MP1 Report

Paul Pham, Khang Phan

I. Status code generation logic:

1) Status code 200 OK:

Meaning:

The client's request has succeeded. The server locates the requested file and sends it back with this status code.

Logic:

a) Requirements:

- If the HTTP request is properly formatted (e.g uses a supported version like HTTP/1.1).
- The requested file exists in the server's root directory.
- The file is readable (no access restrictions).

The server would respond with:

HTTP/1.1 200 OK

Content-Type: text/html

Content-Length: [file_size]

Last-Modified: [date]

b) Request from client:

GET /test.html HTTP/1.1

Host: localhost:12000

Connection: close

2) Status code 304:

Meaning:
The client indicates that it has a cached version of the resource and asks if it's still valid. If the file has not been modified since the date in If-Modified-Since, the server returns 304 Not Modified without sending the file again.

Logic:

a) Requirements:
- The request includes the If-Modified-Since header.
- The server compares this date with the file's actual modification timestamp.
- If the file has not changed since that date, respond with:

The server would respond with:

HTTP/1.1 304 Not Modified

b) Request from client:

GET /test.html HTTP/1.1

Host: localhost:12000

If-Modified-Since: Thu, 01 Jan 1970 00:00:00 GMT

Connection: close

3) Status code 403:

Meaning:
The server understands the request but refuses to authorize

Logic:

a) Requirements:
- The requested resource exists but is marked as restricted.

- For example, the file is inside a protected folder (like /private/), or its name contains "secret".

The server would respond with:

HTTP/1.1 403 Forbidden

b) Request from client:

GET /secret.html HTTP/1.1

Host: localhost:12000

Connection: close

4) Status code 404:

Meaning:

The requested resource could not be found on the server.

Logic:

a) Requirements:
   - The HTTP request is valid and uses a supported version.
   - The requested file does **not exist** in the server's directory.

The server would respond with:

HTTP/1.1 404 Not Found

Content-Type: text/html

b) Request from client:

GET /notfound.html HTTP/1.1

Host: localhost:12000

Connection: close

5) Status code 505:

Meaning:

The server does not support the HTTP protocol version used in the request.

Logic:

a) Requirements:

- When the request line specifies a version other than HTTP/1.0 or HTTP/1.1 (e.g., HTTP/2.0).

The server would respond with:

HTTP/1.1 505 HTTP Version Not Supported

b) Request from client:

GET /test.html HTTP/2.0

Host: localhost:12000

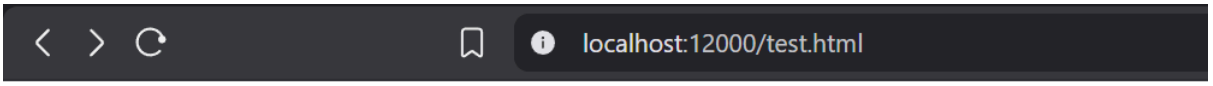
Connection: close

II. Test procedures of the webserver:

1) Test status code 200:

```
C:\Users\User>curl -v http://localhost:12000/test.html
* Host localhost:12000 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:12000...
*   Trying 127.0.0.1:12000...
* Connected to localhost (127.0.0.1) port 12000
* using HTTP/1.x
> GET /test.html HTTP/1.1
> Host: localhost:12000
> User-Agent: curl/8.14.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Content-Type: text/html
< Content-Length: 308
< Last-Modified: Thu, 23 Oct 2025 21:44:58 GMT
<
<!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>
* Connection #0 to host localhost left intact
```

Browser test:



Congratulations! Your Web Server is Working!

2) Test status code 304:

```
C:\Users\User>curl -v -H "If-Modified-Since: Sat, 25 Oct 2025 18:00:00 GMT" http://localhost:12000/test.html
* Host localhost:12000 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:12000...
*   Trying 127.0.0.1:12000...
* Connected to localhost (127.0.0.1) port 12000
* using HTTP/1.x
> GET /test.html HTTP/1.1
> Host: localhost:12000
> User-Agent: curl/8.14.1
> Accept: */*
> If-Modified-Since: Sat, 25 Oct 2025 18:00:00 GMT
>
< HTTP/1.1 304 Not Modified
< Content-Type: text/html
<
* Connection #0 to host localhost left intact
```

If content was modified, the server will send the regular 200 OK with full file content like part 1.

3) Test status code 403:

For this test case, we first set the read permission of the file “testForbidden.html” to restrict users from reading.

```
C:\Users\User>curl -v http://localhost:12000/testForbidden.html
* Host localhost:12000 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:12000...
*   Trying 127.0.0.1:12000...
* Connected to localhost (127.0.0.1) port 12000
* using HTTP/1.x
> GET /testForbidden.html HTTP/1.1
> Host: localhost:12000
> User-Agent: curl/8.14.1
> Accept: */*
>
< HTTP/1.1 403 Forbidden
< Content-Type: text/html
* no chunk, no close, no size. Assume close to signal end
<
<html><body><h1>403 Forbidden</h1><p>You do not have permission to access this file.</p></body></html>* abort upload
* shutting down connection #0
```

4) **Test status code 404:**

```
C:\Users\User>curl -v http://localhost:12000/random_file.html
* Host localhost:12000 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:12000...
*   Trying 127.0.0.1:12000...
* Connected to localhost (127.0.0.1) port 12000
* using HTTP/1.x
> GET /random_file.html HTTP/1.1
> Host: localhost:12000
> User-Agent: curl/8.14.1
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Content-Type: text/html
* no chunk, no close, no size. Assume close to signal end
<
<html><body><h1>404 Not Found</h1><p>The requested file does not exist on the server.</p></body></html>* abort upload
* shutting down connection #0
```

