COMP2322 Computer Networking

Project: Multi-threaded HTTP Server

Author
Wang Yuqi
[redacted]

Lecturer
Dr. LOU Wei

BsC in Computer Science
Academic year: 2024/2025 Sem 2

Checklist

Grading Criteria	Files	Report Pages
Request/response message exchanges	`server.py` `utils.py`	P6-9
`GET` for text & image files	`server.py` `utils.py (L96-102)`	P7-9
`HEAD` command	`server.py (L236-243)`	P9-10
Six types of response code	`server.py (L50-243)`	P8
`Last-Modified` `If-Modified-Since`	`server.py (L170-189)` `utils.py (L106-184)`	P8, 10
`Connection` header field	`server.py (L118-122)`	P9
Persistent/Non-persistent Connections	`server.py (L50-88)`	P7, 9
Program testing & results	`tests/test_server.py` `tests/client.py`	P12-15, 21-27
Complete log file	`server.log`	P25
Source code quality	-	P4-5

Message to TA:

The above checklist shows that all grading criterias are met in this project. The corresponding source code and report pages are listed in the ***Files*** and ***Report Pages*** columns, respectively.

Additionally, the project code has been tested on:

- **Linux** - Ubuntu 22.04, 24.04
- **MacOS** - Sequoia 15.3.1

NOTE: Windows is NOT SUPPORTED!

Therefore, to the best of my knowledge, the code is fully compliant with all the requirements and runs flawlessly on major Unix-like systems. Windows is not supported. Should you have any concerns, please do not hesitate to contact me at: **tony-yuqi.wang@connect.polyu.hk**

Getting Started

Linux/MacOS

Running the server this is straightforward (see README for full list of arguments):

```
cd "project-root-directory" # Parent folder of code/ and report/
python3 code/main.py --host 127.0.0.1 --port 8080 --timeout 15 --
max-conn 100
```

Browset Testing

- Open a web browser and navigate to: `http://127.0.0.1:8080/`
- Then, the instruction for testing on the lander page should be self-explanatory

Running unit tests Keep server running, and start a new terminal session:

```
export TEST_SRVR_HOST="127.0.0.1"
export TEST_SRVR_PORT="8080"
export TEST_SRVR_TIMEOUT="15"
export TEST_SRVR_MAX_CONN="100"
python3 -m unittest discover -v code/tests
```

IMPORTANT NOTE: Ensure these environment variable values (`TEST_SRVR_HOST`, `TEST_SRVR_PORT`, etc.) **match exactly** the corresponding arguments (`--host`, `--port`, etc.) set when starting the server.

Windows (NOT RECOMMENDED)

The handling of connection termination in Windows appears to be vastly different from Unix-like systems. While the server is runnable on Windows, I hereby declare that it is **NOT GUARANTEED** to work flawlessly.

Running the server

```
cd "project-root-directory"
python code\main.py --host 127.0.0.1 --port 8080 --timeout 15 --max-
conn 100
```

Note: Expect to encounter unknown errors (`WinError 10054` and `10053`).

Running unit tests

Does not work on Windows.

Contents

Abstract

This report seeks to give a concise overview of the multi-threaded HTTP server's design and implementation, which I created in Python using low-level socket interfaces. With a small codebase (about 300 lines), its modularized design enables rich features. The server fully satisfies the project's requirements by supporting the HTTP/1.0 and HTTP/1.1 protocols, two distinct methods (`GET` and `HEAD`), six HTTP status codes (200, 304, 400, 403, 404, and 415), caching, and (non)persistent connections. Along with keeping a thorough request log for every client request, path validation and access control are also implemented for security. To demonstrate different response scenarios, a variety of static files are compiled together with an `index.html` for testing. To cover nuanced cases, a unit test script is also created.

Introduction

This project implements a multi-threaded HTTP server via low-level socket programming in Python. To demonstrate the **a)** comprehensiveness of the server functionality in meeting project requirements **b)** quality of the source code in following good practices, and **c)** robustness of the server in handling various scenarios, I hereby divide the report into four sections: **Design**, **Implementation**, **Testing**, and **Appendix**.

The **Design** section will first give a high-level overview of the code design, such as how different modules are designed and interact with each other. This aims to demonstrate the logical soundness and comprehensiveness of the functionalities. Next, the **Implementation** section will give a detailed walkthrough of the implementation of the server. This aims to demonstrate the quality of the source code in following good practices and error handling capacity. Finally, any testing methods are detailed in the **Testing** section, while the results are listed in the **Appendix**. This aims to ensure the flawless execution of the server.

I. Design

Our project code is divided into three main modules: `server.py`, `logger.py`, and `utils.py`, with `main.py` as the entry point linking them together, and the `utils.py` simply a collection of miscellaneous helper methods.

As shown in Figure 1, the server start by calling the `init_srv` function (`init_server` in Figure 1). This function will attempt to create and return a socket binded to an available

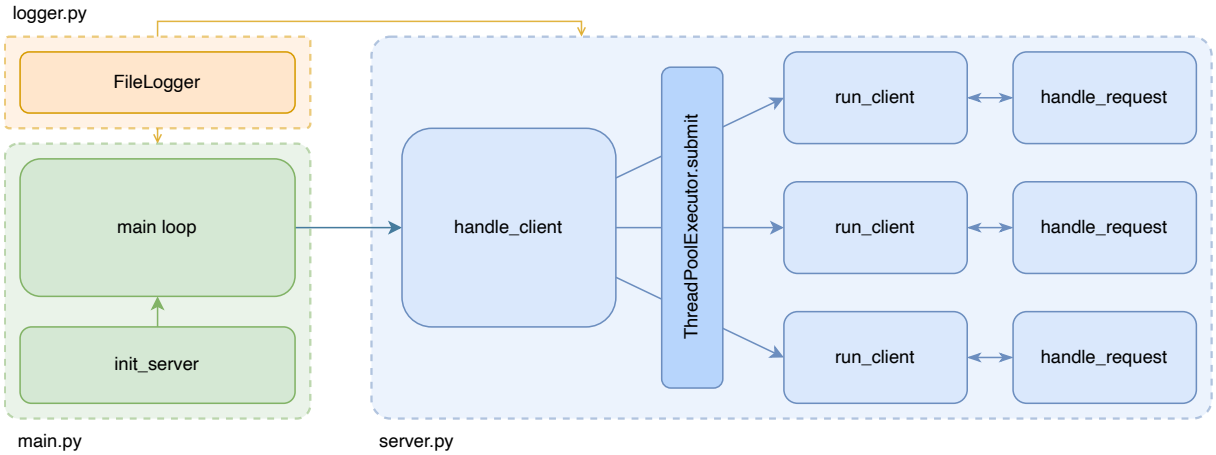


Figure 1: Architecture of the multi-threaded HTTP server

port. The main function will also create a `FileLogger` instance which is responsible for writing to the server log file for the rest of the program

From now on, the main loop function in `main.py` will be actively listening for incoming connections on the socket created by `init_srv`. When an incoming connection is detected, it is immediately passed to the `handle_cli` method (or `handle_client` in Figure 1). This method is responsible for creating and mapping these new connections to new threads, enabling multi-threaded execution. In Figure 1, this process is illustrated as the `handle_client` box diverging into numerous threads via `concurrent.futures.ThreadPoolExecutor` (also see snipet below).

```
def handle_cli(...):
    _POOL.submit(
      run_cli, c_sock, c_addr, logger,
      timeout, max_conn, cache_age)
```

Listing 1: Code snippet of the `handle_cli` method in `server.py`

As shown in Listing 1, the `handle_cli` create new threads by mapping the `run_cli` function (or `run_client` in Figure 1) to the new thread along with necessary arguments. Within each thread, the `run_cli` method is deeply integrated with a `handle_req` method (or `handle_request` in Figure 1). The former maintains a persistent connection (provided that the connection is HTTP/1.1 or HTTP/1.0 that explicitly requested with `keep-alive`). Whereas, the latter handles individual requests.

This whole process of **a)** the loop actively listening for new connections, **b)** delegating them to individual threads, where there is **c)** a `handle_req` method that handles individual requests nested within an outer persistent connection loop in `handle_cli`, eventually exists gracefully upon user `KeyboardInterrupt`.

II. Implementation

Following the overview given in the **Design** section, this section will provide a more detailed walkthrough of the code implementation of the HTTP server.

main.py

As discussed earlier, this file is the entry point of the server. It will first parse any command line arguments via Python's built-in `argparse`, and then call the `init_srv` function to create the server socket. By default, the server will listen on `127.0.0.1:80`, with the maximum number of requests set to 1000. Users can easily change these parameters as shown in Listing 2. See the full list of supported arguments in **Appendix**.

```
python main.py --host 127.0.0.1 --port 8080 --timeout 15 --max-conn 100
```

Listing 2: Example custom CLI arguments

Note that, in case the default port provided in the `--port` argument is unavailable, the `init_srv` function will scan through all ports between 8080 and 65535, and bind to the first available port.

