Project 1: Web Server Implementation

1 Goal

In this project, we are going to develop a Web server in C/C++ that can stream video. Your Web server should implement the following 2 functionalities (more details later):

1. A simple HTTP server to serve files
2. A reverse proxy for video-streaming

We are hoping the implementation will give you a more thorough understanding on how Web browsers and servers work behind the scene, as well as be a fun experience in network programming.

2 Lab Working Environment

Please use the template from `https://github.com/AlanNPC/CS118-F23-Project-1`. Since C/C++ is a cross-platform language, you should be able to compile and run your code on any machine with a C/C++ compiler and BSD socket library installed. No high-level network-layer abstractions (like httplib, Boost.Asio or similar) are allowed in this project. You are allowed to use some high-level abstractions, including C++14 extensions, for parts that are not directly related to networking, such as string parsing.

This project is graded on Ubuntu 22.04 LTS (Jammy Jellyfish). You may develop the project on any compatible Unix environment (including macOS), but we highly recommend testing under the same Ubuntu environment. You may obtain a copy of Ubuntu 22.04 from its official website (`https://releases.ubuntu.com/22.04/`, 64-bit PC AMD64 desktop image) and install it in VirtualBox (`https://www.virtualbox.org`) or other similar VM platforms. Note that the OS you use must have a Web browser that ships with a graphical interface for testing. We do not support Windows for Project 1 because of its different socket programming conventions and behaviors; however, you may use Windows Subsystems for Linux (WSL) (`https://learn.microsoft.com/en-us/windows/wsl/`) or a virtual machine. For macOS users, please also test your code on Ubuntu before submission.

3 Instructions

1. Before you start writing code, read the HTTP sections in Chapter 2 of the textbook carefully. The textbook will help you understand how HTTP works. For details of HTTP, you can refer to RFC 1945 (`https://tools.ietf.org/html/rfc1945`). You should also carefully go over the socket programming slides posted and discussed by the TAs. Note that you must program in **C/C++** for this project, rather than in Java as the textbook shows.

   We have also provided skeleton code for this project; we highly recommend you to read through it before using it.

Note that although we provided the skeleton code in C, you may also submit a C++ solution if you prefer to.
Also, the skeleton code is NOT a mandatory requirement, so if you prefer to write your own solution from scratch, we will accept it as long as it is written in C/C++.

2. First, implement a "simple Web server" by responding to the client's HTTP request. The "simple Web server" should parse the HTTP request from the client, creates an HTTP response message consisting of the requested file preceded by header lines, then sends the response directly to the client. In order to test it, you should first start your Web server, and then initiate a Web client. For example, you may open Mozilla Firefox or Google Chrome and connect to your Web server.

3. Your simple Web server should support several common file formats so that a standard Web browser can display such files directly instead of prompting to save to the local disk. (Hint: set the Content-Type field correctly)

   The minimum supported file types must include the following:

   - plain text files encoded in UTF-8: *.html and *.txt
   - static image files: *.jpg

   Your server should also correctly transmit file content as binary data if the filename does not contain any file extension. Whatever data received by an HTTP client should be identical to the data requested, which can be checked by the `diff` program. (Hint: you can set Content-Type of the HTTP response as application/octet-stream for binary file requests.)

   Additionally, if the client requests a file that does not exist, your program should handle it gracefully (i.e. at the very least it shouldn't crash).

   Your server does **NOT** need to handle the following edge cases:

   - The client requests files in any subdirectories of the server.
   - The client requests a file named with special characters other than alphabets, periods, spaces or % signs.

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

   If you run the server and a Web browser locally (on the same machine), you may use `localhost` or `127.0.0.1` as the name of the machine. **Instead of using port 80 or 8080 for the listening socket, we use port `8081` by default to avoid conflicts. We will also use this port in the grading.** However, you can use the `-b` option to specify other ports while testing.

   After you are done with implementing the web server, you need to test it. 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>:<port>/<filename>` (e.g., `http://localhost:8081/test.html`) and see if it works. Your browser should be able to show the content of the requested file (or display image).

5. Useful tips:

   - It would be helpful for debugging if you print out the HTTP request header that the server received.

- When constructing the HTTP header response, you only need to implement the required header fields. For example, HTTP version, status code, content type, etc.
- If the client does not recognize the body data, please check whether you put a blank line after the last line of your HTTP header, i.e., `\r\n\r\n` .
- If the browser can successfully receive the response of a text file but fails to display the content. This means that your server probably only sends null bytes to the browser. You can verify this by changing the content type to application/octet-stream to download this file and use hexdump to examine the file.

6. After implementing a functioning simple Web server, you can proceed to implement the second functionality: Reverse Proxy. We highly recommend you to use some sort of version control tool (e.g. Git) to keep track of your work, and this is probably a good checkpoint to save and backup your work. For this following part of the project, we have set up a back-end video server. Upon getting a client request for any .ts file, instead of trying to look up the files on the simple Web server, your server should open a socket and connect to the back-end video server, forward the client's request to the video server, and then forward the video server's response back to the client. The default back-end server is at 131.179.176.34 on port 5001. Other backend server and port can be selected using `-r` and `-p` arguments respectively. Please handle it correctly when these arguments are specified (refer to `parse_args` and `proxy_remote_file` in the code template for details).

   Note that 5001 is the only port you are allowed to access on the default server; traffic on other ports may be blocked by the firewall and multiple attempts may lead to your IP address being banned from the server, so please refrain from using other ports.

4 Grading Criteria

Your code will be first checked by a software plagiarism-detecting tool. If we find any plagiarism, you will not get any credit, and we are obliged to report the case to the Dean of Students. Your code will be graded based on several testing rubrics. We list the main grading criteria below.

Simple Web server Functionality:

- The server program compiles and runs normally.
- The server program transmits a binary file (up to 1 MB) correctly. We will test binary file transmission by requesting a filename with no extension. You can test this function with the following commands.

```
$ cat /dev/urandom | head -c 1000000 > binaryfile # generate a 1MB file
$ curl -o downloadfile <machinename>:<port>/binaryfile # request the file
$ diff downloadfile binaryfile # check if the downloaded file is intact
```

- The server program serves a plain text file correctly, and it can show in the browser.
- The server program serves an image file correctly, and it can show in the browser.
- The server program serves a file with a file name containing white spaces (e.g., ucla icon.jpg).

- The server program serves a file with a file name containing % signs (e.g., ucla%icon.jpg).

Reverse Proxy Functionality:

- The reverse proxy successfully displays the video player in the browser.
- When the client press play, the video will play normally.

5 Project Submission

The project due date is posted on Gradescope. Late submission is allowed by up to three days (10% deduction for initial 24 hours, 20% deduction for between 24 to 48 hours, 40% deduction on 48 to 72 hours, and 100% deduction afterwards).

You need to submit a zip archive that contains all the source code and a `Makefile` that can compile your server program. The `Makefile` should be located in the root directory within your submission. The `make` command will compile the program and **produce a single standalone executable named `server`**.

The server program will be graded by an autograder on Gradescope. You will be able to see part of the test result after submission. We don't limit the number of submission attempts, and the final grading is determined by your *last* submission.