Project 1: Socket Programming

Introduction to Data Communication Networks (2024 Fall)

Instructor: Prof. Kyunghan Lee

TAs: Minjae Lee, Wonjun Bang

Deadline: October 29th, 2024

Project Overview

In this project, we will be implementing a common networked application, HTTP Server using socket programming.

In this document, we will first give you a general introduction to HTTP server. We will then give details on what behavior you should expect from your implementation, and what you will have to implement to complete the application. Make sure to read them all carefully.

We included a **TA implementation(compiled)** HTTP server so you could compare them to your implementation and check if everything is working properly. We also included a generous number of hints in the skeleton code.

Details on grading and submission will be presented in the “Grading” and “Submission” sections separately.

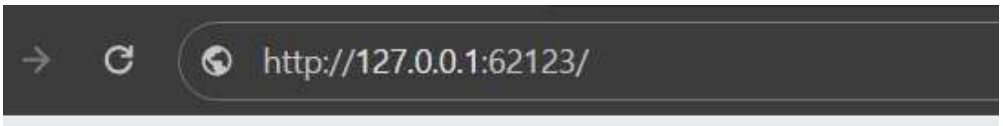
Application: HTTP server

A. Introduction to HTTP server

The Hypertext Transfer Protocol, or HTTP, is used to transfer web page elements such as HTML files or CSS files across the Internet. HTTP clients are usually web browsers, such as Chrome, Firefox, or Safari. HTTP is the protocol that these clients (web browsers) communicate with web servers, such as Apache or NginX to retrieve web pages from the internet. We will use HTTP 1.0 for our project.

HTTP communication is started by the client connecting to the web server using TCP. The port number for HTTP server is usually 80, but in our project, we will be using port **62123**, as port 80 is usually blocked by ISPs / OS in many environments. You can specify which IP and port you are connecting to by entering the following URL in your web browser:

http://<IP>:<PORT>/



You can run the included TA binary and check out the expected result of your HTTP

server, using the following steps. We assume that you have already extracted the project files inside your development environment and have VS code with a terminal open in the development directory.