Then, in the `main` function loop, the server will accept new connections on the socket created by `init_srv`, and pass them to the `handle_cli` method. Along with the client's socket and address, a `timeout` and `max_conn` argument are also passed to the `handle_cli` method, which tells the server how long to wait before resetting a persistent connection, and the maximum number of connections each persistent connection is allowed to handle (i.e., the `timeout=xxx, max=xxx` in `Keep-Alive` header).

```
try:
    while True:
        c_sock: socket.socket
        c_addr: tuple[str, int]
        c_sock, c_addr = srv_sock.accept()
        handle_cli(
          c_sock, c_addr, logger,
          args.timeout, args.max_conn)
except OSError as e: log_err(f"[!] Server error - {e}")
except KeyboardInterrupt: log_info("Shutting down server...")
finally:
    logger.close()
    srv_sock.close()
    exit(0)
```

Listing 3: Main listening loop in `main.py`

server.py

The `server.py` file contains the core logic of the multi-threaded HTTP server (the **blue boxes** in Figure 1). It is responsible for **a)** delegating them to individual threads, **b)** maintaining persistent connections, and **c)** handling individual requests within each connection thread (receive, parse, and send):

- **`handle_cli`**: corresponds to **a)** - delegating new connections to new threads
- **`run_cli`**: corresponds to **b)** - handles persistent connections (if needed)
- **`handle_req`**: corresponds to **c)** - handles individual requests within each connection

`handle_cli(c_sock, c_addr, logger, timeout, max_conn)`

This method is the simplest among the three, as its only job is just to create a new thread and map the `run_cli` method to it. Recall that `timeout` and `max_conn` are arguments specified in the CLI that determines the server's behavior in handling persistent connections; the `logger` is a `FileLogger` instances (more detail in page 11).

`run_cli(c_sock, c_addr, logger, timeout, max_conn)`

This method oversees the entire life-cycle of a particular client-server connection. It will first call the `recv_req` method to receive the HTTP request from the client. The `recv_req` method returns a tuple of the request data `req_data` and a boolean `recv_ok` that indicates whether error occurred. At this point, there are three different scenarios:

1. **`recv_ok = True, req_data = "..."`**: indicates an normal data reception. The received data will thus be passed to `handle_req` for further processing and respond.
2. **`recv_ok = True, req_data = ""`**: since no data is received while no error occurred, it indicates that the client has closed the connection gracefully.
3. **`recv_ok = False`**: indicates an error occurred during data reception. The server will thus send a `400 Bad Request` response to the client and close the connection.
4. **`socket.timeout`**: indicates that the client failed to send any data within the given timeout. The server will close gracefully.

`handle_req(c_sock, c_addr, logger, req_data, timeout, max_conn)`

This method is the most intricate among all. It handles the entire life-cycle of a particular request of a connection thread, from parsing the raw HTTP data, sanitizing content, checking headers, and crafting and sending the final response.

The method starts by parsing the raw HTTP data into a dictionary, where each key-value pair corresponds to a component of the HTTP header. The server assumes no content body in `GET` and `HEAD` requests, and will proceed immediately after the header parsing.

Now that the HTTP header's grammatical correctness is guaranteed, a series of checks are performed on the parsed HTTP request headers:

1. **Version**: only HTTP/1.0 and HTTP/1.1 are supported.
2. **Method**: only `GET` and `HEAD` are supported.
3. **Path Sanity**: the path must be a valid HTTP path (invalid examples: `//`, `../`, `./`).
4. **Path Access**: must be in `/resources/`, the only allowed directory for access.
5. **File Existence**: the path must point to a file that exists on the server.
6. **File Type**: the requested file must be a text file or image file.
7. **If-Modified-Since**: if exists, the date format in `if-modified-since` must be valid.
8. **File Modification**: if `if-modified-since` exists and is valid, fetch the current file modification time and compare it with the date provided in `if-modified-since`.

	Code	Console Log Message
Version	400	Unsupported version: {version}
Method	400	Unsupported method: {method}
Path Sanity	400	Invalid path: {path}
Path Access	403	Access denied: {path}
File Existence	404	File not found: {path}
File Type	415	Unsupported content type: {extension}
If-Modified-Since	400	Invalid if-modified-since: {date string}
File Modification	304	File not modified: {path}

Table 1: Response codes and console log messages for different scenarios

Table 1 above shows the response codes that will be sent under each of the above scenarios, along with the log messages that will be printed to console (not `server.log`). If all the above checks are passed, the server will proceed with a **200 OK** response.

Algorithm 1: Keep-Alive Check

```
If version is HTTP/1.1 then
    keep_alive = true
    If Connection in headers and Connection is close then
        keep_alive = false
If version is HTTP/1.0 then
    keep_alive = false
    If Connection in headers and Connection is keep-alive then
        keep_alive = true
```

In addition, by design the HTTP server also return `Keep-Alive` and `Connection` headers in their responses. The logic for determining their header parameter values, is given in Algorithm 1.

Intuitively, HTTP/1.1 should be kept alive by default, unless received explicit `Connection: close` header. In contrast, HTTP/1.0 should be closed by default, unless received explicit a `Connection: keep-alive` header. The only exception is if the HTTP version is not supported, in which case the server will respond with a `400 Bad Request` and close the connection. This is because `Keep-Alive` and `Connection` headers (see **Appendix G** for more details).

To actually implement sending these response header with the appropriate response code, a `sender` function is created.

Algorithm 2: Sender Function

```
1  Function sender(code, msg, path='INVALID', headers={}, content=empty)
2      If keep_alive = true then
3          Set headers["Connection"] ⟵ "keep-alive"
4          Set headers["Keep-Alive"] ⟵ "timeout=" + timeout + ", max=" + max_conn
5      Else
6          Set headers["Connection"] ⟵ "close"
7      Call send(client_socket, build(status_code, headers, content, method))
8      Log msg to console
9      Log to log file ⟵ (addr, port, path, code)
10     Return keep_alive
```

Algorithm 2 is a helper function called during each HTTP response. It encapsulates the logic for: **a)** updating HTTP response headers based on the `keep_alive` flag (see Algorithm 1), **b)** sending the updated HTTP response, and **c)** logging the response to console and server log file. These are procedures that needs to be performed for every HTTP response, and thus this encapsulation is a good example of code quality.

```
build(status, hdrs, content, method)
```

The method takes the status code, headers, content, and method as parameters, and returns the full HTTP response encoded as bytes.

As detailed in Algorithm 3, the HTTP body is first built: if the method is `HEAD`, the content body is empty. If the status code is not 200 or 304 (i.e., an error occurred), instead of concatenating an empty content body, a custom HTML error page is gener-

Algorithm 3: Build Function

```
Function build(status, hdrs, content, method)
    If method is not HEAD then
        If status_code is 200 then
            Set response_body ← content
        Elif status_code is 304 then
            Set response_body ← empty
        Else
            Generate error_html using status_code, status_message
            Set response_body ← error_html
    Else Set response_body ← empty
    hdrs.Content-Length ← length of response_body
    hdrs.Content-Type ← "text/html"
    Lookup status_message based on status_code
    Initialize header_string ← ""
    For each key, value in hdrs do
        header_string ← header_string + key: value + "\r\n"
    response_headers ← "HTTP/1.1" + status_code + status_message + "\r\n"
    response_headers ← response_headers + header_string + "\r\n"
    Encode response_headers to Bytes
    Encode response_body to Bytes
    Return response_headers + response_body
```

ated using the `status_code` and `status_message`. Then, the HTTP header is built by concatenating the status code, status message, and headers key-value pair with CRLF delimiter in-between.

logger.py

The `logger.py` file contains the implementation of the `FileLogger` class, which is responsible for logging the server's activity to a specific log file.

