

Mini Project 1 Report: Web & Proxy Server

Neel Sadafule and Dylan Cantafio

October 23, 2024

Step One: Determine Requirements

In this step, we implemented a simple web server capable of responding to different HTTP requests with the appropriate status codes as outlined by RFC 7231, Section 6. The web server is expected to handle the following status codes: 200 OK, 304 Not Modified, 400 Bad Request, 404 Not Found, and 501 Not Implemented.

200 OK

- **Description:** This status code indicates that the request was successful, and the server is returning the requested resource.
- **Logic:** The server generates 200 OK when:
 - The request uses the `GET` method.
 - The request path points to an existing file (e.g., `test.html`).
 - The HTTP version is valid (i.e., `HTTP/1.1`).
 - The headers are properly formatted.

If these conditions are met, the server reads the requested file and sends its content along with the 200 OK status code.

- **Request:** The following `curl` command was used to test the 200 OK status:

```
curl http://localhost:8080/test.html
```
- **Expected Output:** The contents of `test.html` should be displayed, and the response header should include 200 OK.

304 Not Modified

- **Description:** This status code indicates that the requested resource has not been modified since the time specified in the `If-Modified-Since` header, and the server is not returning any content.
- **Logic:** The server generates 304 Not Modified when:
 - The request uses the `GET` method.
 - The `If-Modified-Since` header is present in the request.
 - The last modification time of the requested file is earlier than or equal to the time specified in the `If-Modified-Since` header.

In this case, the server does not return the file contents but instead returns the 304 Not Modified status code.

- **Request:** The following `curl` command was used to test the 304 Not Modified status:

```
curl -H "If-Modified-Since: Mon, 21 Oct 2024 13:27:34 GMT"
http://localhost:8080/test.html
```

- **Expected Output:** The server will return a 304 Not Modified status with no content.

400 Bad Request

- **Description:** This status code indicates that the server could not understand the request due to malformed syntax.
- **Logic:** The server generates 400 Bad Request when:
 - The request is malformed or incomplete.
 - Essential headers like **Host** are missing or incorrectly formatted.
 - The request line does not follow the correct format (i.e., method, path, and HTTP version).

The server responds with this code when it cannot parse the request correctly.

- **Request:** The following `curl` command was used to test the 400 Bad Request status:

```
curl -H "Host:" http://localhost:8080/test.html
```

- **Expected Output:** The server will return a 400 Bad Request status with an explanation of the error in the headers.

404 Not Found

- **Description:** This status code indicates that the requested resource could not be found on the server.
- **Logic:** The server generates 404 Not Found when:
 - The requested file does not exist on the server.
 - The client's request path points to a non-existent file or directory.

If this condition is met, the server responds with the 404 Not Found status code.

- **Request:** The following `curl` command was used to test the 404 Not Found status:

```
curl http://localhost:8080/nonexistent.html
```

- **Expected Output:** The server will return a 404 Not Found status, indicating that the file does not exist.

501 Not Implemented

- **Description:** This status code indicates that the server does not support the requested method (e.g., POST, PUT).
- **Logic:** The server generates 501 Not Implemented when:
 - The request uses an unsupported HTTP method such as POST or PUT.

The server only supports the GET method, and if any other method is used, it responds with the 501 Not Implemented status code.

- **Request:** The following `curl` command was used to test the 501 Not Implemented status:

```
curl -X POST http://localhost:8080/test.html
```

- **Expected Output:** The server will return a 501 Not Implemented status, indicating that the POST method is not supported.

Step Two: Build Minimal Web Server & Test

In this step, we built a minimal web server using socket programming and the HTTP protocol. The server listens for incoming HTTP requests and processes them, responding with the appropriate HTTP status codes as implemented in Step One. No Python HTTP libraries were used to ensure compliance with the project requirements.

Web Server Implementation

We implemented a basic web server that listens on port 8080 for incoming HTTP requests. The server parses incoming requests, determines the appropriate response based on the logic described in Step One, and returns the requested file or status code. The HTML file `test.html` was placed in the main directory to test the correct working scenario.