1. Grant execution permission on the included binaries.

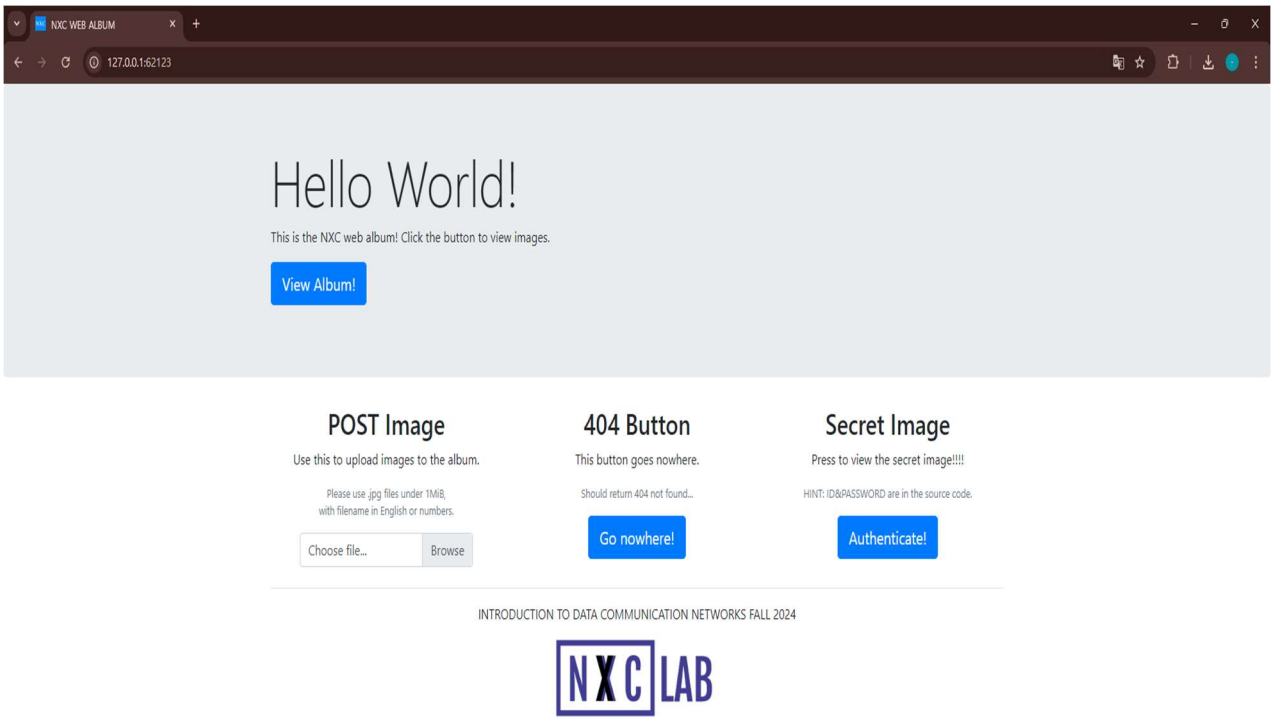
chmod +x ./http_server*

2. Run the appropriate binary depending on your system.

./http_server_<YOUR_SYSTEM> 62123

3. Connect to the server by entering http://127.0.0.1:62123/ as the URL on your favorite web browser. You should be greeted with the following web page.

s



ATTENTION! – WINDOWS USERS!

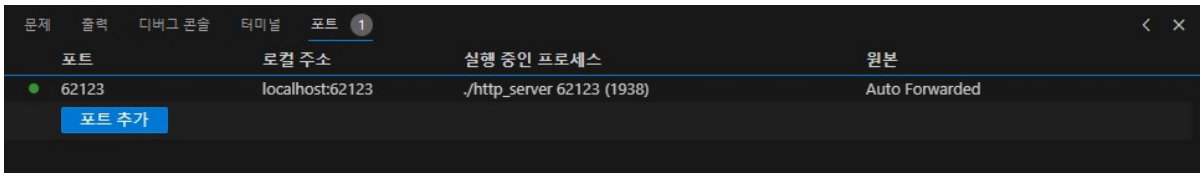
To connect to your web server inside WSL from a browser in Windows, you must **forward your port** using VS code. VS code may auto-forward the port when you run the server application, but if it does not, just follow these steps.

1. Press on the “Port” tab next to the “Terminal” tab in your VS code. Refer to Project 0 on getting to the “Terminal” tab open.

2. Click on the blue “Forward a Port” button.

3. Type 62123 in the textbox and press “Enter”.

You have successfully forwarded your port if your “Port” tab lists are as below.



(You will learn what port forwarding is further down the class when you learn about network address translation or NAT.)

B. HTTP Server Program Behavior

Use your web browser to check the behavior of the HTTP server.

1. When you click on the “View Album!” button, it should take you to a **web album** with 12 images.

2. If you click on the “Browse” button on the “POST Image” section, you should be able to upload a **.jpg image under 1 MB with a filename in English or numbers** to the web album. Try it yourself and check the web album to see if it works.

3. The “Go nowhere!” button will simply return a “404 Not Found” error.

4. The “Authenticate!” button should create a pop-up asking to enter login credentials. Enter **DCN** for the username and **FALL2023** for the password. The page should show a secret image.

If any of the above does not work, please contact the TAs using the Q&A board.