The log message follows the format:

```
[timestamp] Server Started
[timestamp] host:port - filepath - response_code
[timestamp] host:port - filepath - response_code
[timestamp] host:port - filepath - response_code
[timestamp] Server Closed
```

As shown in Listing 4, each line of log message contains timestamp, host, port, filepath, and response code. If the request was invalid (e.g., 400 Bad Request), then the filepath is replaced with "BAD-REQUEST". If the server is closed or just opened, then the log message is simply a timestamp + "Server Started" or "Server Closed". The timestamp is formatted as `YYYY-MM-DD HH:MM:SS`. Below is an example log generated by the server:

```
[2025-04-18 02:30:35] Server Started
[2025-04-18 02:30:38] 127.0.0.1:58573 - index.html - 304
[2025-04-18 02:30:38] 127.0.0.1:58573 - style.css - 200
[2025-04-18 02:32:18] 127.0.0.1:58608 - secrets/secret.jpeg - 403
[2025-04-18 02:32:20] 127.0.0.1:58608 - secrets/public.txt - 403
[2025-04-18 02:32:26] 127.0.0.1:58610 - secrets/private.txt - 403
[2025-04-18 02:32:33] 127.0.0.1:58612 - resources/nonexistent.txt - 404
[2025-04-18 02:33:17] Server Closed
```

In addition to the `FileLogger` class, the `logger.py` file also contains a series of helper functions for logging to the console. These functions are:

`log_info(msg)`	logs a **blue** message to the console
`log_warn(msg)`	logs a **orange** message to the console
`log_err(msg)`	logs a **red** message to the console
`log_succ(msg)`	logs a **green** message to the console
`log_debug(msg)`	logs a **magenta** message to the console
`print_args(args)`	prints the server arguments to the console in a formatted list

utils.py

The `utils.py` file contains a series of utility functions for the HTTP server.

check_bind	checks if the host and port are valid and available
recv_req	`recv()` wrapper with timeout and connection closed handling
valid_path	check if a provided path has valid format
get_path	parse quoted URL path into a processable path
parse_http	parse HTTP request data into a dictionary
get_content	read the content of a file into bytes
get_mod_time	get file modification time in HTTP-Date format (**RFC 9110**)
valid_mod_time	check if a provided date string is valid HTTP-Date format
cmp_mod_time	compare two HTTP-Date strings
allowed_path	check if a provided path is allowed to be accessed

III. Testing

To thoroughly test the functionality of the HTTP server, two approaches were taken:

1. **Unit Testing**: This allows for customized request headers that tests corner cases not easily achievable through browser test.
2. **Browser Testing**: This is a relatively simple test examines static file serving, correctness of status codes, and persistent connections.

Unit Testing

While **Browser Testing** covers basic functionalities, it cannot test corner cases like invalid HTTP headers (as browsers always send valid ones). Therefore, Python's `unittest` library is used to cover these nuances. Table 2 details the list of unit tests that were created to achieve this purpose:

Test Name	Objective	Expectation
`test_get_root`	Basic http root path access	200 OK + HTML
`test_get_txt`	Basic http static text file serving	200 OK + text.txt
`test_head_txt`	Static text file serving w/ `HEAD`	200 OK + empty
`test_404`	Access non-existent file	404 Not Found
`test_403`	Access forbidden file	403 Forbidden
`test_415`	Access unsupported media type	415 Unsupported
`test_400_mth`	Invalid method	400 Bad Request
`test_400_ver`	Invalid HTTP version	400 Bad Request
`test_400_path`	Invalid path	400 Bad Request
`test_304`	Repeat same request twice	200 OK, then 304
`test_old_date`	Send old `If-Modified-Since` date	200 OK
`test_new_date`	Send new `If-Modified-Since` date	304 Not Modified
`test_bad_date`	Send invalid `If-Modified-Since` date	400 Bad Request
`test_alive_http11`	Send 2 HTTP/1.1 requests	200 OK, 200 OK
`test_close_http11`	Send 2 HTTP/1.1 requests, w/ `close`	200 OK, Closed
`test_close_http10`	Send 2 HTTP/1.0 requests	200 OK, then fail
`test_alive_http10`	Send 2 HTTP/1.0 requests, w/ `keep-alive`	200 OK, 200 OK
`test_multi_thread`	Check if multi-threading is working	Pearson $\lvert r \rvert < 0.8$
`test_stress`	Flood server with rapid requests	200 OK $\times N$
`test_timeout`	Check if timeouts after a while	Closed

Table 2: Comprehensive list of `unittest` cases

While most test cases are self-explanatory, the `test_multi_thread()` and `test_stress` cases are particularly worth mentioning due to their relatively complex nature.

`test_multi_thread()`

The `test_multi_thread()` case is designed to test whether the server truly supports handling multiple requests concurrently.

Naively, one might simply test this by launching concurrent threads to send numerous requests simultaneously. But this approach can be misleading, as the responses returned by the server are identical irrespective of whether the requests are handled concurrently or sequentially by the HTTP server.

Therefore, I took a slightly modified approach. First N threads are launched concurrently via `threading.Thread()`. Each thread will request a large file (`/resources/big.txt`, 6.5MB) from the server. The time taken by each thread to complete the request is recorded. Then, a Pearson correlation coefficient r is computed as follows:

$$r = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2 \sum_{i=1}^{N}(y_i - \bar{y})^2}}$$

Where,

- x_i are the indices of the threads
- y_i are the time taken by each thread to complete the request
- N is the number of threads (concurrent requests)
- $\bar{x}$ and $\bar{y}$ are the means of x_i and y_i respectively

The final result r indicates the level of correlation between the indices of the threads and the time taken to complete the request.

- If the server is multi-threaded, then the time taken should be roughly the same with some random fluctuations. Thus $r \to 0.0$.
- If the server is sequential, then the time taken should increase linearly with the index of the thread, because each subsequent request has to wait for the previous one to complete. In this case, $r \to \pm 1.0$ (delay strongly depends on the order of requests).

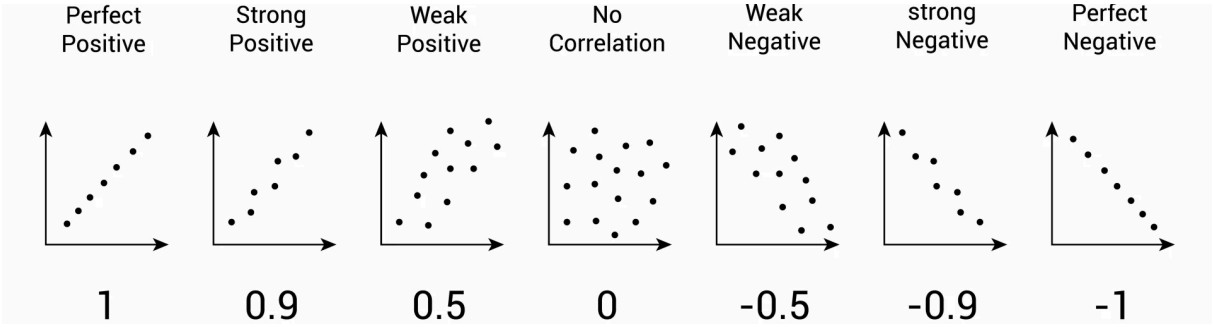


Figure 2: Visual illustration of various Pearson correlation values

Therefore, to determine whether the server is multi-threaded, we can simply check if $|r| > 0.8$ (set to higher values to avoid false positives). For more detail, see Algorithm 3:

Algorithm 3: Test Multi-Thread

Function test_multi_thread():
    Launch N threads
    Record delay y_i for each thread i
    For i from 1 **to** N: $x_i \leftarrow i$
    $r = \left(\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})\right) / \sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2 \sum_{i=1}^{N}(y_i - \bar{y})^2}$
    If $|r| > 0.8$ **then** Fail Test
    Else Pass Test

Algorithm 3: Implementation of `test_multi_thread()`

`test_stress()`

The `test_stress()` case assesses the server's ability to handle a high volume of rapid requests that flood its socket receive buffer. A robust server must correctly process numerous queued requests from this buffer without errors or unexpected behavior. Naive implementations often struggle under such conditions. Common failure are:

1. **Reading chunks until `\r\n\r\n`** (Naive Code 1): If multiple requests accumulate in the buffer, a `recv()` call might inadvertently consume data belonging to a subsequent request. This corrupts the later request's HTTP structure, leading to parsing errors (e.g., `400 Bad Request`) and premature connection termination.

```
while b'\r\n\r\n' not in request_data:
  chunk = client socket.recv(4096)
  request_data += chunk
```

Naive Code 1: Receiving byte chunks until `\r\n\r\n` is encountered

2. **Reading the entire buffer** (Naive Code 2): This approach might incorrectly merge multiple distinct requests into one. Alternatively, if logic is added to split the buffered data into individual requests, the server could stall or crash due to memory exhaustion if the incoming request rate surpasses its processing capacity.

```
while True:
  chunk = client socket.recv(4096)
  request data += chunk
  if len(chunk) < 4096:
    break
```

Naive Code 2: Receiving byte chunks until the buffer is empty

To implement such a test, instead of sending a single request and waiting for it to complete via `recv()`, N requests are sent at once without blocking. Since the server will respond with a large number of responses, high socket buffer sizes (256MB) are set via:

```
sock.setsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF, 256 * 1024 * 10)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, 256 * 1024 * 10)
```

Finally, the client verifies the test's correctness by ensuring it receives exactly N instances of `200 OK` responses from the server.

Browser Testing

To test basic server functionalities, such as static file serving, returning appropriate status codes when the requested files are invalid, and maintaining persistent connection and timeouts, a `index.html` is created to ease this process.

When the server root path is accessed through the browser, the `server.py` automatically redirect the request to `/index.html`. As shown in Figure 4, this index page provides an elegant looking interface for testing different static files paths. Clicking each link will attempt to access the corresponding file paths.

