Introduction

This report documents the implementation and testing of a minimal web server and a proxy server. The web server processes and responds to HTTP requests with appropriate status codes. The proxy server forwards requests and caches responses to enhance performance.

Requirements

The web server must handle HTTP requests and generate specific status codes based on the request details. Here are the specifications for each status code:

200 OK

- ➔ **Requirement**: A valid GET request for an existing file.
- ➔ **Logic**: If the requested file exists and there are no errors in the request, respond with a 200 OK status code and the file's content.

304 Not Modified

- ➔ **Requirement**: A GET request with an “If-Modified-Since” header for a file that hasn't been modified since the specified date.
- ➔ **Logic**: Compare the file's last modified date with the date specified in the “If-Modified-Since” header. If the file has not been modified since that date, respond with a 304 Not Modified status code.

400 Bad Request

- ➔ **Requirement**: Malformed HTTP request.
- ➔ **Logic**: If the request is missing required headers or has syntax errors, respond with a 400 Bad Request status code.

403 Forbidden

- ➔ **Requirement**: A GET request for a forbidden file.
- ➔ **Logic**: If the requested file is forbidden, respond with a 403 Forbidden status code.

404 Not Found

- ➔ **Requirement**: A GET request for a non-existent file.
- ➔ **Logic**: If the requested file does not exist, respond with a 404 Not Found status code.

Web Server Implementation and testing

Web Server implementation (web_server.py)

The main loop listens for incoming connections and creates a separate thread for each client using the “threading” module. Each thread calls the “handle_request” function to handle the client's request (GET, HEAD) independently. “handle_request” parses the request, validates it, serves static files, checks for If-Modified-Since headers, and sends appropriate responses (200 OK, 304 Not Modified, 400 Bad Request, 403 Forbidden, 404 Not Found, 500 Internal Server Error) depending on the situation.

Testing the Web Server

- **Test 1 (200 OK):**
  - **Request**: `curl -i http://localhost:8080/test.html`
  - **Expected Response**: `HTTP/1.1 200 OK\n\n{content of test.html}`
  - **Result**:

```
$ curl -i http://localhost:8080/test.html
HTTP/1.1 200 OK
Last-Modified: Fri, 14 Jun 2024 10:23:44 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>
```

- **Test 2 (304 Not Modified):**
  - **Request**: `curl -I -H "If-Modified-Since: Sat, 15 Jun 2024 07:28:00 GMT" http://localhost:8080/test.html`

- **Expected Response**: `HTTP/1.1 304 Not Modified`
- **Result**:

```
$ curl -I -H "If-Modified-Since: Sat, 15 Jun 2024 07:28:00 GMT" http://localhost:8080/test.html
HTTP/1.1 304 Not Modified
```

- **Test 3 (400 Bad Request):**
  - **Request**: `curl -I -H "Random: Request" http://localhost:8080/test.html`
  - **Expected Response**: `HTTP/1.1 400 Bad Request`
  - **Result**:

```
$ curl -I -H "Random: Request" http://localhost:8080/test.html
HTTP/1.1 400 Bad Request
```

- **Test 4 (403 Forbidden):**
  - **Request**: `curl -i http://localhost:8080/forbidden.html`
  - **Expected Response**: `HTTP/1.1 403 Forbidden\n\nForbidden`
  - **Result**:

```
$ curl -i http://localhost:8080/forbidden.html
HTTP/1.1 403 Forbidden

Forbidden
```

- **Test 5 (404 Not Found):**
  - **Request**: `curl -i http://localhost:8080/nonexistent.html`
  - **Expected Response**: `HTTP/1.1 404 Not Found\n\nNot Found`
  - **Result**:

```
$ curl -i http://localhost:8080/nonexistent.htm
HTTP/1.1 404 Not Found

Not Found
```

Performance

Proxy Server Specifications

A proxy server acts as an intermediary between clients and the main server. The minimal proxy server implemented here supports basic functionality such as request forwarding, response caching, and handling of “If-Modified-Since” headers.