C. Implementation Objectives

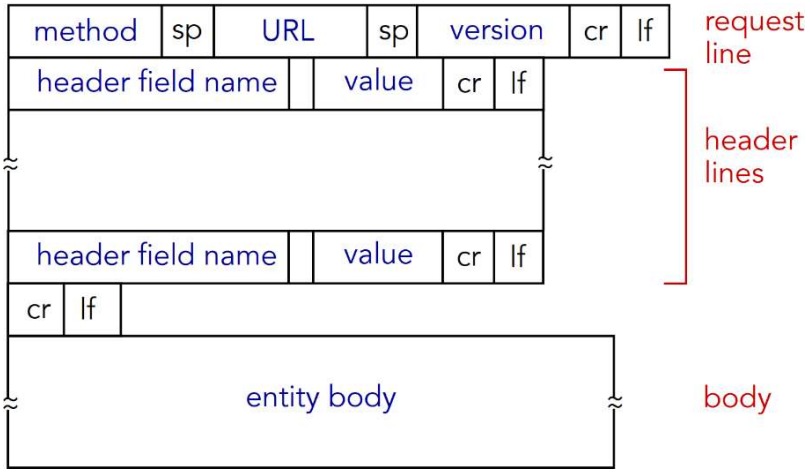
Your server will 1. create a listening socket and accept incoming connection from web browsers, 2. receive and parse the HTTP request, 3. take appropriate action based on the request, 4. create an HTTP response based on the actions it took, and 5. return the HTTP response back to the web browser. Steps 2 ~ 5 will be repeated indefinitely. **You only have to modify the “http_engine.c” file for your implementation.**

1. Creating a Listening Socket and Accepting Connections

The function server_engine() in http_engine.c is called by the main function when starting your server. Initialize a TCP socket and bind it to the server_port argument. Inside an infinite loop, accept incoming connections and serve the connections using the server_routine () function.

2. Receiving and Parsing HTTP Requests

HTTP requests take the following form. Refer to the class materials for details.



“sp” refers to the space character (‘ ‘), “cr” refers to the carriage return character (‘\r’), and “lf” returns to the line feed character (‘\n’).

Inside the server_routine function, a header_buffer character array and an infinite loop is presented. Receive the HTTP header inside the loop, until 1. an end of header delimiter is received (i.e., \r\n\r\n) 2. An error occurs or the client disconnects, or 3. the header message from the client is too long.

After receiving the header successfully, you will have to parse the message to know what the client is requesting. That is, you must extract the method, URL, and header field name & header field value tuples from the received header string.

We recommend you implement the parse_http_header () function, which converts the header string into a http_t struct. http_t struct is a C struct that we created to help you easily manage and manipulate various HTTP elements, using the various helper functions provided in the http_util.c. However, implementing parse_http_header () is **NOT** necessary, and you can parse the header in any way you like.

3. Take appropriate action

The requests from the web browser will be either a GET or a POST method. GET methods are used to retrieve web page elements, such as HTML files or CSS files from the web server. POST methods are used to upload data from the client to the HTTP server, such as user images.

On a GET method, you must return the file requested by the URL as the HTTP response. The body of the GET method HTTP request will be empty.

On a POST method, you must retrieve the file from the HTTP request's body, save it as a file in the server, and update the HTML file to correctly display the newly uploaded file in the web album.

The cases that you must support for the expected server functionalities are documented in the http_engine.c as comments. Please refer to them for details.

(Uploading using GET will not be covered by this project.)