For example, clicking on "`secrets/secret.txt`" will attempt to access `http://localhost:8080/secrets/secret.txt`. However, since only the `/resources/` direc-

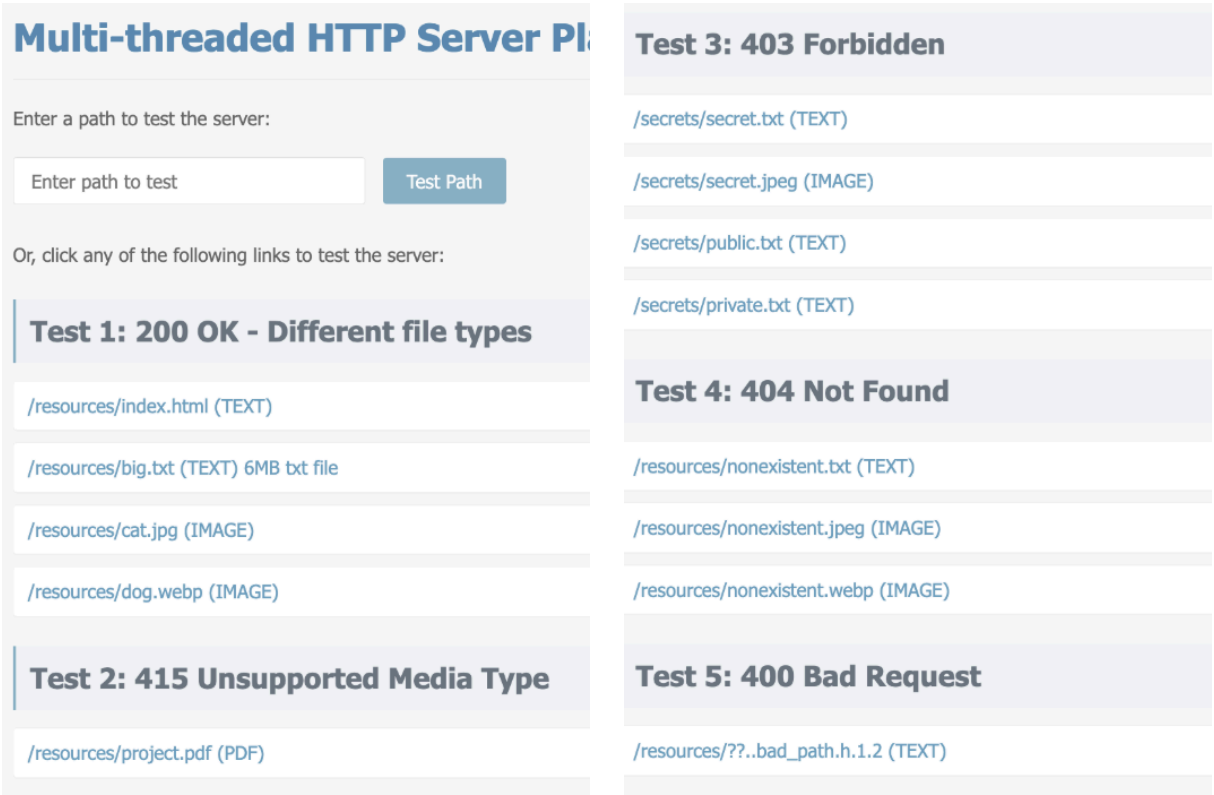


Figure 4: The `index.html` file

tory is allowed to be accessed, the request will fail the **Path Access** check (see Table 1 for more details), hence resulting in a **403 Forbidden** error, as shown in Figure 5. Recall that the error page is defined in the `build` function (see page 11).

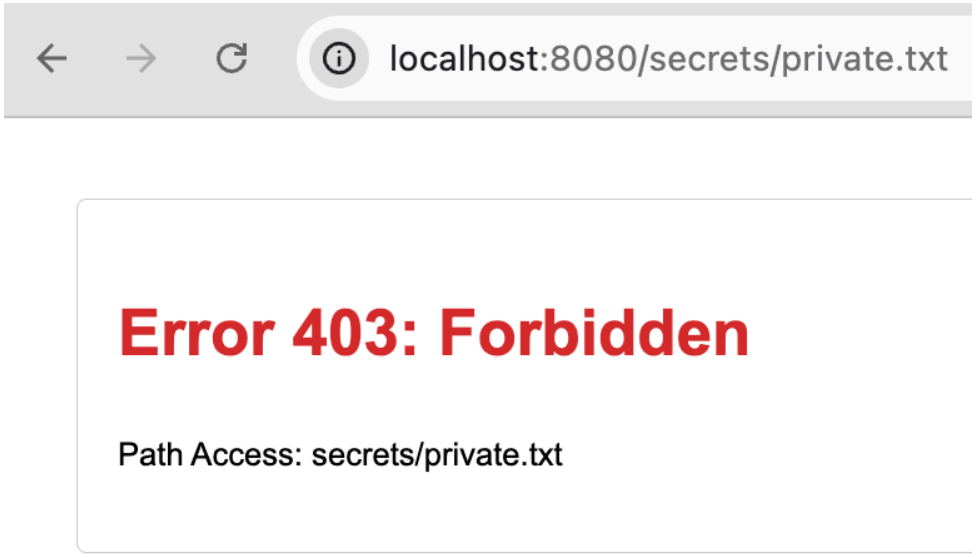


Figure 5: Error page when accessing `secrets/secret.txt`

This setup also allows us to test the server's ability to handle persistent connections. One can see from the server console output that the server will not close the connection after sending the response, until timeouts or client-initiated connection termination.

Test Results

The full test results can be found in the **Appendix** section. This section focuses only on the testing procedure and showcasing its comprehensiveness.

IV. Assumptions

A. HTTP Versions

The implemented multi-threaded HTTP server supports **only HTTP/1.0 and HTTP/1.1**. Here, we first justify such version choices; then, we justify the implementation strategy for responding in HTTP/1.1 irrespective of the client's advertised version.

A.1 HTTP/1.0 and HTTP/1.1

The choice of HTTP protocol version for this project is dictated strictly by the project specifications and the technical requirements outlined by the relevant RFC standards.

According to the project description, we are "required to use basic socket programming classes to build the Web server from scratch." The assignment also requires plain-text request/response handling, the use of headers such as `Last-Modified` and `If-Modified-Since`, support for specific HTTP methods (GET, HEAD), explicit connection management via the `Connection` header, and the accurate delivery of common HTTP status codes. All of these features are the core to HTTP/1.0 and 1.1 as defined in both **RFC 1945** (HTTP/1.0) and **RFC 2616** (HTTP/1.1) (later updated to **RFC 9110**). Hence, HTTP/1.0 and 1.1 align exactly with the project objectives provided.

In sharp contrast, HTTP/2.0 and higher (defined by **RFC9113** for HTTP/2) strictly diverge and fundamentally depart from HTTP/1. HTTP/2 is a binary-framing protocol, utilizing multiplexing, flow control, header compression, and other advanced non-textual features.

- RFC 9113, Section 4.1: Messages as binary frames with a complex 9-byte headers.
- RFC 9113, Section 4.3: Field section compression and decompression.
- RFC 9113, Section 5: Stream, multiplexing, and flow control.

Clearly, implementing binary-level framing, streaming, and multiplexing would violate explict statements of the project description with require no use of HTTP-level abstraction layers. Without high-level libraries drastically exceeds the project's workloads, and is thus unfeasible. Thus, HTTP/1.0 and HTTP/1.1 best fit within the constraints and mandates of the assignment, as they are the only versions accessible and feasible with pure socket-level interfaces.

A.2 Responding in HTTP/1.1

We choose to **always** respond in HTTP/1.1, regardless of the client's advertised version. This is because HTTP/1.1 and 1.0 does not obsolete each other, and are

mutually compatible. Sending higher version response is also recommended in **RFC 9110**, which writes:

> *All three major versions of HTTP rely on the semantics defined by this document. They have not obsoleted each other because each one has specific benefits and limitations depending on the context of use. Implementations are expected to choose the most appropriate transport and messaging syntax for their particular context.*
>
> *This revision of HTTP separates the definition of semantics (this document) and caching ([CACHING]) from the current HTTP/1.1 messaging syntax ([HTTP/1.1]) to allow each major protocol version to progress independently while referring to the same core semantics.*
>
> — RFC 9110, Section 1.2 History and Evolution

Followed by a recommended strategy also according to **RFC 9110**:

> *A server SHOULD send a response version equal to the highest version to which the server is conformant that has a major version less than or equal to the one received in the request. A server MUST NOT send a version to which it is not conformant. A server can send a 505 (HTTP Version Not Supported) response if it wishes, for any reason, to refuse service of the client's major protocol version.*
>
> — RFC 9110, Section 6.2 Control Data

In simpler terms, when the server is generating a response, it should pick the **highest HTTP version it supports** within the same major version number as what the client sent in the request (or lower). Therefore, my implementation reasonably choose to respond in HTTP/1.1 irrespective of whether the client is HTTP/1.0 or HTTP/1.1.

