High Performance Programming and Systems

Assignment Four

Webserver

Rosa Meyer Jacob Julius Gerlach-Hansen Daniel Riegels
(Group 77)

January 8, 2023

Introduction

In this document, we will briefly describe our implementation of our webserver. To run our server, and the testcases shown in this document, the contents of the included files should be placed in a folder. A terminal should then navigate to this folder, and run the file server.py, by executing "python server.py". To run our client, another terminal should be opend and run command "python client.py". The results of the first 11 test will then be shown i the client terminal. To run the browser-tests, open a browser and type the "127.0.0.1:12345" in the addressbar. This will display test 12. To run test 13, write "http://127.0.0.1:12345/Dog_meme.png" in the addressbar.

Web server design

Our webserver is made using the python programming language, and the locale, math, os, sys, re, socketserver, gzip, struct and datetime libraries. We took inspiration from the excercise server "framework" and modified that to achieve our own implementation. We designed the server using a object-oriented approach, with our main classes being the HTTPServer and RequestHandler. The main design of the server offcurs in RequestHandler. This class, and all the concluded methods, handle the HTTP requests, and makes sure everything is prossesed correctly. The central method for handling the requests is handle(). handle() processes the response by converting the request to a readable format, then uses the readable format to ask the sub methods to process and respond to the appropriate request. handle() then combines the responses, and either calls the respond method or the handle_error() method which modifies the appropriate response before sending to the client. We made handle methods for each of the required request headers, which are all handled by the handle_headers method.

Web servers limitations

Our web-server has many limitations. Keeping in mind that the task was to implement a HTTP compatible webserver, as well as the fact that we should implement simple ways of demonstrating each header, this excuses us for some of the shortcomings. That being said, we know we do not:

- Support HTTP versions different from 1.1
- Accept all file-types or encoding-types described in the RFC
- Support a myriade of request-headers
- Support large file transfers
- Support non-local server-requests
- Block many possible hacking possibilities

We have not thoroughly investigated the limits of file-transfer sizes, security, maximum request amount etc. But we are able to run our server, and make HTTP requests both from our home-made python client as well as directly from both Firefox and Safari browsers (Probably more, we have not tested).