Requirements for the Proxy Server

- ➔ **Request Handling:**
  - ◆ The proxy server must be able to parse incoming HTTP requests, extract the method, full URL, and HTTP version.
  - ◆ It should validate the HTTP version to ensure compatibility (HTTP/1.1 or HTTP/1.0).
- ➔ **URL Parsing:**
  - ◆ The server should parse the full URL to extract the hostname, optional port number, and path.
  - ◆ Regular expressions are used for parsing, ensuring flexibility and accuracy in extracting these components.
- ➔ **Header Handling:**
  - ◆ The proxy should correctly interpret and manage headers like “If-Modified-Since”, allowing conditional requests and handling caching based on modification times.
- ➔ **Caching Mechanism:**
  - ◆ Responses from remote servers can be cached locally to improve performance and reduce bandwidth usage for subsequent identical requests within a certain timeframe.
  - ◆ The cache timeout in this implementation is set to 5 minutes (`timedelta(minutes=5)`), ensuring freshness of cached responses.
- ➔ **Error Handling:**
  - ◆ Proper error handling mechanisms are in place to respond appropriately to various scenarios:
    - Malformed requests (`400 Bad Request`).
    - Internal server errors (`500 Internal Server Error`).
    - Other exceptions that may occur during request processing.

Testing the Proxy Server

- **Test 1 (Forwarding Requests):**
  - **Request**: `curl -x http://localhost:8888 http://localhost:8080/test.html`

- **Expected Response**: The content of http://localhost:8080/test.html
- **Result**:

```
$ curl -x http://localhost:8888 http://localhost:8080/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>
```

- **Test 2 (Caching and Serving from Cache):**
  - **Initial Request**: `curl -x http://localhost:8888 http://localhost:8080/test.html`
  - **Expected Response**: The content of http://localhost:8080/test.html
  - **Subsequent Request within 5 minute**: `curl -x http://localhost:8888 http://localhost:8080/test.html`
  - **Expected Response**: The cached content of http://localhost:8080/test.html
  - **Result**: from the client side

```
$ curl -x http://localhost:8888 http://localhost:8080/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>
```

From the proxy

```
Proxy server started on port 8888
Connection from ('127.0.0.1', 50877)
Connection from ('127.0.0.1', 50879)
cached response
```

- **Test 3 (Handling If-Modified-Since Header):**
  - **Request**: `curl -x http://localhost:8888 -I -H "If-Modified-Since: Sat, 15 Jun 2024 07:28:00 GMT" http://localhost:8080/test.html`
  - **Expected Response**: Forwarded to the destination server, response based on the actual modification time.
  - **Result**:

```
$ curl -x http://localhost:8888 -I -H "If-Modified-Since: Sat, 15 Jun 2024 07:28:00 GMT" http://localhost:8080/test.html
HTTP/1.1 304 Not Modified
```

- **Test 5 (500 Internal Server Error):**
  - **Request**: curl -x http://localhost:8888 http://localhost/test.html
  - **Expected Response**: `HTTP/1.1 500 Internal Server Error\n\n{error message}`
  - **Result**:

```
$ curl -v -x http://localhost:8888 http://localhost/test.html
*   Trying 127.0.0.1:8888...
* Connected to localhost (127.0.0.1) port 8888 (#0)
> GET http://localhost/test.html HTTP/1.1
> Host: localhost
> User-Agent: curl/7.81.0
> Accept: */*
> Proxy-Connection: Keep-Alive
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 500 Internal Server Error
* no chunk, no close, no size. Assume close to signal end
<
[WinError 10061] No connection could be made because the target machine actively refused it* Closing connection 0
```

Multi-Threaded Web Server

We used multi-threading for our web server implementation. Using multi-threading the web server creates a new thread for each incoming request. This allows the server to handle multiple requests at the same time.

Impact on Performance

The server's throughput is significantly increased. This is because the server can handle many requests in parallel. This is particularly important for high-traffic scenarios. Clients also experience lower latency. The lower latency is because their requests are processed without waiting for previous requests to complete. The server can also make better use of multi-core processors. It does that by having threads running on different cores and performing tasks simultaneously.

Avoiding HOL Problem

We created a separate file to make changes to our server(web_server_HOL.py). The primary goal is to avoid the Head-of-Line (HOL) blocking problem.

Implementation

We implemented using “asyncio” for improved concurrency, scalability, and efficiency.

- **Asyncio Integration**: The server was refactored to use asyncio (asyncio library), adopting the async/await syntax for defining coroutines (`async def` functions).
- **Event-Driven Model**: Asyncio's event loop was employed to manage multiple I/O-bound tasks efficiently within a single thread. Which in turn will optimize resource utilization.
- **Non-Blocking I/O Operations**: Blocking I/O operations (`client_socket.recv(1024)`) in the original implementation were replaced with non-blocking equivalents (`await reader.read(1024)` in asyncio).
- **Concurrency and Scalability**: The server achieved higher concurrency and scalability. This is achieved by leveraging asyncio's non-blocking I/O and cooperative multitasking capabilities.