B. Parser

As described in the **Implementation** section, we intentionally designed the HTTP parser to shutdown connections when requests blatantly violate the HTTP grammar. This design choice is closely aligned with the **RFC 9112** specification, which writes:

> *When a server listening only for HTTP request messages, or processing what appears from the start-line to be an HTTP request message, receives a sequence of octets that does not match the HTTP-message grammar aside from the robustness exceptions listed above, the server SHOULD respond with a 400 (Bad Request) response and close the connection.*
>
> — RFC 9112, Section 2.2 Message Parsing

C. Timeouts

The design choice of graceful closing upon timeout, instead of returning an HTTP 408 Request Timeout, too closely aligns with the **RFC 9112** specification, which writes:

> *A client or server that wishes to time out SHOULD issue a graceful close on the connection.*
>
> — RFC 9112, Section 9.5 Failures and Timeouts

D. Content Body

By design choice, the HTTP server will assume no content body in `GET` and `HEAD` requests. Any content trailing the HTTP request headers will be treated as the next request. This will obviously fail the parser's HTTP grammar check in the next round, hence causing 400 Bad Request and shutting down the connection. This design choice is, again, backed by two quotes from **RFC 9110** specification:

> *Although request message framing is independent of the method used, content received in a GET request has no generally defined semantics, cannot alter the meaning or target of the request, and might lead some implementations to reject the request and close the connection because of its potential as a request smuggling attack (Section 11.2 of [HTTP/1.1]).*
>
> — RFC 9110, Section 9.3.1 GET

followed by,

> *A client SHOULD NOT generate content in a HEAD request unless it is made directly to an origin server that has previously indicated, in or out of band, that such a request has a purpose and will be adequately supported. An origin server SHOULD NOT rely on private agreements to receive content, since participants in HTTP communication are often unaware of intermediaries along the request chain.*
>
> — RFC 9110, Section 9.3.2 HEAD

E. Access Privileges

For demonstration purposes (specifically to showcase 403 Forbidden), the server will only allow access to the `/resources/` directory and its subdirectories. This prevents unauthorized access to arbitrary files on the server. As for testing, a folder named `/secrets/` is created adjacent to `/resources/` to showcase access denial.

F. Supported File Types

In the project requirements, it is stated that the `GET` command should support fetching *both text files and image files* from the server. Since, the supported file extensions are not specified explicitly, I assumed a broader interpretation of "text files" and "image files", and thus supported the following file extensions:

```
'html': 'text/html',          'ico': 'image/x-icon',
'css': 'text/css',            'svg': 'image/svg+xml',
'js': 'text/javascript',      'webp': 'image/webp',
'png': 'image/png',           'txt': 'text/plain',
'jpg': 'image/jpeg',          'json': 'application/json',
'jpeg': 'image/jpeg',         'xml': 'application/xml',
'gif': 'image/gif',
```

G. `Connection` and `Keep-Alive` Header

The `Keep-Alive` header is not required in standard implementation. For the client, sending `Keep-Alive` headers can be highly error-prone, as described in **RFC 2068**:

> *The problem was that some existing 1.0 clients may be sending Keep-Alive to a proxy server that doesn't understand Connection, which would then erroneously forward it to the next inbound server, which would establish the Keep-Alive connection and result in a hung HTTP/1.0 proxy waiting for the close on the response. The result is that HTTP/1.0 clients must be prevented from using Keep-Alive when talking to proxies. However, talking to proxies is the most important use of persistent connections, so that prohibition is clearly unacceptable.*
>
> — RFC 2068, Section 19.7.1

Whereas, for the server, while this is not a problem, there are still some considerations against using `Keep-Alive` header. For example, HTTP/1.1 does not define any official `Keep-Alive` parameters; `Keep-Alive` header is even entirely ignored in HTTP/2.0 and later version. For this, **RFC 2068** and **MDN docs** writes:

> *The Keep-Alive header itself is optional, and is used only if a parameter is being sent. HTTP/1.1 does not define any parameters.*
>
> — RFC 2068, Section 19.7.1

> *Warning: Connection-specific header fields such as Connection and Keep-Alive are prohibited in HTTP/2 and HTTP/3. Chrome and Firefox ignore them in HTTP/2 responses, but Safari conforms to the HTTP/2 specification requirements and does not load any response that contains them.*

— MDN Docs, `Keep-Alive`

However, since we are implementing a simple HTTP/1.0 and HTTP/1.1 server, these are not direct prohibitations. Instead, using these headers will allow for backward compatibility with older HTTP/1.0 clients' implementations. As MDN docs further writes:

Some clients and servers might wish to be compatible with previous approaches to persistent connections, and can do this with a Connection: keep-alive request header. Additional parameters for the connection can be requested with the Keep-Alive header.

— MDN Docs, `Keep-Alive`

Therefore, after thorough consideration, it is decided that the this particular HTTP implementation is should include `Connection` and `Keep-Alive` header for 200 OK responses.

H. `Cache-Control` Header

Obviously, `200 OK` responses should include this header. It can also be inferred from the RFC standard that the `Cache-Control` header is incorporated with `304 Not Modified` responses. Hence, my implementation will include this header for both `200 OK` and `304 Not Modified` responses.

When a cache receives a 304 (Not Modified) response, it needs to identify stored responses that are suitable for updating with the new information provided, and then do so. […] For each stored response identified, the cache MUST update its header fields with the header fields provided in the 304 (Not Modified) response, as per Section 3.2.

— RFC 9111, Section 4.3.4

(See Next Page)

V. Appendix

A. Unit Testing Results

```
(base) ➜  code git:(main) ✗ export TEST_SRVR_HOST=127.0.0.1
export TEST_SRVR_PORT=8081
python3 -m unittest discover -v tests
test_304 (test_server.TestHTTPServer.test_304) ... ok
test_400_mth (test_server.TestHTTPServer.test_400_mth) ... ok
test_400_path (test_server.TestHTTPServer.test_400_path) ... ok
test_400_ver (test_server.TestHTTPServer.test_400_ver) ... ok
test_403 (test_server.TestHTTPServer.test_403) ... ok
test_404 (test_server.TestHTTPServer.test_404) ... ok
test_415 (test_server.TestHTTPServer.test_415) ... ok
test_alive_http10 (test_server.TestHTTPServer.test_alive_http10) ... ok
test_alive_http11 (test_server.TestHTTPServer.test_alive_http11) ... ok
test_bad_date (test_server.TestHTTPServer.test_bad_date) ... ok
test_close_http10 (test_server.TestHTTPServer.test_close_http10) ... ok
test_close_http11 (test_server.TestHTTPServer.test_close_http11) ... ok
test_get_root (test_server.TestHTTPServer.test_get_root) ... ok
test_get_txt (test_server.TestHTTPServer.test_get_txt) ... ok
test_head_txt (test_server.TestHTTPServer.test_head_txt) ... ok
test_multi_thread (test_server.TestHTTPServer.test_multi_thread) ... Pearson Correlation = 0.109
ok
test_new_date (test_server.TestHTTPServer.test_new_date) ... ok
test_old_date (test_server.TestHTTPServer.test_old_date) ... ok
test_timeout (test_server.TestHTTPServer.test_timeout) ... ok

----------------------------------------------------------------------
Ran 19 tests in 21.810s
```

Figure 6: Unit testing results

B. Multi-thread Testing Results

To robustly show that the server is multi-threaded and concurrent, the `/tests/test_thread_repeat.py` is created to run the `test_multi_thread` case as shown in

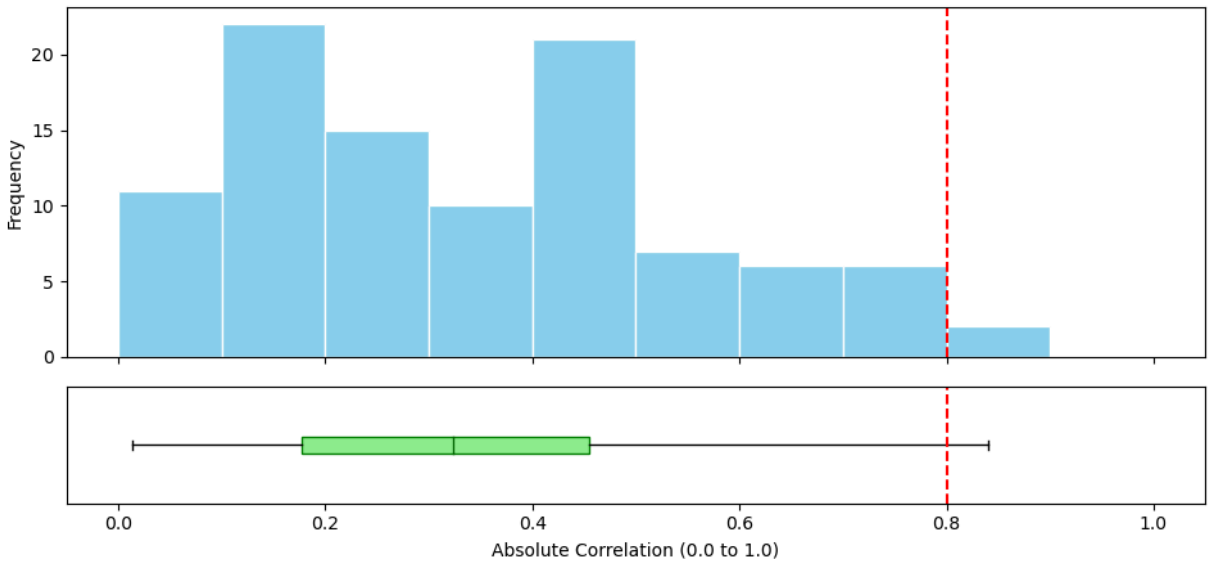


Figure 7: Multi-thread testing results

Table 2 for 100 iterations. Then, their absolute Pearson correlation coefficients $|r|$ is plotted as a histogram and box plot. As shown in Figure 7, the majority of $|r|$ are less than 0.5. Only 2 iterations have $|r| > 0.8$, which translates to a 2% false positive rate. Therefore, it can be confidently conclude that the server is multi-threaded and concurrent.

