HTTP Web Server

Mahan Madani

Abstract

The goal of this project is to simulate the Hypertext Transfer Protocol (HTTP) using Python. HTTP is an application layer protocol so it can be simulated without much trouble using socket programming. This assignment only focuses on the GET method of HTTP and its respective Request and Response formats. Other HTTP methods and features (such as caching) can also be implemented if needed.

Setting Up the HTTP Server

I decided to start with the server before moving on to the client. All of the server code can be found in 'ServerMain.py' located alongside this report.

The server requires an address (in this case, localhost) and a port that is arbitrarily chosen from all available port values (1023<). Afterward, a server socket is created and bound to the host address and port. The server then enters Listening mode, where it awaits new client connections.

Once a client requests a connection with the server, the server can accept the request and start a new thread to handle the client. Each client can send one GET request and receive a response for it, after which the connection is closed.

Both the request and response follow the HTTP message format, containing a status code, a number of headers, and finally, the message body which is an HTML file in this case. If the client requests to view an available webpage, the requested data is sent to it with status code 200. Otherwise, the client will receive status code 404 and an HTML file indicating the requested file wasn't found on the server.

Note that the server is designed to respond to HTTP requests sent from browsers as well. Running the program and opening the server's address from a browser will display 'webpage.html' by default, as shown in Figure 1.

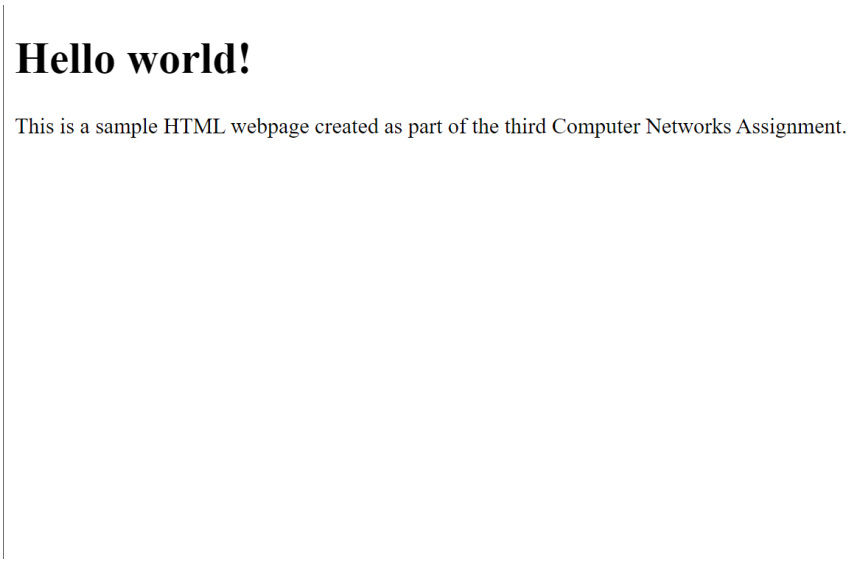

Figure 1: Accessing the server from a browser

Connecting to the server from a browser sends an HTTP GET request to the server. Figure 2 displays this request alongside the server log.

```
Server listening on http://127.0.0.1:3000
Connected to ('127.0.0.1', 61836)
Received request from ('127.0.0.1', 61836)

---------------HTTP REQUEST---------------
GET /webpage.html HTTP/1.1
Host: 127.0.0.1:3000
Connection: keep-alive
sec-ch-ua: "Chromium";v="124", "Google Chrome";v="124", "Not-A.Brand";v="99"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9


Request Type: GET - Path: webpage.html
Requested file found - Status Code 200
Sending response to ('127.0.0.1', 61836)
Connection with ('127.0.0.1', 61836) closed.
```

Figure 2: Browser GET request

As shown, the server can handle GET requests from browsers and send the requested webpage to them. In the next section, the Client side of the program will also be set up to demonstrate how the server handles requests from a custom client.

Setting Up the Client

The client side of the code must have the server's IP address and its port to create a socket. After connecting to the server, the client can send HTTP GET requests to access various files, but it must include the path to the file to be able to access it.

All of the client code can be found in 'ClientMain.py' located alongside this report. Running the client will ask the user to choose a webpage to access, after which the server sends an appropriate response to the client.

I used a basic request template for the client and different requests are generated using this template. The first line of the request indicates its type (GET), the requested file's path, and the version of HTTP being used by the server. The second line contains the host header with the server's address and port.

```
Failed to connect to server. Try again? y/n y
Connected to 127.0.0.1:3000
Please enter the title of the webpage you wish to access (Leave empty to view the default page): webpage.html
Request sent to the server: GET /webpage.html HTTP/1.1

---------------HTTP RESPONSE---------------
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 252

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Webpage</title>
</head>

<body>
    <h1>Hello world!</h1>
    <p>This is a sample HTML webpage created as part of the third Computer Networks Assignment.</p>
</body>
</html>

Connect again? y/n n

Client disconnected.
```

Figure 3: The client's log after a GET request for an existing file

Figure 3 displays the result of requesting a valid webpage from the server. It contains the client's log and the response it received from the server. The response includes the 200 status code and 2 other headers, in addition to the HTML file that was requested.

```
Server listening on http://127.0.0.1:3000
Connected to ('127.0.0.1', 62558)
Received request from ('127.0.0.1', 62558)

---------------HTTP REQUEST---------------
GET /webpage.html HTTP/1.1
Host: 127.0.0.1:3000
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9

Request Type: GET - Path: webpage.html
Requested file found - Status Code 200
Sending response to ('127.0.0.1', 62558)
Connection with ('127.0.0.1', 62558) closed.
```

Figure 4: The Server's log after a GET request for an existing file

Figure 4 displays the result of requesting a valid webpage from the server. It contains the server's log and the request it received from the client. We can see the server has found the requested file, sent it to the client over a socket and finally closed the connection with the client.

Status Code 404: Not Found

If the client requests a non-existent webpage, the server will not be able to find it and as such, will send the default File Not Found webpage to the client alongside status code 404. The connection will be terminated regardless of the status code, so the client must establish a new connection and request the correct webpage later.

Figure 5 displays both the client log and the server log for the interaction described above. We can see the response includes the 404 status code and the notfound.html file.

```
Request sent to the server: GET /random.html HTTP/1.1

---------------HTTP RESPONSE---------------
HTTP/1.1 404 Not Found
Content-Type: text/html
Content-Length: 183

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Error 404</title>
</head>
<body>
    <h2>Error 404:</h2>
    <p>webpage not found! :(</p>
</body>
</html>
```

```
Connected to ('127.0.0.1', 62624)
Received request from ('127.0.0.1', 62624)

---------------HTTP REQUEST---------------
GET /random.html HTTP/1.1
Host: 127.0.0.1:3000
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64
Accept: text/html,application/xhtml+xml,applica
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9

Request Type: GET - Path: random.html
Requested file not found - Status Code 404
Sending response to ('127.0.0.1', 62624)
Connection with ('127.0.0.1', 62624) closed.
```

Figure 5: The result of requesting a non-existent file. The client's log is displayed on the left and the server's log is displayed on the right.

Conclusion

Knowing the general format of HTTP messages and the way the protocol works allows for a rather straightforward implementation. The program can be expanded to simulate additional functions with specific response and request formats. HTTP is a complex protocol but when broken into smaller pieces, it's easy to understand.