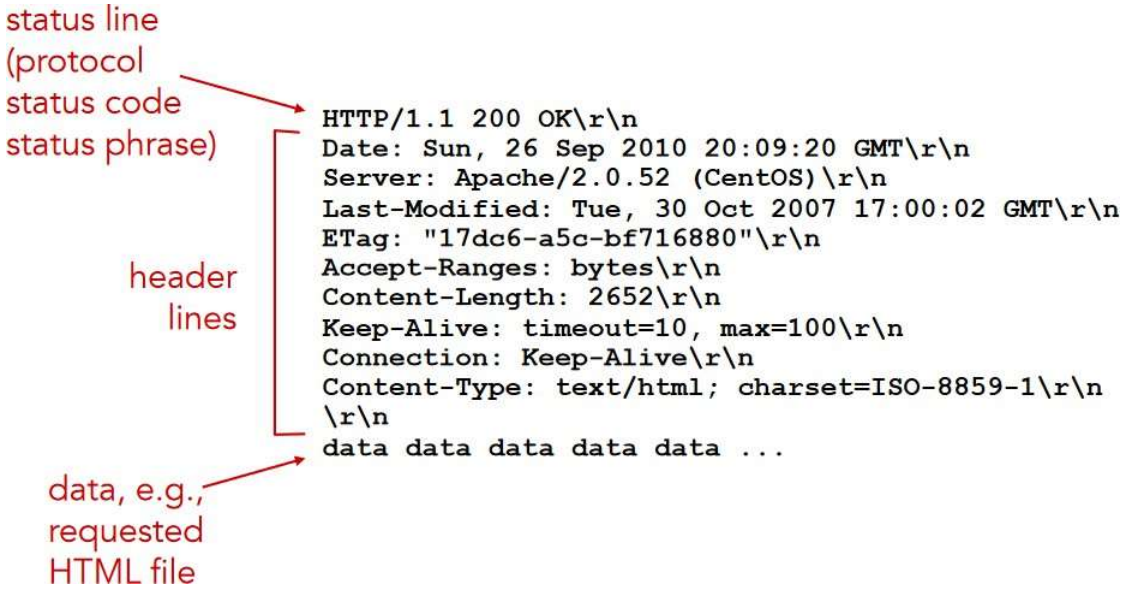
4. Creating HTTP responses

Depending on the client request, and the actions you took, there will be different HTTP responses that should be sent back to your web browser. Again, using the included http_t struct and the helper functions will make things much easier for you. You simply have to initialize a response with the init_http_with_arg () function with the appropriate response code and append the right header fields and body to the response.

An example of using the http_t struct and its functions to create an HTTP response is included in the http_engine.c file, on sending a "431 Header too large" response.

5. Returning the HTTP response

When your HTTP response is ready, you must send the response back to your client. The response message must be formatted according to the HTTP protocol standards, following a similar structure to an HTTP request message, but with a status line instead of a request line.



Fortunately, **we have implemented** the formatting and sending of the HTTP response for you. It will automatically format an http_t struct into a correct HTTP response

message and send it back to your web browser. Checking out the write_http_to_buffer () function may help you better understand the formatting of an HTTP response message and may also help you implement the parse_http_header () function.

6. Dealing with multiple outstanding requests

In the current skeleton implementation, when a request takes a long time to handle, the server cannot process additional requests.

Think about why this happens, and modify the skeleton code in the relevant part to handle multiple outstanding requests.

The easiest way to verify the correctness of your implementation is by creating another internet tab and connecting to the server while playing the video in the "NXC web album" on the webpage.

D. Important Notes

To build your server, simply enter "**make**" and hit enter. Make sure to "make" after saving changes to your source code to run the updated binary. Use ctrl+c to exit the server program.

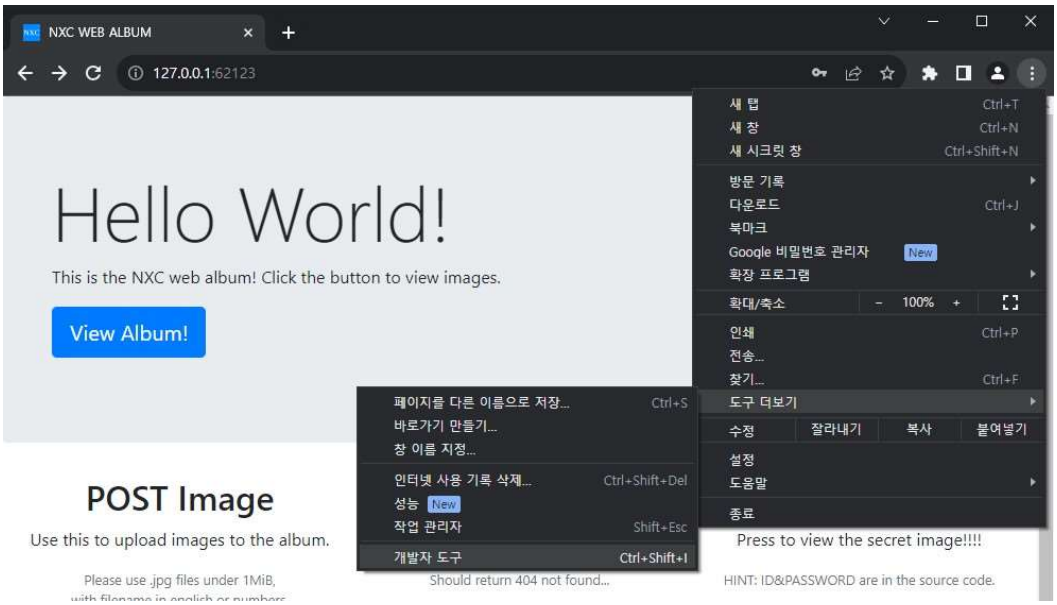
Also, **do not change** the function return values, function name, and the function arguments of any of the defined functions.