C. Browser Testing Results

```
code
├── resources
│   ├── big.txt
│   ├── cat.jpg
│   ├── dog.webp
│   ├── project.pdf
│   ├── index.html
│   └── style.css
└── secrets
    ├── credential.txt
    ├── private.txt
    ├── public.txt
    └── secret.jpeg
```

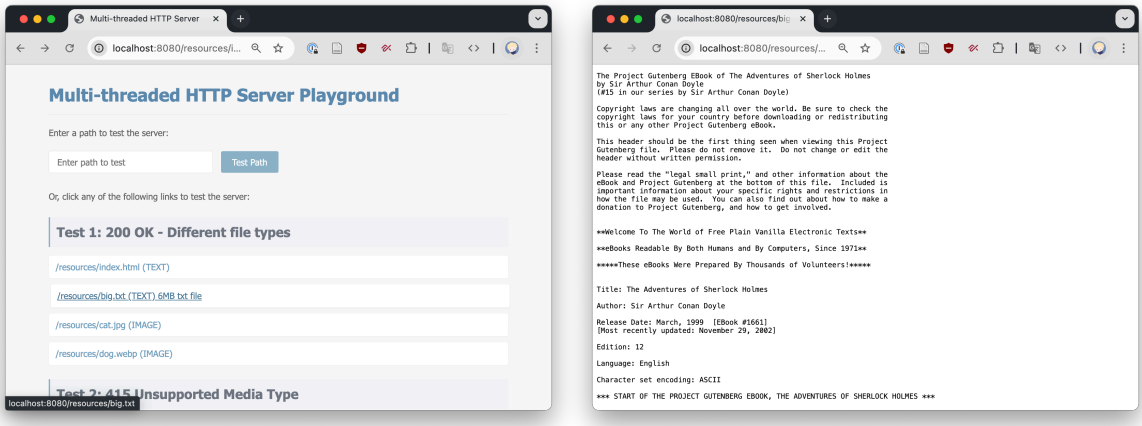
Expected Results

`/resources/big.txt`	200 OK
`/resources/cat.jpg`	200 OK
`/resources/dog.webp`	200 OK
`/resources/project.pdf`	415 Unsupported
`/resources/index.html`	200 OK
`/resources/style.css`	200 OK
`/secrets/credential.txt`	403 Forbidden
`/secrets/private.txt`	403 Forbidden
`/secrets/public.txt`	403 Forbidden
`/secrets/secret.jpeg`	403 Forbidden

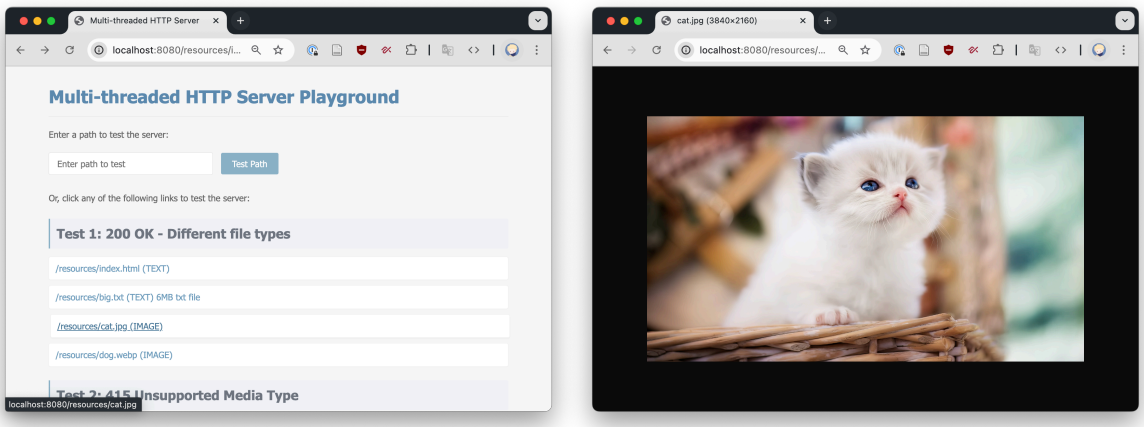
Listing 6: Directory structure and exepcted status codes

The directory of the server's resources and their expected status code is illustrated in Listing 6. Below we show screenshots of the browser testing results when accessing different files via the `index.html` page described in Figure 4 (recall **Testing** section).

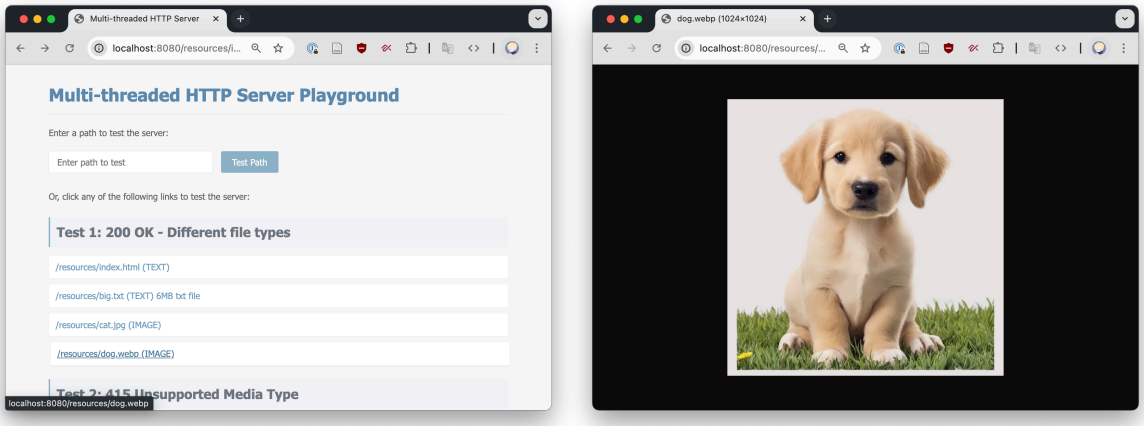
C1. Test 1 - 200 OK



Screenshots 1: Browser testing results for `/resources/big.txt`

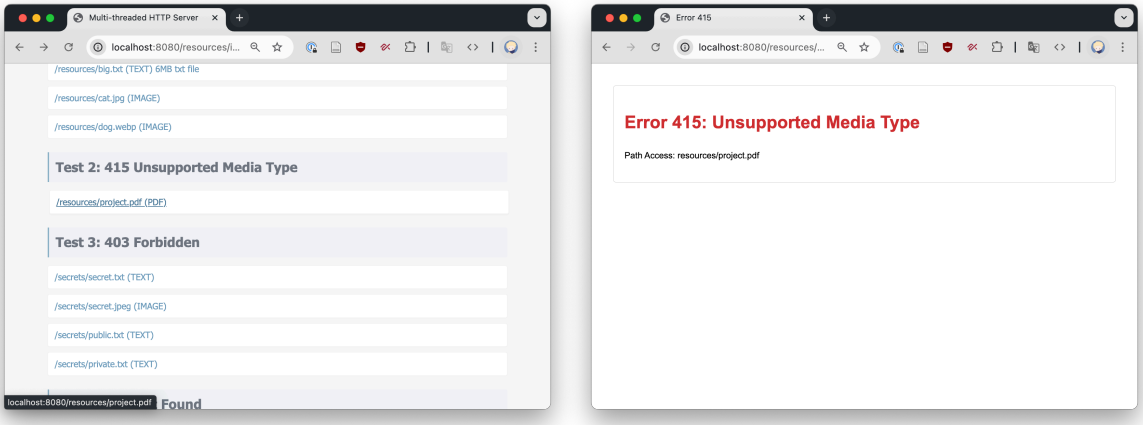


Screenshots 2: Browser testing results for `/resources/cat.jpg`



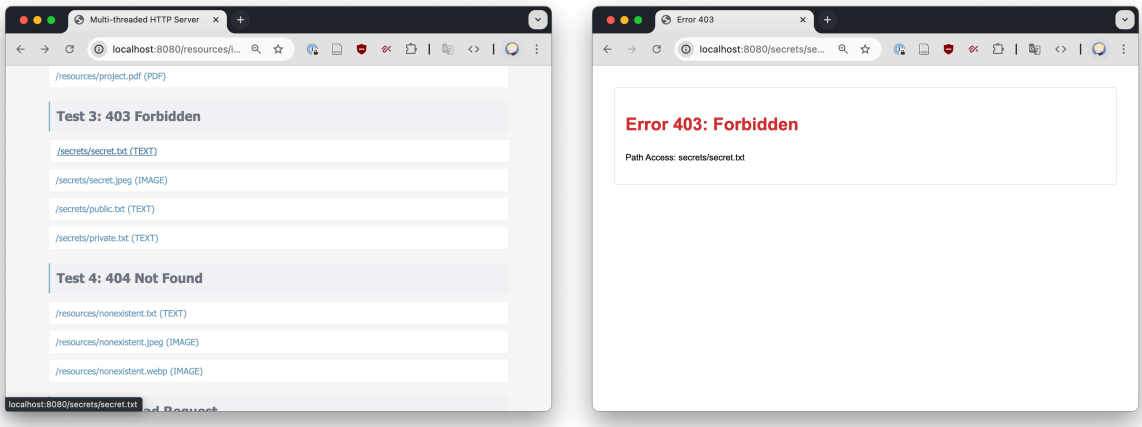
Screenshots 3: Browser testing results for `/resources/dog.webp`

