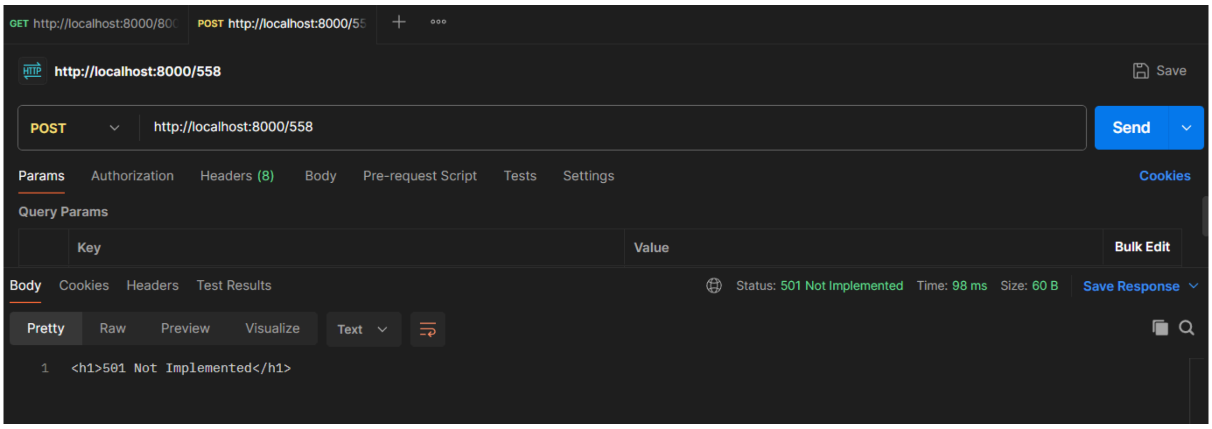
http://localhost:8000/55880

GET http://localhost:8000/55880

Status: 400 Bad Request Time: 111 ms Size: 52 B

```
<h1>400 Bad Request</h1>
```

- A valid request rather than GET with proper size is sent to the server.



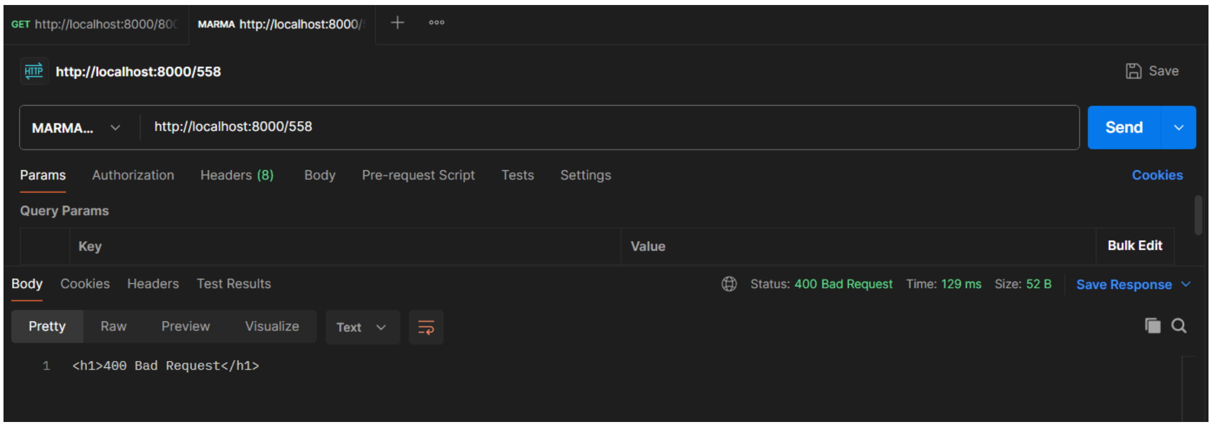
http://localhost:8000/558

POST http://localhost:8000/558

Status: 501 Not Implemented Time: 98 ms Size: 60 B

```
<h1>501 Not Implemented</h1>
```

- An invalid request with proper size is sent to the server.