5) **Test status code 505:**

We used netcat to create the HTTP/2.0 request and send it to our server

```
C:\Users\User>echo "GET /test.html HTTP/2.0`r`nHost: localhost`r`nConnection: close`r`n`r`n" | ncat localhost 12000
HTTP/1.1 505 HTTP Version Not Supported
Content-Type: text/html

<html><body><h1>505 HTTP Version Not Supported</h1><p>The server only supports HTTP/1.0 and HTTP/1.1.</p></body></html>
```

III. Proxy Server:

Part A:

1) **What is different in request handling in a proxy server and a web server that hosts your files?**

Web Server hosts and serves local files like test.html, and Proxy server acts as an intermediary - forwarding client requests to the Web server and retrieving the response from the Web Server and returning back to the client.

The Web Server generates the HTTP response itself (200, 304, 403, 404, 500), and the proxy server only acts as an intermediary routing request to server and routing request to client.

The web server reads files from its own disk. On the other hand, the proxy server opens a new TCP connection to the destination host/port, sends the same request (after stripping the absolute URL), and relays the reply.

2) **Minimal Proxy Server specification:**

	Specification
1. Socket Setup	Create a TCP socket on the proxy host (e.g., localhost:12001), SO_REUSEADDR = 1, listen(1) for clients.
2. Accept Connection	Wait for an incoming HTTP client (browser or curl -x).
3. Receive Request	Read the first 4 KB of the client's HTTP request.
4. Parse Target Host and Path	From the request line GET http://host:port/path HTTP/1.1, extract host → localhost, port → 12000, path → /test.html.
5. Rewrite Request Line	Change absolute URL to relative path: GET /test.html HTTP/1.1 (per RFC 7230 §5.3.2).
6. Connect to Destination	Open a new TCP connection to (host, port) = (localhost, 12000).
7. Forward Request	Send the rewritten HTTP request to the destination web server.
8. Relay Response	Read all response chunks (recv(4096) loop) from the web server and immediately forward them to the client.
9. Error Handling	If connection fails, return HTTP/1.1 502 Bad Gateway.
10. Logging	Print request line and connection info for verification.

Part B:

Testing example: (Proxy server is hosted on 12001 and the web server is hosted on 12000)

```
C:\Users\User>curl -x localhost:12001 http://localhost:12000/test.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>

C:\Users\User>
```

Browser based test:

Congratulations! Your Web Server is Working!

Part C:

Multithreaded:

The proxy server is multithreaded so that many clients can concurrently send requests and receive responses from the proxy server.

Example:

```
(base) PS C:\Users\User\Desktop\371\371-MP1> python3 proxyServer.py
Proxy server listening on port 12001 ...

[+] New connection from ('127.0.0.1', 15890)
[Active Threads: 1]

[+] New connection from ('127.0.0.1', 15892)[Active Threads: 2]

[('127.0.0.1', 15890)] Request line: GET http://localhost:12000/test.html HTTP/1.1
[('127.0.0.1', 15890)] Response relayed successfully.
[-] Connection closed: ('127.0.0.1', 15890)
[('127.0.0.1', 15892)] Request line: GET http://localhost:12000/test.html HTTP/1.1
[('127.0.0.1', 15892)] Response relayed successfully.
[-] Connection closed: ('127.0.0.1', 15892)
```

We made the proxy server to be multithreaded instead of the web server since the proxy server is the first point of contact when the client interacts with the server. Additionally, the cache is usually at the proxy server, so it would make sense to implement multithreading at the proxy server.

How it is multithreaded:

```
# Create and start a new thread for each client
thread = threading.Thread(target=handle_client, args=(client_conn, client_addr))
thread.start()
```

We use Python's threading module to create a new thread for each client connection, allowing every request to be handled independently, making the proxy server multithreaded and able to handle multiple clients at the same time.

Multithreaded performance:

The reason why multithreaded performance is better is because in a single threaded server, only 1 request can be processed at a time and new clients have to wait until the current one finishes. On the other hand, multithreaded servers handle multiple requests simultaneously, where each client is served by a different thread. This significantly improves the performance of the server, as users won't have to queue for a long time waiting for the single thread to open up.

IV. HOL Problem

Steps we implemented on top of the proxy server to solve the HOL problem:

1) Frame Splitting: Instead of processing entire requests at once, we split each request into small frames. (idea taken from CMPT 371 course slides, Application Layer, Slide 38)
2) Priority Queue: All frames from different requests go into a single priority queue that processes them in round robin order (ensuring all frames are processed fairly).
3) Independent Processing: Since each frame is processed separately, so a slow request doesn't block fast requests

How It Solves HOL Blocking:

Before (HOL Problem):
Request 1 (slow) → Request 2 (fast) → Request 3 (fast)
Request 2 and 3 had to wait for Request 1 to finish completely

After (Frame Solution):
Frame 1 from Request 1 → Frame 1 from Request 2 → Frame 1 from Request 3 → Frame 2 from Request 1...
Fast requests can finish while slow requests are still being processed.

References:

Also mentioned in the README.md file in the zip.
[1] IETF, "RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," Internet Engineering Task Force, 2014. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7231

[2] IETF, "RFC 7540: Hypertext Transfer Protocol -- HTTP/2," Internet Engineering Task Force, 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7540

[3] IETF, "RFC 3040: Internet Web Replication and Caching Taxonomy," Internet Engineering Task Force, 2001. [Online]. Available: https://www.ietf.org/rfc/rfc3040.txt