You can modify other files and functions for testing purposes, **but you will only submit the "http_engine.c" file.** Therefore, the changes you make on other files will be lost in your submissions.

You are **allowed to define** more functions, global variables, enums, structs, etc. inside "http_engine.c" if you need to. You are also allowed to use additional libraries if it is included in the C / POSIX standard**.** You are allowed to modify the skeleton code if needed, as long as the intended functionalities doesn't change. External libraries that require installation will **NOT** be supported in your grading environment.

Please read the comments in the "http_engine.c" closely, as they contain guides and useful hints on implementing your project. Also, please make good use of the included http_t struct and its helper functions, as it will make the management and manipulation of HTTP elements much easier. We **strongly recommend reading the descriptions and definitions of http_t struct and its functions in http_functions.h and http_util.c** before starting the project.

Also, the "**Web Developer Tools**" of your web browser can help you debug your server. Its "Network" tab allows you to see the HTTP request message and the HTTP response messages exchanged between your server and the web browser.



We will not grade for memory management. However, **understanding C pointers and memory structures will help you greatly in this project**, as this project requires a tremendous amount of dynamic memory allocations, of C string parsing, and reading/writing from/to sockets and files. We strongly recommend getting a general understanding of C pointers and memory structure before starting the project.

Front-end web page elements, such as HTML, CSS, or JavaScript, are also an important part of the web. However, the web front-end is not included in the scope of this project. We provide all front-end elements for this project, and you do not have to modify any of these elements.

Here are some documents that may help you in this project.

HTTP Overview: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

HTTP GET: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET

HTTP POST: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

HTTP Response Codes: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

HTTP Headers: https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

C strings: https://www.geeksforgeeks.org/strings-in-c/

C string functions: https://www.geeksforgeeks.org/c-library-string-h/

C pointers: https://www.geeksforgeeks.org/c-pointers/

Pthreads: https://man7.org/linux/man-pages/man7/pthreads.7.html

Grading

HTTP Server

The HTTP server will be graded on functionality. That is, we will connect to your server implementation using a web browser and check the functionality of your server. These are the functionalities we will be checking to grade your submission.

(5 points) Connection establishment (Port opened and connection accepted)

(5 points) Successful HTTP response message sent back. (of ANY kind)

(10 points) Successful handling of a GET request.
(Status 200, the main index.html page pops up.)

(10 points) Successful handling of a GET request for a non-existent page.
(Status 404)

(10 points) Successful handling of all GET requests except authorization
(All HTTP elements - HTML, CSS, JS, images, etc. - are properly loaded.)

(20 points) Successful handling of a GET request which requires authorization
(Status 401, requesting WWW-Authenticate, and handling BASE-64 codes)

(20 points) Successful handling of a POST message of a .jpg image.
(Receiving image, saving as a file, and updating the HTML file to show the image)

(20 points) Successful handing of multiple outstanding requests
(If you can create another tab to connect to the server while playing the video in the NXC web album, you will receive full credit)

A total of 100 points are allocated for the HTTP server.

Submission

Take your “http_engine.c” file and compress it into a single zip file, with the name “<YOUR_NAME>_<YOUR_STUDENT_ID>_Proj1.zip”. **DO NOT include any other files than “http_engine.c” in the .zip file.** Upload the .zip file to the ETL.

We hope you have fun!

2024. 10. 8.

TAs