URL: http://localhost:8000/558

Status: 400 Bad Request Time: 129 ms Size: 52 B

```
<h1>400 Bad Request</h1>
```

· **Proxy server implementation**

The proxy server is designed to act as an intermediary between clients and a target web server. It accepts HTTP client requests, validates and forwards them to the destination server, and then relays the responses back to the clients. The server handles multiple client connections concurrently using threads and ensures proper error handling and request validation.

Implementation Details

The implementation consists of three main components: Proxy Setup, Client Request Handling, Request Validation and Forwarding.

Proxy Setup

The proxy server operates on a specified port, listens for incoming connections, and creates a new thread for each client to handle requests concurrently. The start_proxy function initializes the server socket, binds it to all interfaces, and listens for connections.

```
def start_proxy(self):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
server_socket:
        server_socket.bind(('0.0.0.0', self.proxy_port))
        server_socket.listen(5)
        print(f"Proxy Server is running on port {self.proxy_port}")

        while True:
            client_socket, client_address = server_socket.accept()
            print(f"Accepted connection from {client_address}")
            threading.Thread(target=self.handle_client,
args=(client_socket,)).start()
```

Client Request Handling

Each client request is processed in the handle_client method. The method reads the HTTP request, extracts the request line, and parses the method and URI. Valid requests will be forwarded to the target server with forward method, while invalid ones receive error responses.

```
    def handle_client(self, client_socket):
        try:
            request = client_socket.recv(1024).decode('utf-8')
```

```
                print(f"Request: {request}")

                request_line = request.splitlines()[0]
                method, uri, _ = request_line.split(" ")
        # Send the response back to the client
                response = self.forward(uri, )

                client_socket.sendall(response.encode('utf-8'))
```

In the proxy client handling, we also need to set a new restriction for proxy to not send any request to the server that has more than 9999 size uri.

```
size = int(uri.lstrip('/'))
            if size > 9999: #size limitation
                response = "HTTP/1.1 414 URI Too Long\r\n\r\n<h1>URI
Too Long</h1>"
                client_socket.sendall(response.encode('utf-8'))
                return
```

Send Connection Request from POSTMAN:

curl (GET method)
```
    http://localhost:8888/2000 #HTTP Request
```

Start Proxy Server on Terminal:

bash
```
    PS C:\Users\emre> python proxy.py localhost 8000
    ->Proxy Server is running on port 8888 #terminal response
```

We need to define the server port number on the terminal while starting the proxy. We also need to define the host.

Request Validation and Forwarding

Valid requests are forwarded to the destination server using a new socket connection. The response is read and sent back to the client with forward method.

