

COMP 2322 Computer Networking
Multi-thread Web Server
Project report

Student Name: YEChenwei

Student ID: 21103853d

Project Implementation:

The `http_client` function receives the client socket object as an argument. It works following the progress below: receiving the client request using the socket's `recv` method, sending the request to the `http_server` function to handle it, getting the response back from the `http_server` function, sending the response to the client using the socket's `sendall` method, and finally closing the socket connection.

The server program uses the Python built-in socket library to create a TCP socket and bind it to a specific IP address and port number. It sets the maximum access number to 4 and listens for incoming client connections. Once a client connects to the server, the program creates a new thread to handle the client request.

The code first parses the HTTP request header to extract the request type, file name, and other key information. If the request type is GET, the server tries to read the requested file. The program also implements a simple caching mechanism to handle the If-Modified-Since HTTP header. If the request header contains an If-Modified-Since header in the request, the server checks if the requested file has been modified since the specified time. If the file has not been modified, the server sends a 304 Not Modified response back to the client, and if it has been modified, it constructs an HTTP response with the requested file content and other headers, including Last-Modified header. The server sends a 200 OK response with the updated file content.

The code sets the Content-Type based on the file extension and reads the file in a different way depending on the file type. For instance, for text files, it decodes the file content as a string.

Finally, the code builds the response, including the HTTP status line, Last-Modified and Content-Type headers, and the file contents. The function also writes information about the request, such as the client IP address, access time, requested file name, and response status code, to a log file named `recording.log`.

Multi-threaded Web server

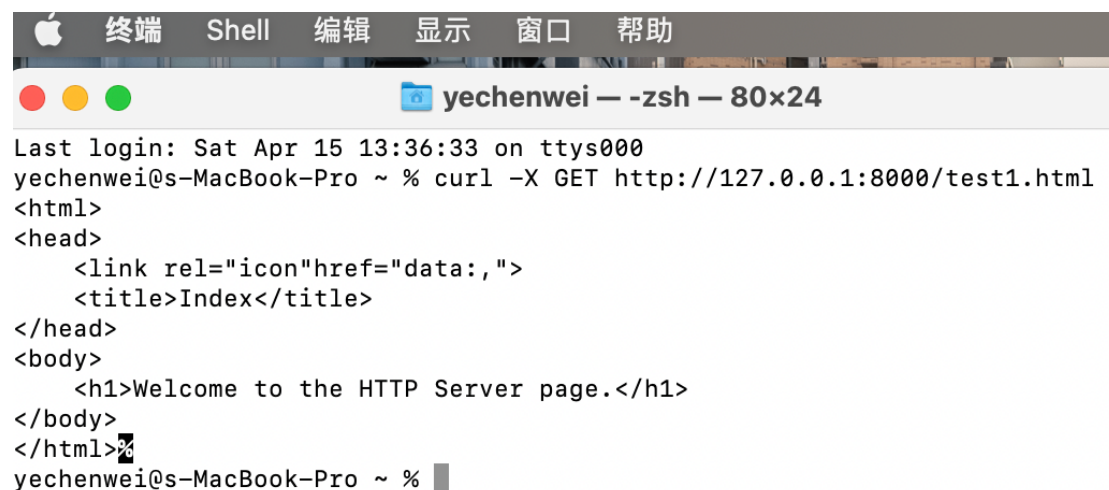
This Python code can handle multiple requests simultaneously using multithreading. When the server starts, it creates a TCP socket binding to specific IP address and port number. The server then enters into an infinite loop and waits for incoming client connections.

When a client connection is established, the server creates a new thread to handle the request from the client. The `http_client` function receives the client connection socket object as an argument and then reads the HTTP request from the client. The received request is passed to the `http_server` function for further processing.

After running main part of the programme, the `http_client` function closes the client connection socket object, and the thread terminates. The server goes back to the infinite loop to wait for new client connections.

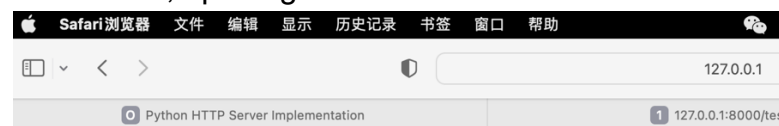
Demonstration of functions:

1. Multi-thread function. Start with running HTTP Server, type the command “`curl -X GET http://127.0.0.1:8000/test1.html`” in the terminal, the respond shows as below.



```
终端 Shell 编辑 显示 窗口 帮助
yeichenwei — -zsh — 80x24
Last login: Sat Apr 15 13:36:33 on ttys000
yeichenwei@s-MacBook-Pro ~ % curl -X GET http://127.0.0.1:8000/test1.html
<html>
<head>
  <link rel="icon"href="data:,">
  <title>Index</title>
</head>
<body>
  <h1>Welcome to the HTTP Server page.</h1>
</body>
</html>
yeichenwei@s-MacBook-Pro ~ %
```

Meanwhile, opening the Safari and Chrome to access the same file.