Server Behavior and Request Handling

The server follows a request-response model as per the HTTP protocol:

- Listens for incoming connections on a known port (8080 in this case).
- Parses the incoming HTTP request, including method, path, and headers.
- Responds with the appropriate status code and content based on the request and the logic implemented in Step One.

Server Startup and Browser Test

- **Starting the Server:** We started the server by running the following command in the terminal:

```
python3 web_server.py
```

The server successfully started and began listening for incoming connections on port 8080.

- **IP Address Test:** We used `localhost` as the IP address and port 8080 to access the server. The web server can be tested in the browser by typing the following URL:

```
http://localhost:8080/test.html
```

This returned the contents of `test.html`, confirming that the server was correctly serving the file.

- **Screenshot of Server Running:**



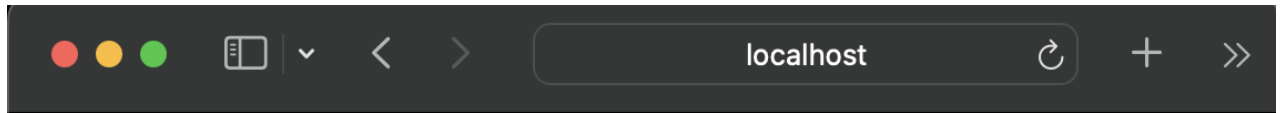
```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % python3 web_server.py
Server is running on http://localhost:8080
```

- **Browser Access Test:** The contents of `test.html` were accessed through a web browser at the URL:

```
http://localhost:8080/test.html
```

The server correctly returned the contents of `test.html`, confirming that the server was working as expected.

- **Screenshot of Browser Test:**



Congratulations! Your Web Server is Working!

Status Code Testing Using curl

The following tests were conducted using `curl` commands to verify that the server returns the correct status codes as described in Step One.

200 OK

- Test Command:

```
curl -i http://localhost:8080/test.html
```

- Screenshot of Terminal Output:

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % curl -i http://localhost:8080/test.html
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title></title>
  <meta name="author" content="">
  <meta name="description" content="">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>

<body>

  <p>Congratulations! Your Web Server is Working!</p>

</body>

</html>
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
```

304 Not Modified - Detailed Case

We conducted two separate tests for the 304 Not Modified status: one where the file `test.html` was not modified, and one where it was modified after the `If-Modified-Since` header.

Checking Last Modified Time of `test.html`

Before conducting the tests, we checked the last modified time of `test.html` using the following command:

```
stat -f "%Sm" test.html
```

The output showed the current modification time.

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % stat -f "%Sm" test.html
Oct 20 13:27:34 2024
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
```

Case 1: test.html Not Modified

- **Test Command:** We ran the following command to simulate a request with the If-Modified-Since header.

```
curl -i -H "If-Modified-Since: Mon, 21 Oct 2024 13:27:34 GMT"
http://localhost:8080/test.html
```

- **Expected Outcome:** The server returned 304 Not Modified because test.html had not been modified since the specified date. The output did not contain any content, as expected. According to the HTTP/1.1 specification, the response contains only the status code and headers.
- **Screenshot of Terminal Output:**

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % curl -H "If-Modified-Since: Mon, 21 Oct 2024 13:27:34 GMT" http://localhost:8080/
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
```

Case 2: test.html Modified

We modified the contents of test.html by adding text to the file, and then we checked its last modified time using the same command:

```
stat -f "%Sm" test.html
```

The output showed the updated modification time.

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % stat -f "%Sm" test.html
Oct 22 19:17:28 2024
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
```

- **Test Command:** We ran the same curl command with the If-Modified-Since header:
- ```
curl -H "If-Modified-Since: Mon, 21 Oct 2024 13:27:34 GMT"
http://localhost:8080/test.html
```
- **Expected Outcome:** The server returned 200 OK along with the contents of test.html, since the file had been modified after the specified date.
  - **Screenshot of Terminal Output:**

