# Project 1: A Simple Web Server

**Due: 11:59 pm Friday, April 28, 2023**

## Overview

In this project, we are going to develop a Web server in C/C++. Also, we will take the chance to learn a little bit more about how the Web browser and server work behind the scenes.

All implementations should be written in C/C++ using BSD sockets. **No additional third party libraries are allowed in this project.** You are allowed to use C++ standard libraries up to C++20, such as string parsing, multi-threading.

The objective of this project is to learn basic operations of BSD sockets, understand implications of using the API, as well as to discover common pitfalls when working with network operations.

Please use Project 0's repo as the starting point and test environment and put source files in the `project` subfolder. You are encouraged to use `git` to track the progress of your work, but there is no requirement or extra grades for using git.

You are encouraged to host your code in **private** repositories on GitHub, GitLab, Bitbucket, etc. At the same time, you are PROHIBITED to make your code for the class project public during the class or any time after the class. If you do so, you will be violating academic honesty policy that you have signed, as well as the student code of conduct and be subject to serious sanctions.

## Task Description

The project contains two parts:

- Implement a "Web Server" that dumps HTTP request messages to the console. This is a good chance to observe how HTTP works. You should first start your Web server, and then initiate a Web client. For example, you may open Mozilla Firefox and connect to your Web server. Your Web server should print out the HTTP request header it received.
- Complete your `Web Server` by responding to the client's HTTP request. The `Web Server` should parse the HTTP request from the client, create an HTTP response

message consisting of the requested file preceded by header lines, then send the response directly to the client.

## Specification

For part one, you should be able to explain what the fields in the message mean by looking them up in the textbook or RFC 1945. Do not provide extra debugging output in your submitted code. The client connects to the server and as soon as connection is established, the server sends the content of a file to the client.

For part two, your server should correctly transmit file content as binary data. Whatever data received by an HTTP client should be identical to the data requested, which can be checked by the diff program. If the requested file does not exist, your server should display an HTML page showing **404 not found error**.

Your server should support several common file formats, so that a standard Web browser can **display** such files directly, instead of prompting to save to local disk if the file format is not supported.

The minimum supported file types must include the seven types listed below:

- plain text files encoded in UTF-8: `*.html, *.htm, and *.txt`
- static image files: `*.jpg, *.jpeg, and *.png`

File names are handled in a **case-insensitive** fashion (i.e. `ABC.jpg` and for `abc.JPG` are referring to the same file), therefore an HTTP request for `ABC.jpg` should get a valid response if `abc.JPG` exists on the requested path on the server. There won't be two files in any folder that have the same case-insensitive name.

Your server code also needs to handle the case where the file **name contains space and/or percent**. For example, a file named `new image%file.jpg` on the server should be served correctly if it is requested. Your server does not need to handle requests for files in any subdirectories. All files are supposed to be accessed in the same directory as the actual server program.

Pay attention to the following issues when you are implementing and testing the project.

- If you run the server and a Web browser on the same machine, you may use `localhost` or `127.0.0.1` as the name of the machine.
- After you are done with both parts, you need to test your server. You can first put an HTML file in the directory of your server program. Then, you should connect to the server from a browser using the URL http://<machinename>:8080/<filename> and see if it works. For your server in

Part A, you should be able to see HTTP requests in the console of your server machine. For your server in Part B, your browser should be able to show the content of the requested file (or displaying image). **Note: In the given docker file, we specify port 8080 as the port to be used. The 8080 port in the Docker container is forwarded to the host machine, so you should be able to access it via http://localhost:8080/<filename> in the host's browser.**

### Other Requirements

- The server must open a listening socket on the specified port number (8080).
- The server should run without any command line arguments.
- The server should be able to handle files up to `100 MiB`.
- The server should use HTTP/1.0 or HTTP/1.1.
- Your code should not use any additional library except for those provided during the Docker setup. Please make sure not to use `apt`.
- We will use g++-11 to compile your program. Therefore, if you use any C++-20 feature, make sure it is supported by the compiler.
- (Just in case) Do not try any *advanced* networking API that requires a higher Linux kernel version or privilege to run. Include but not limited to: zero copy message passing, memory mapping, io_uring, packet filters, express data path, etc.