Welcome to the HTTP Server page.



Welcome to the HTTP Server page.

The test result shows that my HTTP Server will be able to handle multiple requests at the same time.

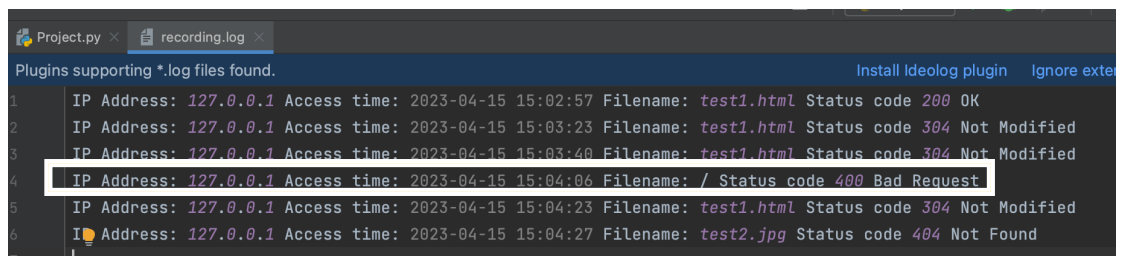
2. Status code.

- 1) 400 Bad requests. Test by issuing the command in MacOS terminal:
`curl -X 123 http://127.0.0.1:8000/`. These three figures showed below are from terminal, log file, and compiler correspondingly.

```
yechenwei@s-MacBook-Pro ~ % curl -X 123 http://127.0.0.1:8000/
```

```
Request Not Supported%
```

```
yechenwei@s-MacBook-Pro ~ %
```

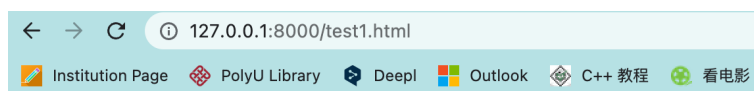


```
Project.py x recording.log x
Plugins supporting *.log files found.
1 IP Address: 127.0.0.1 Access time: 2023-04-15 15:02:57 Filename: test1.html Status code 200 OK
2 IP Address: 127.0.0.1 Access time: 2023-04-15 15:03:23 Filename: test1.html Status code 304 Not Modified
3 IP Address: 127.0.0.1 Access time: 2023-04-15 15:03:40 Filename: test1.html Status code 304 Not Modified
4 IP Address: 127.0.0.1 Access time: 2023-04-15 15:04:06 Filename: / Status code 400 Bad Request
5 IP Address: 127.0.0.1 Access time: 2023-04-15 15:04:23 Filename: test1.html Status code 304 Not Modified
6 IP Address: 127.0.0.1 Access time: 2023-04-15 15:04:27 Filename: test2.jpg Status code 404 Not Found
```

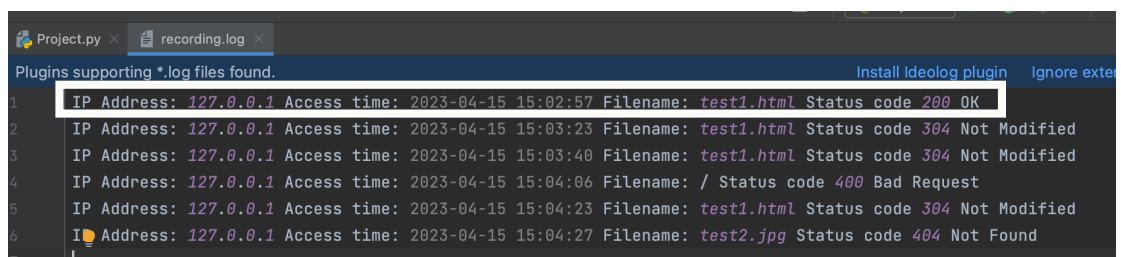
```
Selected Headers:
123 / HTTP/1.1
Host: 127.0.0.1:8000

b'HTTP/1.1 400 Bad Request\n\nRequest Not Supported'
Content-Type: text/html
```

- 2) 200 OK. Test simply by accessing test1.html:
`http://127.0.0.1:8000/test1.html`. These three figures showed below are from Chrome browser, log file, and compiler correspondingly.