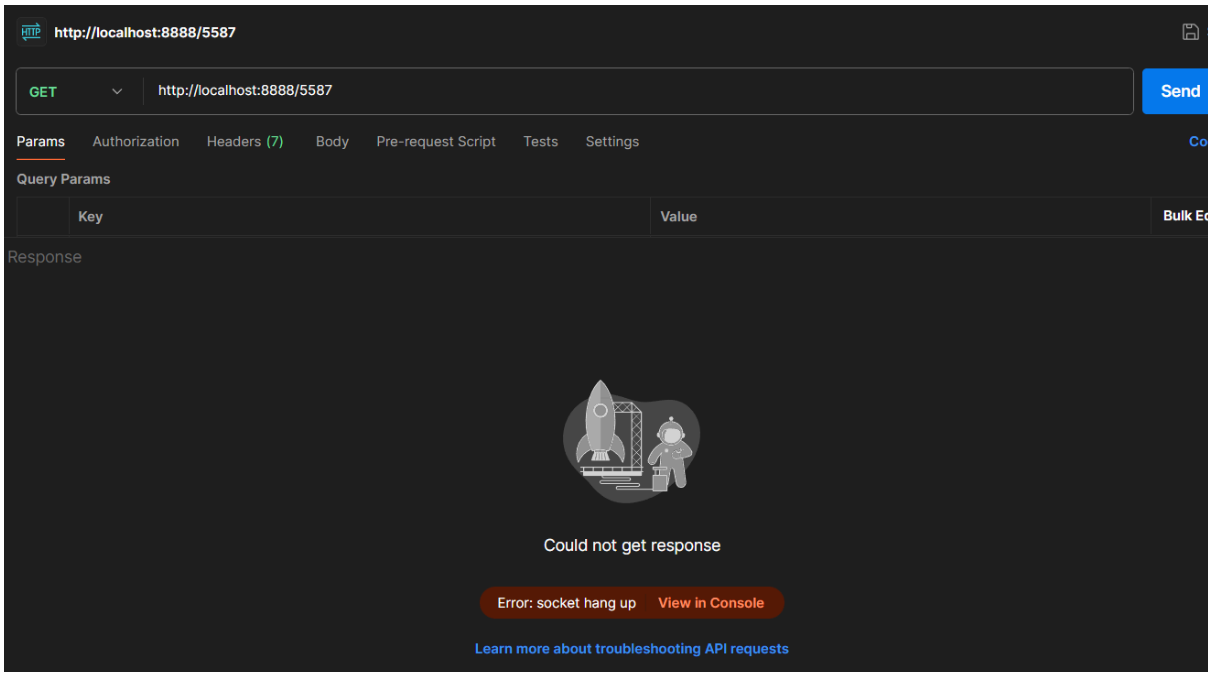
```
      #Function to forward data between client and server.
          def forward(self, uri):
              with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
server_socket:
                  server_socket.connect((self.server_host,
self.server_port))
```

```
                    request_line = f"GET {uri} HTTP/1.1\r\n"
                    headers = f"Host:
{self.server_host}:{self.server_port}\r\n\r\n"
                    server_socket.sendall(request_line.encode('utf-8') +
headers.encode('utf-8'))

                #forward the http request to the main server.
                response = b""

                while True:
                    #try:
                    request = server_socket.recv(4096)
                    if len(request) == 0:
                        break
                        response += request

                    print(f"debug - Proxy received response from
server:\n{response.decode('utf-8')}")
                    return response.decode('utf-8')
```

We begin by constructing a new socket using Python's `socket` module. This socket serves as our communication channel to the target server. To initiate the connection, we use the `connect` method, providing the target server's hostname (server_host) and port (server_port) as arguments.
After establishing the connection, we prepare the HTTP request that the proxy server forwards.
After receiving the complete response from the server, we decode it back into a string and return it to the handle_client method, which will relay it to the client.

OUTPUTS:

-A valid connection request to the proxy.

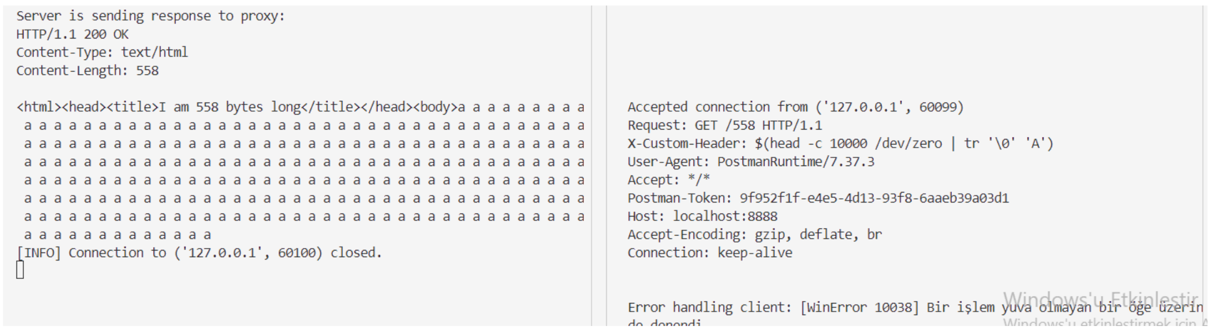


```
GET http://localhost:8888/5587
```