## A Few Hints

- Guide to Network Programming Using Sockets

## Environment Setup

- Project 0 Description

### Notes

- The code base contains the basic `Makefile` and one empty file `server.c`.
- You are now free to add more files in the `project` folder and modify the Makefile to make the `server.c` a full-fledged implementation.
- The default `Makefile` compiles server.c with a C compiler. If you want to use C++, you can rename `server.c` into `server.cpp` and modify the `Makefile`

correspondingly. Note that for the autograder to work, the command `make` (without arguments) must successfully compile the binary `server`.

# Submission Requirements

To submit your project, you need to prepare:

1. A `README.md` file placed in your code (project folder) that includes:
    - Your name, UCLA ID
    - The high level design of your server
    - The problems you ran into and how you solved the problems
    - Acknowledgement of any online tutorials or code examples (except class website) you have been using.
2. All your source code, `Makefile`, in the `project` folder.

    Then submit the resulting archive to Gradescope. You can choose either upload a `zip` file containing the repo (not just the `project` folder) or specify a PRIVATE GitHub repository to submit. If you choose to upload a `zip` file, the files in the repo should be in the root folder of your `zip` file. For example, `project/Makefile` in the repo should be `project/Makefile` in the `zip` file, **not** something like `project1/project/Makefile`.

Note:

1. Your code will be judged by an autograder, and you will know your score shortly after your submission.
2. There is no limit to the number of resubmissions. Only the **last** submission will be used to grade. If you made a late resubmission, we will always apply the penalty without considering the submission history.
3. We will not reveal the details of the autograder. However, if you believe your code is correct but the autograder does not give the expected score anyway, we are glad to examine upon request.

# Grading

Your code will be first checked by a software plagiarism detection tool. If we find any plagiarism, you will not get any credit and we are obliged to report the case to the Dean of Student. You will receive full grades if your demo meets the following grading criterias.

## Grading Criteria

Your code will be graded based on several testing rubrics. We list the main grading criteria below.

1. The server program compiles and runs normally. (12 pt)
2. The server can print out the correct HTTP request messages to the console. (0 pt)
    - We don't test this in the autograder because there could be some issues with encoding and output. But please print the request.
3. The server program transmits a small text file (up to 512 bytes) correctly. (15 pt)
    - Correct HTTP response header (2 pt)
    - The response content is exactly the same as the file. (9 pt)
    - Correct MIME type (4 pt)
4. The server program transmits a large binary file without file extension (up to 1 MB) correctly. (15 pt)
    - Correct HTTP response header (2 pt)
    - The response content is exactly the same as the file. (9 pt)
    - Correct MIME type (4 pt)
5. The server program serves an HTML file correctly and it can show in the browser. (12 pt)
    - Correct HTTP response header (2 pt)
    - The response content is exactly the same as the file. (6 pt)
    - Correct MIME type (4 pt)
6. The server program serves an JPG file correctly and it can show in the browser. (12 pt)
    - Correct HTTP response header (2 pt)
    - The response content is exactly the same as the file. (6 pt)
    - Correct MIME type (4 pt)
7. The server program serves a file whose name is handled in a case-insensitive fashion. (12 pt)
    - Correct HTTP response header (2 pt)
    - The response content is exactly the same as the file. (6 pt)
    - Correct MIME type (4 pt)
8. The server program serves a file whose name contains space correctly in the browser. (6 pt)
    - Correct HTTP response header (2 pt)
    - The response content is exactly the same as the file. (4 pt)
9. The server program serves a file whose name contains % correctly in the browser. (6 pt)
    - Correct HTTP response header (2 pt)
    - The response content is exactly the same as the file. (4 pt)
10. The server program replies 404 to non-existing files. (10 pt)