Welcome to the HTTP Server page.



```
Project.py x recording.log x
Plugins supporting *.log files found.
1 IP Address: 127.0.0.1 Access time: 2023-04-15 15:02:57 Filename: test1.html Status code 200 OK
2 IP Address: 127.0.0.1 Access time: 2023-04-15 15:03:23 Filename: test1.html Status code 304 Not Modified
3 IP Address: 127.0.0.1 Access time: 2023-04-15 15:03:40 Filename: test1.html Status code 304 Not Modified
4 IP Address: 127.0.0.1 Access time: 2023-04-15 15:04:06 Filename: / Status code 400 Bad Request
5 IP Address: 127.0.0.1 Access time: 2023-04-15 15:04:23 Filename: test1.html Status code 304 Not Modified
6 IP Address: 127.0.0.1 Access time: 2023-04-15 15:04:27 Filename: test2.jpg Status code 404 Not Found
```

```

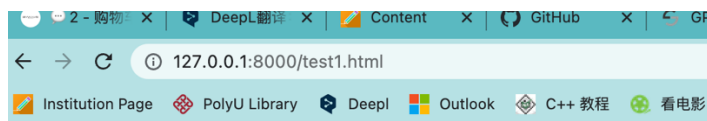
/usr/local/bin/python3.9 /Users/yeichenwei/Desktop/22-23-2/COMP2322/PythonProject/Project.py
This is my HTTP Server: http://127.0.0.1:8000/test1.html
Selected Headers:
GET /test1.html HTTP/1.1
Host: 127.0.0.1:8000
Connection: keep-alive

HTTP/1.1 200 OK
Last-Modified: Sat, 15 Apr 2023 14:25:31 GMT

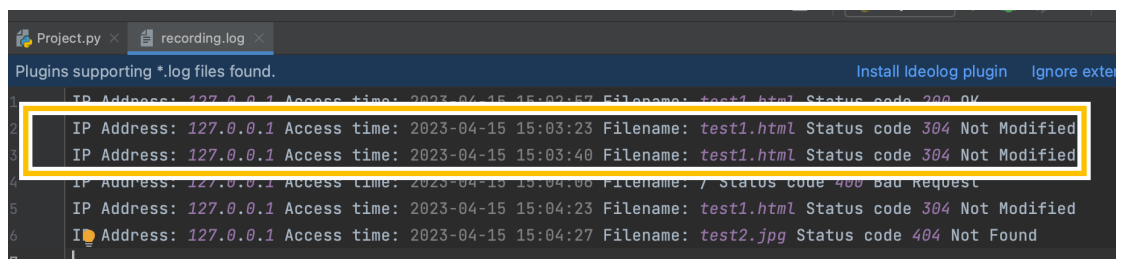
<html>
<head>
  <link rel="icon"href="data:,">
  <title>Index</title>
</head>
<body>
  <h1>Welcome to the HTTP Server page.</h1>
</body>
</html>
Content-Type: text/html

```

- 3) 304 Not Modified. Test by accessing test1.html twice:
<http://127.0.0.1:8000/test1.html>. These three figures showed below are from Chrome browser, log file, and compiler correspondingly.



Welcome to the HTTP Server page.



```

This is my HTTP Server: http://127.0.0.1:8000/test1.html
Selected Headers:
GET /test1.html HTTP/1.1
Host: 127.0.0.1:8000
Connection: keep-alive

HTTP/1.1 200 OK
Last-Modified: Sat, 15 Apr 2023 14:25:31 GMT

<html>
<head>
  <link rel="icon"href="data:,">
  <title>Index</title>
</head>
<body>
  <h1>Welcome to the HTTP Server page.</h1>
</body>
</html>
Content-Type: text/html

Selected Headers:
GET /test1.html HTTP/1.1
Host: 127.0.0.1:8000
Connection: keep-alive

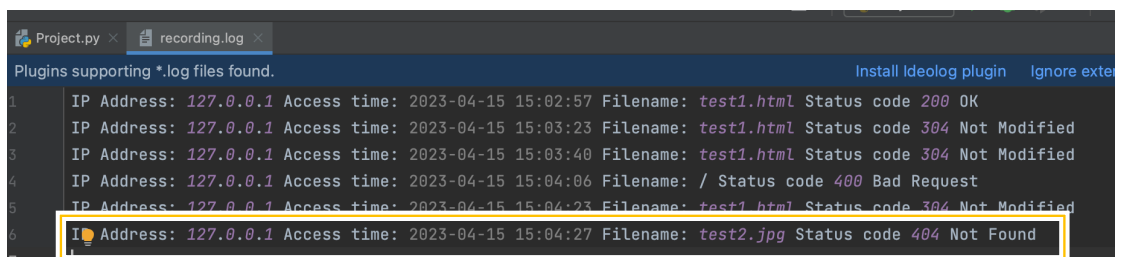
If-Modified-Since: Sat, 15 Apr 2023 14:25:31 GMT
HTTP/1.1 304 Not Modified

```

- 4) 404 Not Found. Test by accessing a file not existed in root folder like: <http://127.0.0.1:8000/test2.jpg>. These three figures showed below are from Chrome browser, log file, and compiler correspondingly.