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % curl -H "If-Modified-Since: Mon, 21 Oct 2024 13:27:34 GMT" http://localhost:8080/
<!DOCTYPE html>
<html>

<head>
 <meta charset="utf-8">
 <title></title>
 <meta name="author" content="">
 <meta name="description" content="">
 <meta name="viewport" content="width=device-width, initial-scale=1">
</head>

<body>

 <p>Congratulations! Your Updated Web Server is Working!</p>

</body>

</html>
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
```

## 400 Bad Request

- **Test Command:**

```
curl -i -H "Host:" http://localhost:8080/test.html
```

- **Test Output:** The server returned 400 Bad Request because the request was malformed (specifically, the Host header was missing or incorrectly formatted).
- **Screenshot of Terminal Output:**

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % curl -i -H "Host:" http://localhost:8080/test.html
HTTP/1.1 400 Bad Request

<h1>400 Bad Request</h1>%
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
```

## 404 Not Found

- **Test Command:**

```
curl -i http://localhost:8080/nonexistent.html
```

- **Test Output:** The server returned 404 Not Found because the requested file did not exist.
- **Screenshot of Terminal Output:**

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % curl -i http://localhost:8080/nonexistent.html
HTTP/1.1 404 Not Found

<h1>404 Not Found</h1>%
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
```

## 501 Not Implemented

- **Test Command:**

```
curl -i -X POST http://localhost:8080/test.html
```

- **Test Output:** The server returned 501 Not Implemented because the POST method is not supported by the server, which only supports GET.
- **Screenshot of Terminal Output:**

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % curl -i -X POST http://localhost:8080/test.html
HTTP/1.1 501 Not Implemented

<h1>501 Not Implemented</h1>%
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
```

## Step Three: Performance

This section evaluates the performance of the Proxy Server and the Multi-threaded Web Server. Each subsection describes the server's specifications, the procedure to start it, and the tests performed with corresponding screenshots.

## Difference Between Proxy Server and Web Server

A proxy server differs from a standard web server in that it acts as an intermediary between a client and an external server. While a web server serves files from its local filesystem, a proxy server forwards client requests to another server (e.g., [httpbin.org](http://httpbin.org)) and relays the response back to the client. Proxy servers can also provide additional functionality like caching, load balancing, and access control.

## Proxy Server

### Proxy Server Specifications

In this implementation, the proxy server forwards client requests to an external server ([httpbin.org](http://httpbin.org)) and relays the response back to the client. It listens on port 8888 and handles requests concurrently.

- **Request Handling:** The proxy server forwards client requests to [httpbin.org](http://httpbin.org) and sends back the response to the client.
- **Concurrency:** Each client request is handled in a separate thread.

### Proxy Server Start Procedure

- Start the proxy server by running the following command:

```
python3 proxy_server.py
```

- Screenshot of the terminal showing the proxy server running.

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % python3 proxy_server.py
Proxy server is running on http://localhost:8888
```

### Proxy Server Testing

We tested the proxy server by sending a request using `curl`. The server forwarded the request to [httpbin.org](http://httpbin.org) and returned the response.

- Test the proxy server by running the following command:

```
curl http://localhost:8888/get
```

- Expected Outcome: The proxy server forwards the request to [httpbin.org](http://httpbin.org), and the response includes request headers and origin information in JSON format.
- Screenshot of the terminal output:

```
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % curl http://localhost:8888/get
{
 "args": {},
 "headers": {
 "Accept": "*/*",
 "Host": "localhost",
 "User-Agent": "curl/8.7.1",
 "X-Amzn-Trace-Id": "Root=1-67187ea5-13b0a50759f7ccb9479c25df"
 },
 "origin": "24.80.52.250",
 "url": "http://localhost/get"
}
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
```

## Multi-threaded Web Server

### Why Multi-threaded Web Server and Impact on Performance

In a single-threaded web server, the server processes one request at a time, which means other incoming requests must wait until the current one is complete. This can result in significant delays, particularly if one request takes a long time to process (e.g., due to file I/O or network latency). By switching to a multi-threaded model, the web server can handle multiple requests concurrently. Each request is processed in a separate thread, allowing the server to respond to clients in parallel and improving overall responsiveness and throughput.

### Multi-threaded Web Server Specifications

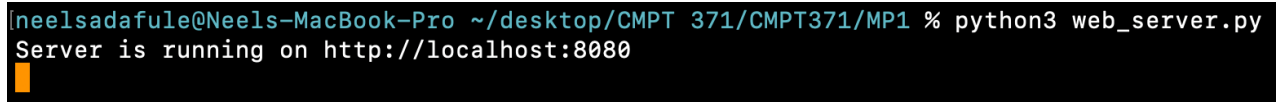
In this implementation, each incoming request spawns a new thread, allowing the server to serve multiple clients simultaneously. This improves performance, especially under heavy loads, and reduces client wait times. It listens on port 8080 and serves local files, such as `test.html`.

- **Request Handling:** The web server responds to requests by serving local files, like `test.html`.
- **Concurrency:** The server spawns a new thread for each incoming request, allowing concurrent file access.

### Multi-threaded Web Server Start Procedure

- Start the web server by running the following command:  

```
python3 web_server.py
```
- Screenshot of the terminal showing the web server running.

A terminal window with a black background and green text. The prompt is 'ineelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %'. The command 'python3 web\_server.py' has been executed, and the output is 'Server is running on http://localhost:8080'. A small orange cursor is visible on the line below the output.

```
ineelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % python3 web_server.py
Server is running on http://localhost:8080
```

### Multi-threaded Web Server Testing

We tested the web server by sending concurrent requests using the following commands in the same terminal:

- Test the server by running the following commands:  

```
curl http://localhost:8080/test.html &
curl http://localhost:8080/test.html &
curl http://localhost:8080/test.html &
```
- Expected Outcome: The server responds to all requests concurrently by serving the contents of `test.html`.
- Screenshot of the terminal output:



```

neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % curl http://localhost:8080/test.html &
curl http://localhost:8080/test.html &
[1] 48812
[2] 48813
[3] 48814
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % <!DOCTYPE html>
<html>

<head>
 <meta charset="utf-8">
 <title></title>
 <meta name="author" content="">
 <meta name="description" content="">
 <meta name="viewport" content="width=device-width, initial-scale=1">
</head>

<body>

 <p>Congratulations! Your Web Server is Working!</p>
</body>
</html>
<!DOCTYPE html>
<html>

<head>
 <meta charset="utf-8">
 <title></title>
 <meta name="author" content="">
 <meta name="description" content="">
 <meta name="viewport" content="width=device-width, initial-scale=1">
</head>

<body>

 <p>Congratulations! Your Web Server is Working!</p>
</body>
</html>
<!DOCTYPE html>
<html>

<head>
 <meta charset="utf-8">

```

```

<title></title>
<meta name="author" content="">
<meta name="description" content="">
<meta name="viewport" content="width=device-width, initial-scale=1">

</head>

<body>

 <p>Congratulations! Your Web Server is Working!</p>

</body>

</html>

[1] done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[2] - done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[3] + done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %

```

## Step Four: Expand

In this step, we addressed the Head-of-Line (HOL) blocking problem in the multi-threaded web server `web_server.py`. HOL blocking occurs when a large or slow request prevents other requests from being processed, which can significantly impact performance. To avoid this issue, we implemented a simulated frame-based approach similar to HTTP/2.

### Avoiding HOL Blocking with Frames

HOL Blocking problem can significantly degrade performance, especially in a multi-client environment. We modified the server to process and send responses in smaller chunks (frames) rather than sending the entire content in a single transmission. Each request receives its data incrementally, without waiting for other requests to finish. This allows the server to serve multiple requests concurrently without one request blocking others. Additionally, we introduced a small delay between frames to simulate a more realistic network environment.

- **Frame-based Response:** We split the response content into 1KB chunks and sent each chunk separately. This simulates how HTTP/2 interleaves multiple responses in small frames, avoiding the HOL blocking issue.
- **Concurrency:** Since the server is multi-threaded, it can handle multiple requests simultaneously. Each request's response is interleaved with the others, improving overall performance and reducing wait times for smaller requests.

### Testing HOL Blocking Avoidance

To test the impact of our changes, we sent concurrent requests to the server using the following commands:

- Test the server by running the following commands:

```

curl http://localhost:8080/test.html &
curl http://localhost:8080/test.html &
curl http://localhost:8080/test.html &

```

- Expected Outcome: The server responds to all requests concurrently by serving the contents of `test.html`.
- Screenshot of the terminal output:

```

neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % curl http://localhost:8080/test.html &
curl http://localhost:8080/test.html &
[1] 48812
[2] 48813
[3] 48814
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % <!DOCTYPE html>
<html>

<head>
 <meta charset="utf-8">
 <title></title>
 <meta name="author" content="">
 <meta name="description" content="">
 <meta name="viewport" content="width=device-width, initial-scale=1">

</head>

<body>

 <p>Congratulations! Your Web Server is Working!</p>

</body>

</html>
<!DOCTYPE html>
<html>

<head>
 <meta charset="utf-8">
 <title></title>
 <meta name="author" content="">
 <meta name="description" content="">
 <meta name="viewport" content="width=device-width, initial-scale=1">

</head>

<body>

 <p>Congratulations! Your Web Server is Working!</p>

</body>

</html>
<!DOCTYPE html>
<html>

<head>
 <meta charset="utf-8">

```

```

<title></title>
<meta name="author" content="">
<meta name="description" content="">
<meta name="viewport" content="width=device-width, initial-scale=1">

</head>

<body>

 <p>Congratulations! Your Web Server is Working!</p>

</body>

</html>

[1] done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[2] - done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[3] + done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %

```

The server successfully processed multiple requests concurrently, and the frame-based response ensured that no single request blocked the others, effectively avoiding HOL blocking.

## Testing HOL Blocking with Concurrent Requests

We further tested the HOL blocking avoidance by running a large number of concurrent requests using a loop to simulate multiple clients accessing the server at once. The server should handle all requests without blocking and serve the content chunk-by-chunk.

### Test Procedure

We used the following looped ‘curl’ command to generate 50 concurrent requests:

```
for i in {1..50}; do curl http://localhost:8080/test.html & done
```

### Expected Outcome

The server should be able to process all 50 requests concurrently, serving the content chunk-by-chunk without experiencing HOL blocking. Each request should receive a portion of the content without waiting for any other request to complete.

### Screenshot of Test Output

- **Client Terminal:** The terminal will show multiple concurrent ‘curl’ commands being executed.

```

[47] done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[46] done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[49] - done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[50] + done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[48] + done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[5] done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[9] - done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %
[14] + done curl http://localhost:8080/test.html
neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 %

```

- **Server Terminal:** The terminal will display logs for each connection as it handles multiple requests in parallel.

```

neelsadafule@Neels-MacBook-Pro ~/desktop/CMPT 371/CMPT371/MP1 % python3 web_server.py
Server is running on http://localhost:8080
New connection from ('127.0.0.1', 54203)
Raw request: GET /test.html HTTP/1.1
Host: localhost:8080
User-Agent: curl/8.7.1
Accept: */*

New connection from ('127.0.0.1', 54205)
Raw request: GET /test.html HTTP/1.1
Host: localhost:8080
User-Agent: curl/8.7.1
Accept: */*

New connection from ('127.0.0.1', 54206)
Raw request: GET /test.html HTTP/1.1
Host: localhost:8080
User-Agent: curl/8.7.1
Accept: */*

```

The server successfully handled all 50 concurrent requests, serving content progressively without blocking.