Could not get response

Error: socket hang up

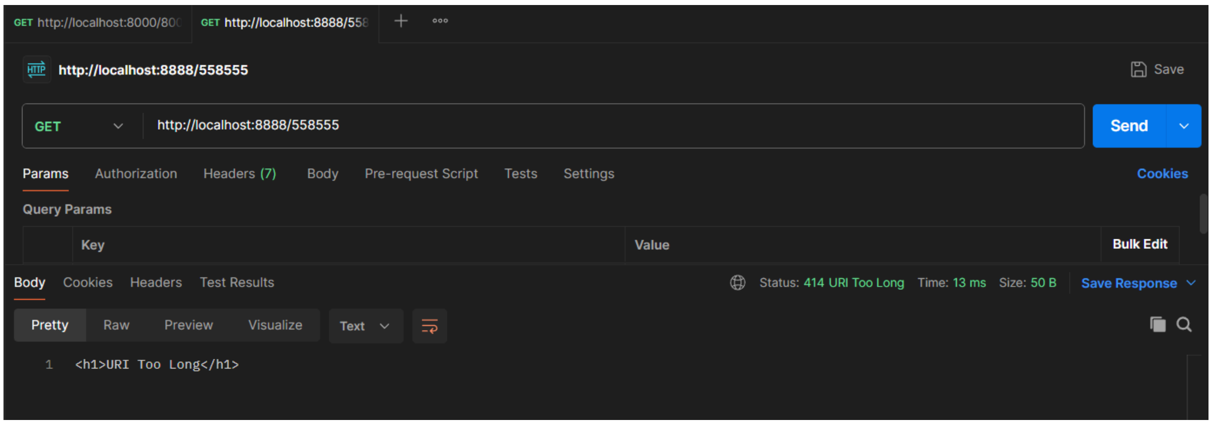
Since we had a problem with sending a proper response to the client, we get this error. We tried to debug it, however, we could not solve it.



```
Server is sending response to proxy:
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 558

<html><head><title>I am 558 bytes long</title></head><body>a a a a a a a a
 a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
 a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
 a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
 a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
 a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
 a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
 a a a a a a a a a a a a
[INFO] Connection to ('127.0.0.1', 60100) closed.
```

```
Accepted connection from ('127.0.0.1', 60099)
Request: GET /558 HTTP/1.1
X-Custom-Header: $(head -c 10000 /dev/zero | tr '\0' 'A')
User-Agent: PostmanRuntime/7.37.3
Accept: */*
Postman-Token: 9f952f1f-e4e5-4d13-93f8-6aaeb39a03d1
Host: localhost:8888
Accept-Encoding: gzip, deflate, br
Connection: keep-alive


Error handling client: [WinError 10038] Bir işlem yuva olmayan bir öğe üzerin
de denendi
```

-An invalid connection request with a long URI to the proxy.



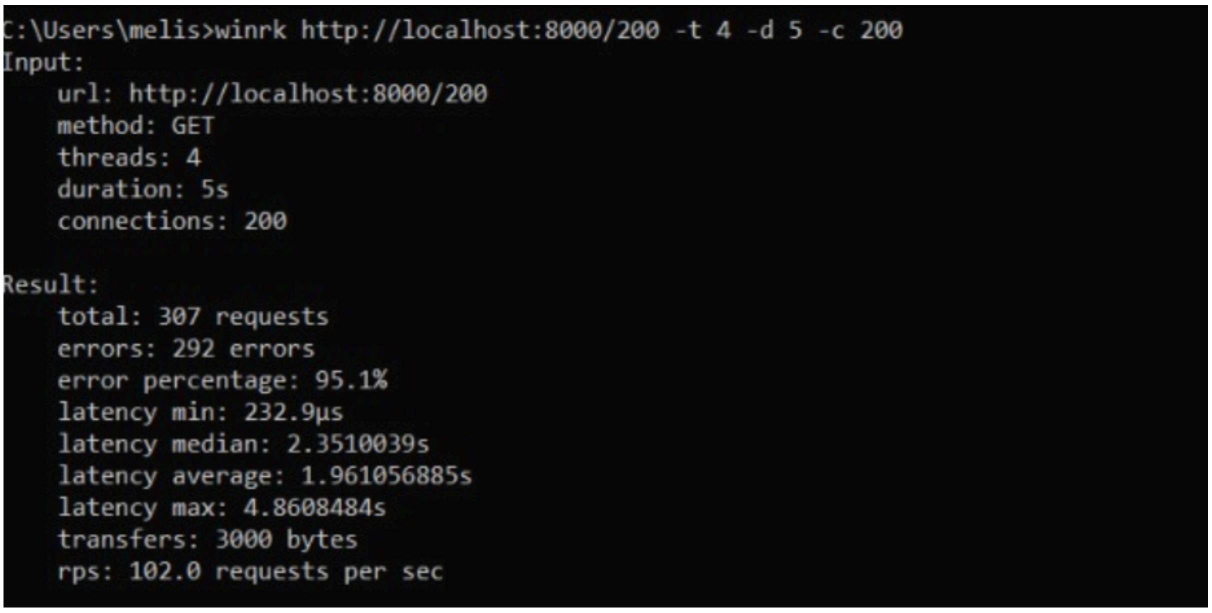
```
GET http://localhost:8888/558555
```