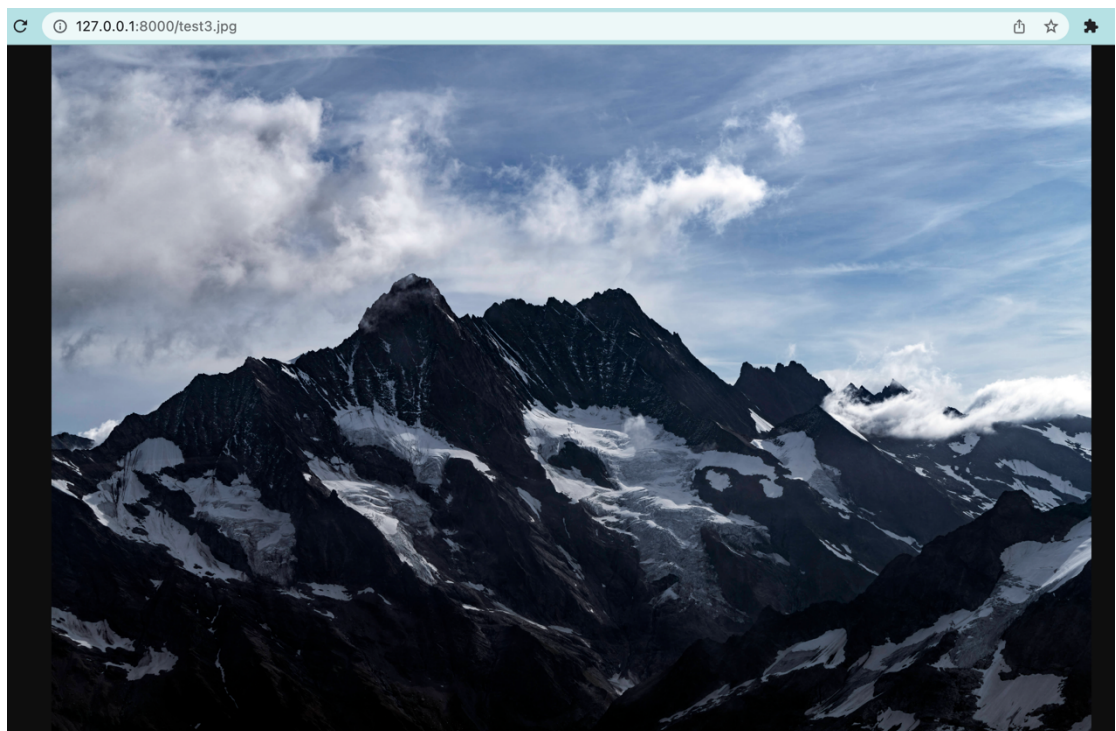


File Not Found



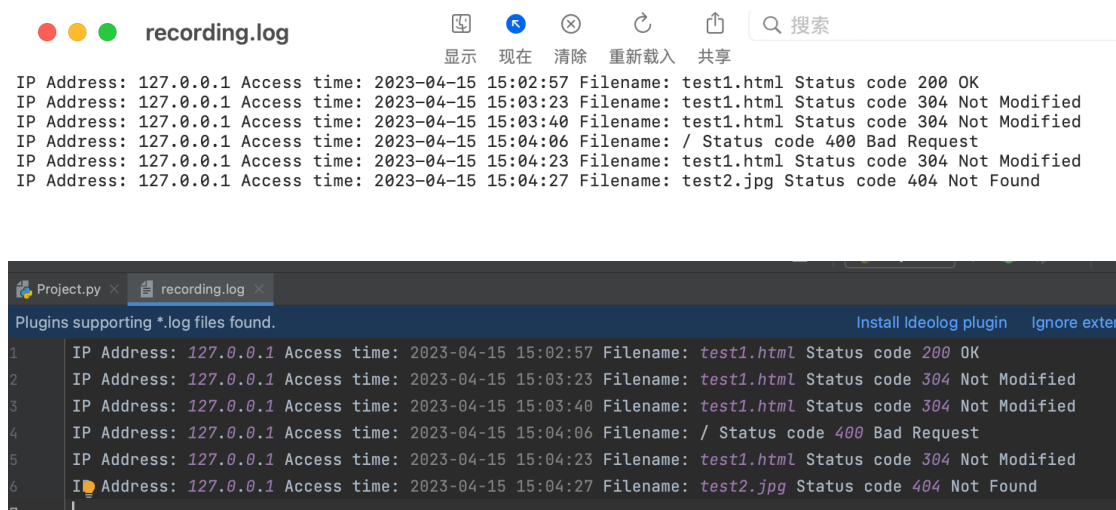
```
Selected Headers:  
GET /test2.jpg HTTP/1.1  
Host: 127.0.0.1:8000  
Connection: keep-alive  
  
HTTP/1.1 404 Not Found  
  
File Not Found
```