C2. Test 2 - 415 Unsupported

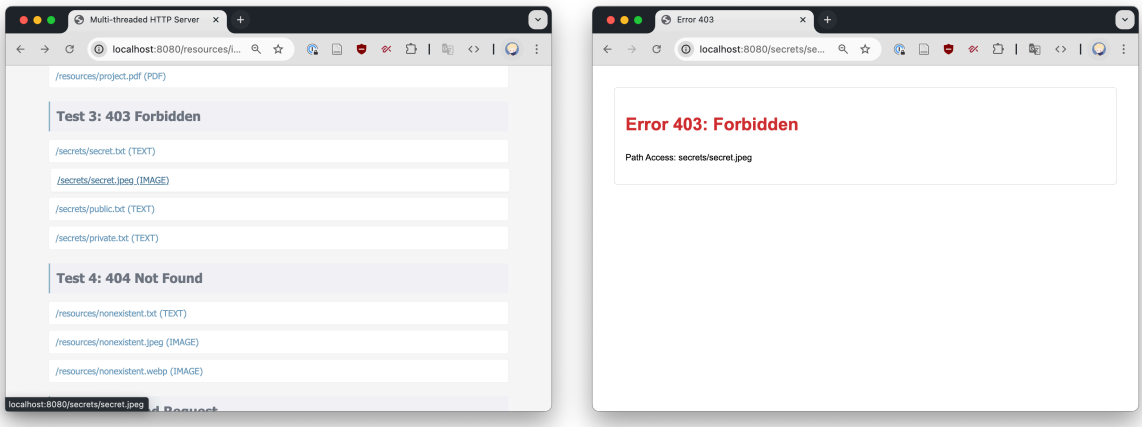


Screenshots 4: Browser testing results for `/resources/project.pdf`

C3. Test 3 - 403 Forbidden

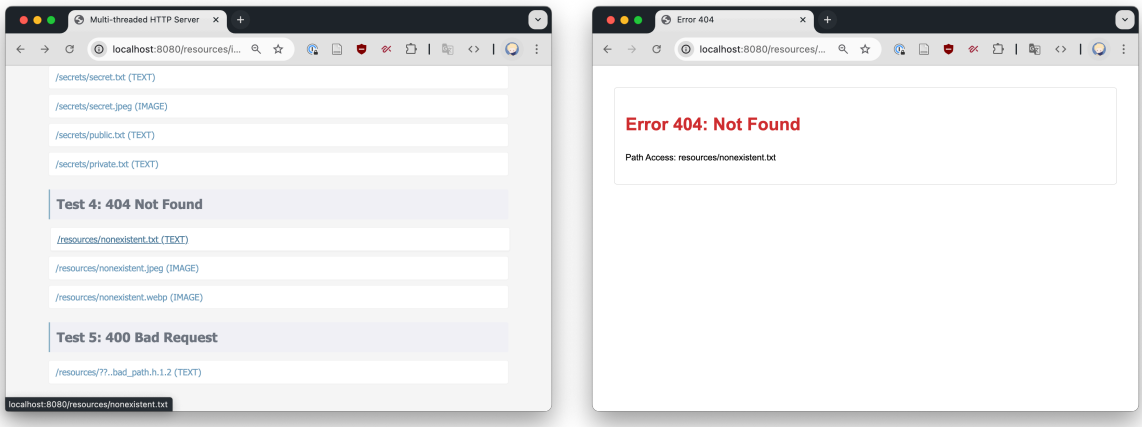


Screenshots 5: Browser testing results for `/secrets/secret.txt`



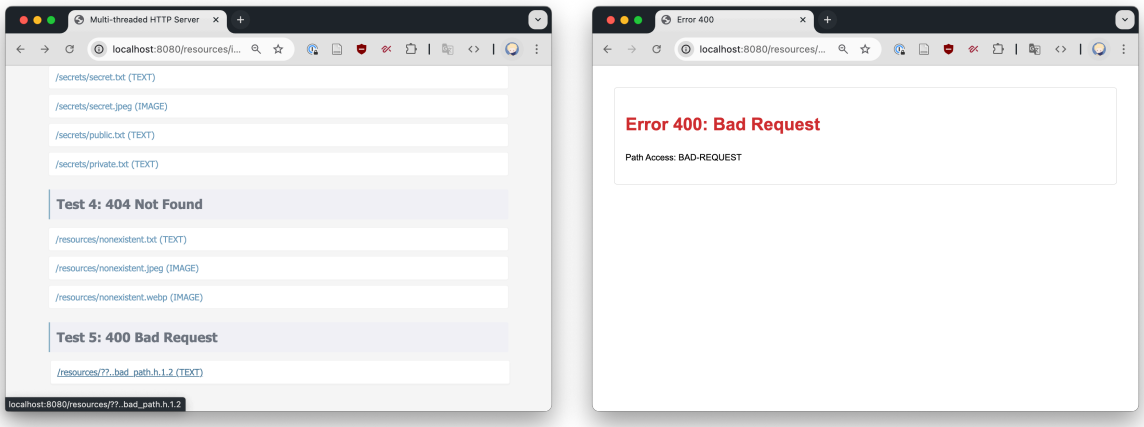
Screenshots 6: Browser testing results for `/secrets/secret.jpeg`

C4. Test 4 - 404 Not Found



Screenshots 7: Browser testing results for `/resources/nonexistent.txt`

C5. Test 5 - 400 Bad Request



Screenshots 8: Browser testing results for `/resources/nonexistent.txt`

D. Server Log Output

```
Server arguments:
        host: 127.0.0.1
        port: 80
        max_requests: 1000
        max_threads: 100
        timeout: 15
        max_conn: 100
        cache_age: 86400
        log_file: server.log

[!] port 80 is already in use
Server started on 127.0.0.1:8080
Received connection from ('127.0.0.1', 51324)
[*] File sent: resources/index.html
[*] File sent: resources/style.css
[!] File not found: resources/favicon.ico
[*] File sent: resources/big.txt
[*] File sent: resources/cat.jpg
[*] File sent: resources/dog.webp
[!] Unsupported content type: pdf
[!] Access denied: secrets/secret.txt
[!] Access denied: secrets/secret.jpeg
[!] File not found: resources/nonexistent.txt
[!] Invalid path: /resources/??..bad_path.h.1.2
[*] Connection timeout: ('127.0.0.1', 51324)
[*] Connection closed gracefully: ('127.0.0.1', 51324)
^CShutting down server...
```

server.log

```
12  [2025-04-21 17:15:58] Server Started
11  [2025-04-21 17:16:03] 127.0.0.1:51324 - resources/index.html - 200
10  [2025-04-21 17:16:03] 127.0.0.1:51324 - resources/style.css - 200
 9  [2025-04-21 17:16:03] 127.0.0.1:51324 - resources/favicon.ico - 404
 8  [2025-04-21 17:16:07] 127.0.0.1:51324 - resources/big.txt - 200
 7  [2025-04-21 17:16:10] 127.0.0.1:51324 - resources/cat.jpg - 200
 6  [2025-04-21 17:16:16] 127.0.0.1:51324 - resources/dog.webp - 200
 5  [2025-04-21 17:16:20] 127.0.0.1:51324 - resources/project.pdf - 415
 4  [2025-04-21 17:16:23] 127.0.0.1:51324 - secrets/secret.txt - 403
 3  [2025-04-21 17:16:25] 127.0.0.1:51324 - secrets/secret.jpeg - 403
 2  [2025-04-21 17:16:29] 127.0.0.1:51324 - resources/nonexistent.txt - 404
 1  [2025-04-21 17:16:33] 127.0.0.1:51324 - BAD-REQUEST - 400
13  [2025-04-21 17:16:49] Server Closed
```

These screenshots display the `server.log` file and console log outputs that were generated as a direct result of executing all the browser tests documented in **Subsections C1 through C5** above. Both the server log file and console output confirm that the HTTP server correctly returned the appropriate status codes for each test case, validating the server's proper response handling.