Status: 414 URI Too Long Time: 13 ms Size: 50 B

```
<h1>URI Too Long</h1>
```

3. PERFORMANCE TESTS AND RESULTS ·

HTTP server performance



```
C:\Users\melis>winrk http://localhost:8000/200 -t 4 -d 5 -c 200
Input:
    url: http://localhost:8000/200
    method: GET
    threads: 4
    duration: 5s
    connections: 200

Result:
    total: 307 requests
    errors: 292 errors
    error percentage: 95.1%
    latency min: 232.9µs
    latency median: 2.3510039s
    latency average: 1.961056885s
    latency max: 4.8608484s
    transfers: 3000 bytes
    rps: 102.0 requests per sec
```

URL : `http://localhost:8000/200`

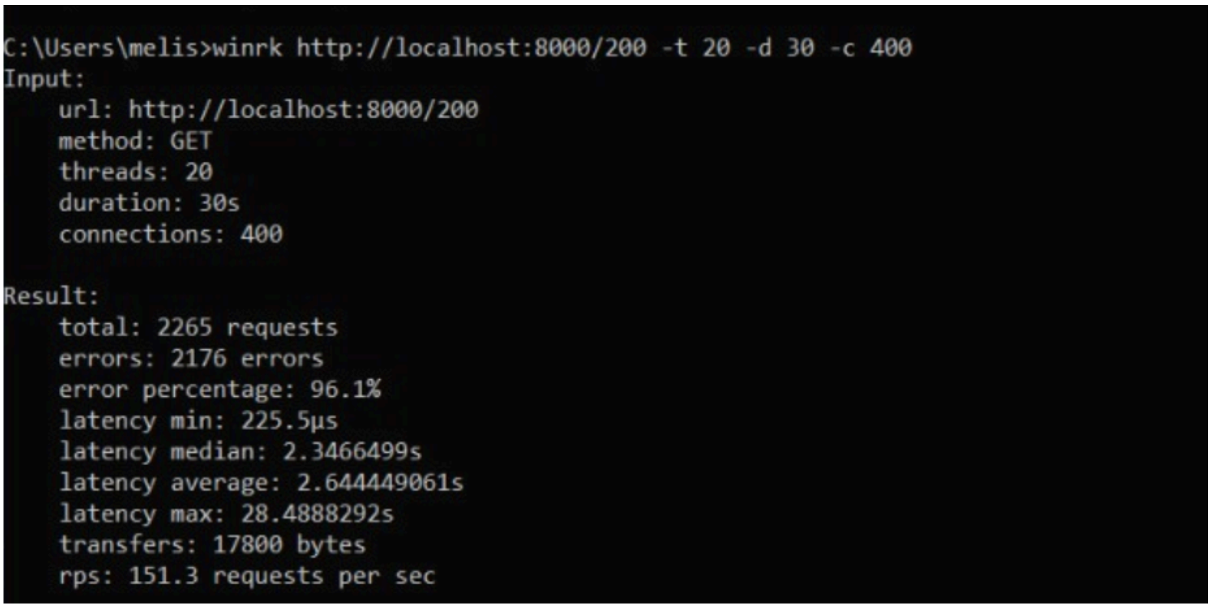
method: GET

threads: 4

duration: 5s

connections: 200

.



```
C:\Users\melis>winrk http://localhost:8000/200 -t 20 -d 30 -c 400
Input:
    url: http://localhost:8000/200
    method: GET
    threads: 20
    duration: 30s
    connections: 400

Result:
    total: 2265 requests
    errors: 2176 errors
    error percentage: 96.1%
    latency min: 225.5µs
    latency median: 2.3466499s
    latency average: 2.644449061s
    latency max: 28.4888292s
    transfers: 17800 bytes
    rps: 151.3 requests per sec
```