- 5) The ways to test JPG/PNG file are similar as mentioned. Here is an example of accessing jpg file.



3. Log file:

It records the historical information based on those tests above.



The image shows two screenshots. The top screenshot is a file explorer window titled 'recording.log'. It displays a list of log entries with columns for IP Address, Access time, Filename, and Status code. The entries are as follows:

IP Address	Access time	Filename	Status code
127.0.0.1	2023-04-15 15:02:57	test1.html	200 OK
127.0.0.1	2023-04-15 15:03:23	test1.html	304 Not Modified
127.0.0.1	2023-04-15 15:03:40	test1.html	304 Not Modified
127.0.0.1	2023-04-15 15:04:06	/	400 Bad Request
127.0.0.1	2023-04-15 15:04:23	test1.html	304 Not Modified
127.0.0.1	2023-04-15 15:04:27	test2.jpg	404 Not Found

The bottom screenshot is a code editor window showing the same log data as a list of lines. The lines are numbered 1 through 7. The log data is as follows:

```
1 IP Address: 127.0.0.1 Access time: 2023-04-15 15:02:57 Filename: test1.html Status code 200 OK
2 IP Address: 127.0.0.1 Access time: 2023-04-15 15:03:23 Filename: test1.html Status code 304 Not Modified
3 IP Address: 127.0.0.1 Access time: 2023-04-15 15:03:40 Filename: test1.html Status code 304 Not Modified
4 IP Address: 127.0.0.1 Access time: 2023-04-15 15:04:06 Filename: / Status code 400 Bad Request
5 IP Address: 127.0.0.1 Access time: 2023-04-15 15:04:23 Filename: test1.html Status code 304 Not Modified
6 IP Address: 127.0.0.1 Access time: 2023-04-15 15:04:27 Filename: test2.jpg Status code 404 Not Found
7
```

Summary:

This Python code is a simplified HTTP server that handles HTTP requests and returns the contents of the file. It focuses on GET requests and sends the requested file back to the client as a response. It binds to a specific IP address and port number. After that, it waits for client connections. When receiving a connection request, it spawns a new thread to handle it and waits for another connection.

The server responds to HTTP GET requests by sending back the content of the requested file. It can handle files such as HTML, txt. If the requested file is not found, it returns a 404 Not Found status code. If the file has not been modified since the last request, it returns a 304 Not Modified status code, which means that the client can use a cached to access the file more efficiently.

The server also logs each request in a file named recording.log, including the IP address, access time, filename, and status code of each request.

Moreover, when using Chrome browser to access local file, web browsers may automatically request the favicon.ico file when requesting a webpage. This is a common issue where a browser will automatically send a request for a favicon, which is a small icon that appears in the browser tab. If the request contains this substring, the log file will always record this information. To solve this question, I add an if condition in client function.


```

def http_client(client_socket):
    # Get the client request
    request = client_socket.recv(1024).decode()

    # Ignore request automatically sent by browser
    if 'favicon.ico' in request:
        client_socket.close()
        return

    # Send HTTP response
    response = http_server(request)
    client_socket.sendall(response)
    client_socket.close()

```

If the request contains this substring, the function assumes it is an automated request from the browser and immediately closes the connection to the client. The result shows it does work. Neither in log file nor the compiler shows the accessing information about this favicon.ico file.

Overall, this project can achieve multi-threaded, and send proper request and response messages. Also, it can use the GET command for both text files and image files. When running the program, it will show the HEAD command after the client received specific files. All of those four types of response statuses: 200 OK, 304 Not modified, 400 Bad Request, and 404 Not Found can be acquired correctly. This program also handles Last-Modified and If-Modified-Since header fields.