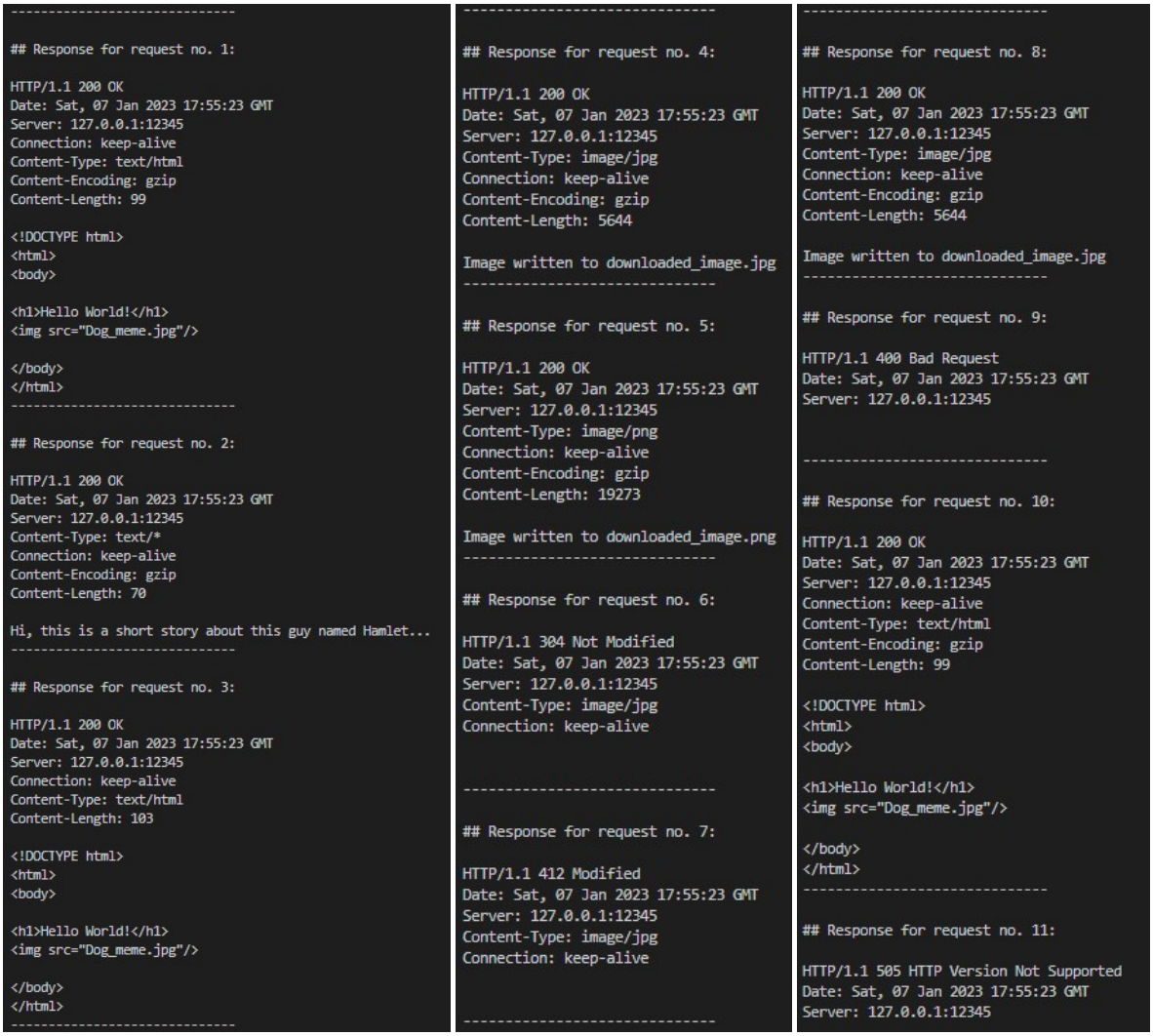
Testing our web server

We made many both white- and black-box tests during the development of the server. Many of these running tests were made to ensure that the individual methods worked correctly independently. With our current state, we made 11 unique white-box tests of the full HTTP request. As we knew each header had to be supported, these 11 tests are made to touch upon each header. The tests were:

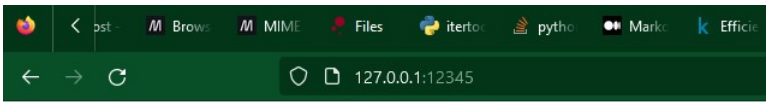
1. Correct request: root folder, encoded
2. Correct request, specific textfile
3. Correct request, no encoding
4. Correct request, jpg requested
5. Correct request, png requested
6. Modified since, should not return the file
7. Unmodified since, should not return the file
8. Unmodified since, should return the file
9. Unsupported method

10. Wrong Host
11. Wrong HTTP version
12. Browser call
13. Browser call for png file

Their results can be seen here:

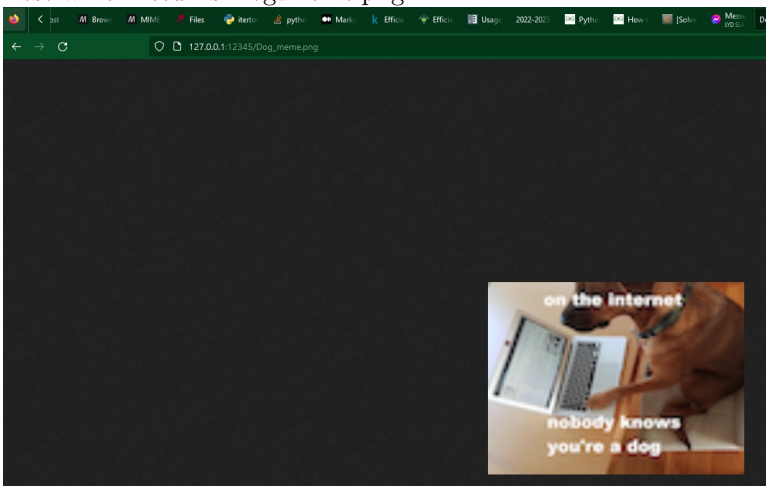


Test which returns index.html



Hello World!

Test which returns Dog_meme.png



As can be seen from the test results above, all tests return an appropriate error message if failed, as well as a 200 OK if accepted with the correct entity body. All responses include appropriate response headers for the given test case. Files are returned in a readable format, and can be read by a browser which is our way of making an easy validation.

Tests that could have made sense, depending on the requirements for the server, are of course the ones described in the limitations tab, i.e. file-transfer sizes, security, maximum request amount etc. In connection to the requirements for the given task, we are however very pleased with the results.

Supported headers and requirements

We implemented support for the request headers given by the task. To store values inter-header_handley, we stored values in the self parameter, eg. response headers. The following lists the supported headers and describes how we meet

the requirements:

- Host: As Host specifies the host and port number, and we run our server locally, host just had to be equal to the server ip and port, and return 400 Bad request if it is not.
- Accept: Accept specifies the prioritisation of MIME type and MIME subtype of the requested file. We implemented this by constructing a dict of our own choice of possible types and subtypes, as well as some logic about how to handle the content negotiation on the server in regards to the q values. The server now searches for filetypes following the requested q-value prioritisation, adds the url for the file to the self-parameter and appends a response header to self.response_headers.
- Accept-Encoding: We implemented Accept-Encoding to work only for gzip encoding. We check if the encoding is requested, store the encoding-type in self.encodeing, and writes 406 Encoding not acceptable if gzip was not requested.
- Connection: We recieve the requested connection state, and writes the requested connection state as a resonse header.
- If-Modified-Since, If-Unmodified-Since: These work in the same way, but reverse of each other. The last date/time of the requested file is found, and compared to the date given by the request header. Does it match the requirement, the file will be sent. If not, in the case of the If-Modified-Since, will write a 304 and the If-Unmodified-Since will write a 412.
- User-Agent: As useragent takes everything, and only fails in very specific circumstances, we implemented this by acknowledgeing that it is there with its own method that does nothing.

These were the request-headers. An appropriate amount of response headers are always returned, depending on the request. As can be seen in the tests, an error message will always include date and server as well. A successful request will include both date and server, as well as the appropriate additional information. That could per example be Connection, Content-Type, Content-Encoding, Content-Length, as well as the entity body of course.