E. Cache Testing Results

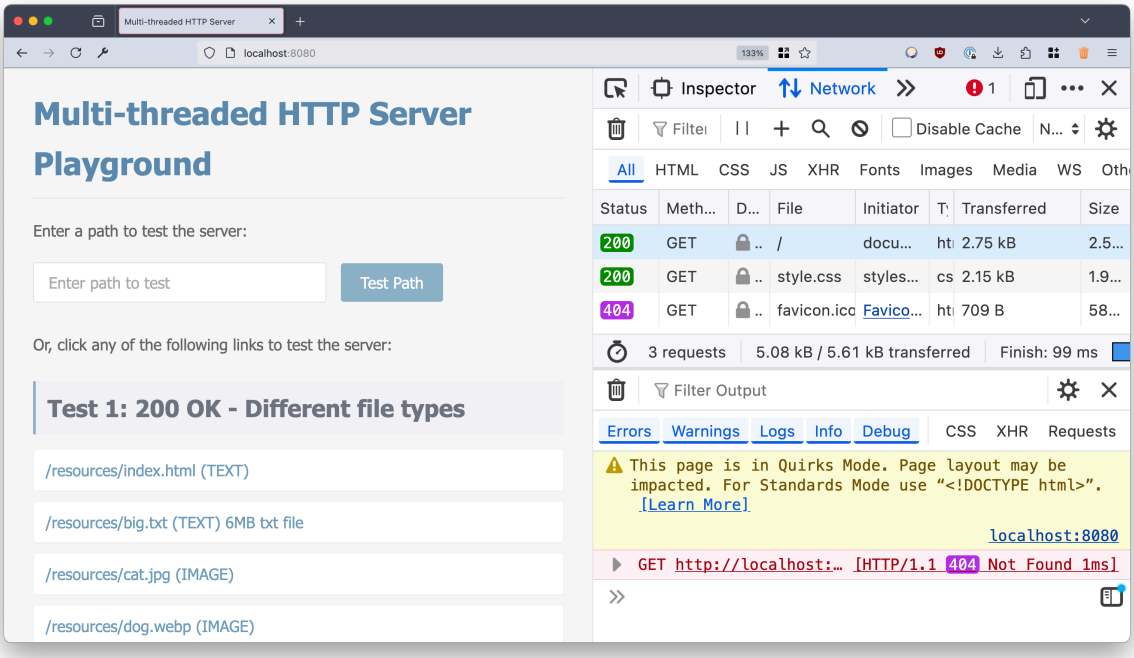
The caching test (ability to handle `If-Modified-Since` header) is actually already thoroughly tested by the

- `test_304`,
- `test_new_date`,
- `test_old_date`

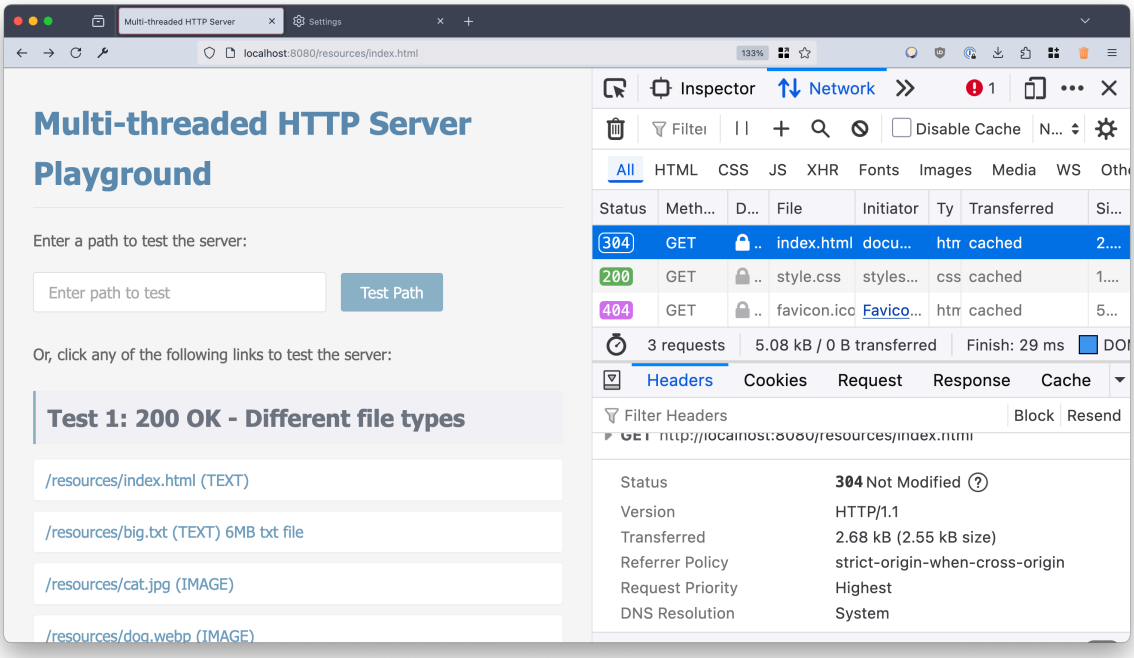
unit tests (see Table 2) in `tests/test_server.py`, and were all passed as shown in Figure 6 (see page 23). Therefore, it can be confidently concluded that the server is able to handle `If-Modified-Since` header correctly.

However, just for illustrative purposes, I further show its handling procedure by:

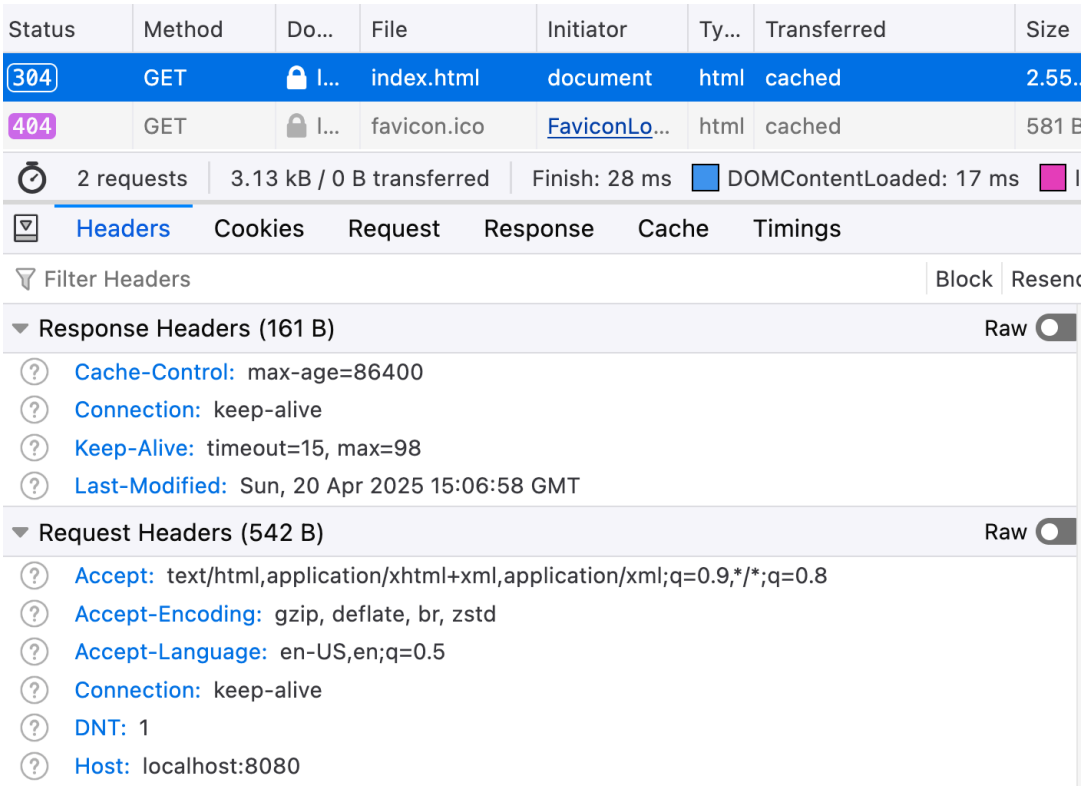
1. Starting an fresh session in Firefox (cache cleared)
2. Enable inspector tools and navigate to the *Network* tab
3. Visiting the `index.html` page for the first time
4. Re-open the browser and visit the `index.html` page again



Screenshots 9: Accessing the `index.html` page for the **first time**



Screenshots 10: Accessing the `index.html` page for the **second time**



Screenshots 11: The **response header** of the **second access**

During the first access (Screenshots 9), the server responded with a `200 OK` status code. Then, after refreshing the page (Screenshots 10), the server responded with a `304 Not Modified` status code, a `Cache-Control` header and a `Last-Modified` header.