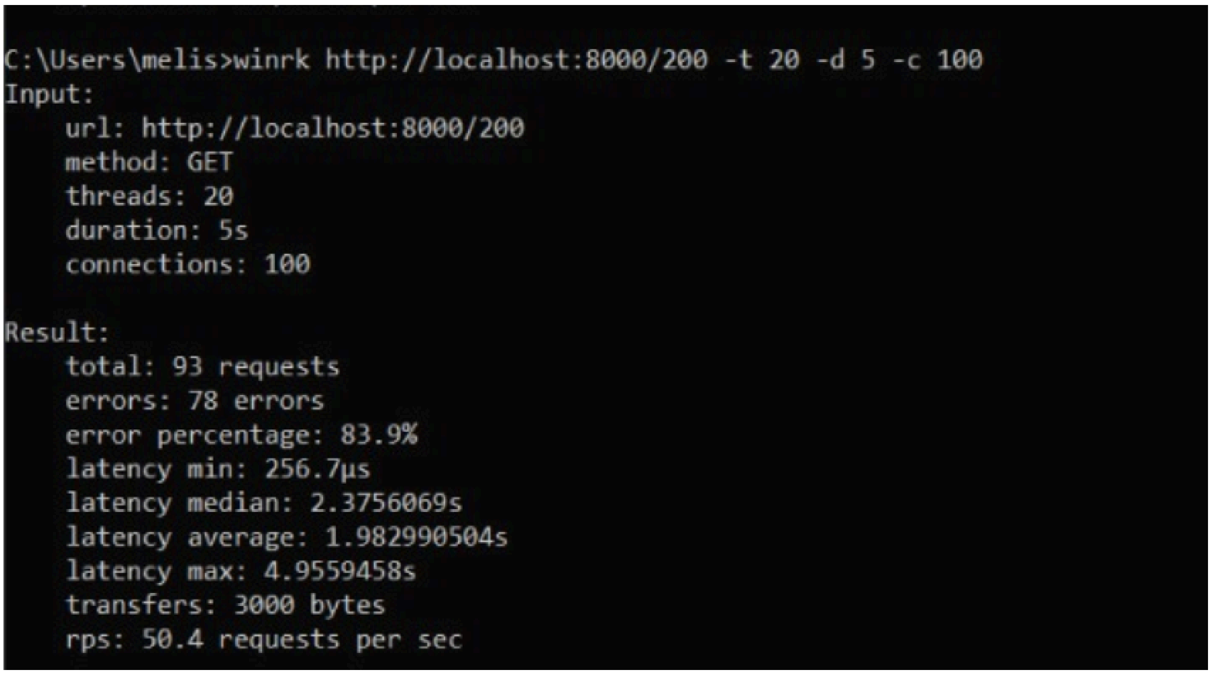
URL : `http://localhost:8000/200`

method: GET

threads: 20

duration: 30s

connections: 400



```
C:\Users\melis>winrk http://localhost:8000/200 -t 20 -d 5 -c 100
Input:
    url: http://localhost:8000/200
    method: GET
    threads: 20
    duration: 5s
    connections: 100

Result:
    total: 93 requests
    errors: 78 errors
    error percentage: 83.9%
    latency min: 256.7µs
    latency median: 2.3756069s
    latency average: 1.982990504s
    latency max: 4.9559458s
    transfers: 3000 bytes
    rps: 50.4 requests per sec
```

URL : `http://localhost:8000/200`

method: GET

threads: 20

duration: 5s

connections: 100

4. CONCLUSION

This project successfully implemented a multi-threaded HTTP server and a proxy server, showcasing key concepts of socket programming and concurrency. The HTTP server handled dynamic HTML responses efficiently, while the proxy server reliably forwarded requests and enforced URI restrictions.

Performance tests demonstrated the servers' scalability and ability to handle concurrent requests effectively. Despite minor challenges in debugging, the project provided valuable insights into network application design and testing. Future improvements could focus on optimizing proxy functionality and error handling