Department of Computer Science and Engineering, College of Informatics, Korea University

COSE342 – Computer Networks (컴퓨터네트워크)

Instructor: Prof. Wonjun Lee

Term Project, Spring 2023

Due: June 12, 2023, 11:59 PM

Term Project: Implementing an HTTP/1.1-Subset Web Server

I. Objective

To understand how web servers and browsers work and interact to load web pages, especially focusing on (1) the underlying HTTP communications, (2) the I/O model Berkeley socket APIs rely on, and (3) how multiple file descriptors are managed by a single control flow through I/O multiplexing.

II. Overview

Web servers play a key role in today's Internet. With the dominant number of computing endpoints supporting HTTP, web servers have become a standard interface to provide various web-based services on the Internet. These services are implemented through the exchange of a *web page,* which consists of HTML, JavaScript, CSS, or image files carefully structured by web developers, or a *web resource* intended for machine interactions to provide so-called web APIs. At the center of these services are web servers that respond to their clients' requests. HTTP is the application-layer protocol web servers and web browsers rely on to deliver messages.

As learned in the class, application-layer protocols rely on a transport-layer protocol to deliver their messages. In this case, HTTP relies on TCP (except for HTTP/3, which is on top of a UDP-based transport-layer protocol, QUIC), which is implemented in the kernel of almost every operating system (*i.e.,* Unix-like operating systems, Apple macOS, Microsoft Windows, *etc.*). These operating systems expose a unified interface called *sockets.* Their origin is the Berkeley Software Distribution system, and therefore many people call them BSD sockets. Though the prevalence of HTTP libraries makes most developers not need to know the details of the socket APIs and simply use more easy-to-use APIs wrapping the socket APIs (*e.g.,* `http.createServer()` in node.js), socket APIs exposed by the kernel are always used by web servers and web browsers under the hood.

Since web servers communicate through HTTP, and HTTP is implemented on top of socket APIs to use TCP, efficiently managing sockets is a paramount task for web servers to achieve high performance, especially when web servers handle many clients simultaneously. One of the well-known approaches for this is I/O multiplexing, the capability to inform the kernel that the application needs to be notified when one or more I/O conditions are ready. Though I/O multiplexing is not used only for this purpose, it is used for multi-client support in this context.

In this project, we are going to implement a simple web server supporting a subset of HTTP/1.1, which is a widespread application-layer protocol most people use every day. Our primary goal is to understand what happens between web browsers and web servers during a web page load at the protocol level (*i.e.,* HTTP) and how these web servers are implemented to support HTTP using BSD socket APIs. By implementing a web server with C, we can become familiar with using socket APIs with a low-level systems programming language and comprehend the I/O model behind socket APIs. Furthermore, by the end of the project, we will be able to understand how I/O models affect performance when a single server is serving multiple clients. Most interestingly, the web server you would implement can be tested through a well-known client such as Chrome or Firefox.

III. Components

The components of this project are two-folds: the implementation part and the description part.

A. Implementation part

The implementation part is the source code of your **web server** and the build script, if any. Your web server handles HTTP clients' requests as defined in the HTTP/1.1 standard. However, not every aspect of the specification is required to be implemented, and only a subset of HTTP/1.1 is addressed in this project, for example, only the GET request method is considered. Your web server simultaneously serves multiple clients using I/O multiplexing with one of the `select()`, `poll()`, or `epoll()` functions. Though **implementing an HTTP client is not required** in this project because production-level HTTP clients (*e.g.,* Firefox, Chrome, or cURL) are off-the-shelf, it is recommended to implement one on your own to test whether your server is able to treat *wrong* requests. However, you should exclude the HTTP client you implemented when submitting your work.

B. Description part

The description part is a project report, which includes how to build your source code, implementation details, the description of the used APIs, and the execution result (*i.e.,* screen dump) described in Section IV.

IV. Requirements

Your work should satisfy the following requirements: The requirements themselves are the evaluation criteria for both the implementation part and the description part. Double-check that your work conforms to the requirements below before submission.

A. Environment

The execution environment is provided as a VM (Ubuntu Desktop 22.04.2 LTS on x86), and the evaluation will take place in the VM. It is free to develop your program in any environment, but check before submission that there are no different behaviors from your development environment when they are done in the provided VM.

The implementation should be written with **(1) the C programming language** and **(2) BSD socket APIs.** Using other programming languages (*e.g.,* Python, Go, Java), high-level wrapper libraries for sockets or I/O operations, and an open-source web server itself (*e.g.,* Apache HTTP Server, Nginx) are not allowed. Your web server should be written from scratch using basic socket APIs. Use `gcc` deployed in the VM to compile your program (*i.e.,* choose an appropriate C version that `gcc` supports). Writing a script for build (*e.g.,* Makefile) is highly recommended. In addition, unexpected crashes or improper memory (de)allocation affect the grading.

Do not use standard output stream (`stdout`). Printing any HTTP messages or others in the standard output is not allowed (*e.g.,* do not use `printf()` function). This requirement is for the automated evaluation, and thus using `stdout` may disrupts the evaluation. Remove related functions or make them comments before submission.

B. A subset of HTTP/1.1

HTTP/1.1 defines various request methods to indicate actions on the target resource. In this project, assume the client only sends a request with the GET method to retrieve a target resource. You must implement the web server that responds to GET requests with the corresponding content of the identified resource. Other methods, such as POST, are not considered in this project.

You must implement a web server as defined in the HTTP/1.1 specification. Your web server should be able to respond to requests from well-known web browsers (*e.g.,* Firefox, Chrome, *etc.*) and any legitimate clients supporting HTTP/1.1 (*e.g.,* cURL or a self-written HTTP client) in a way defined in the standard. Persistent connections of HTTP/1.1 should be supported. Your server maintains a TCP connection after the completion of a response until the client sends a close signal (*i.e.,* Connection: close). Supporting response status codes 200, 400, and 404 is required, and the others are optional. Refer to the specification to see the semantics of the status codes and make your server respond with an appropriate status code.

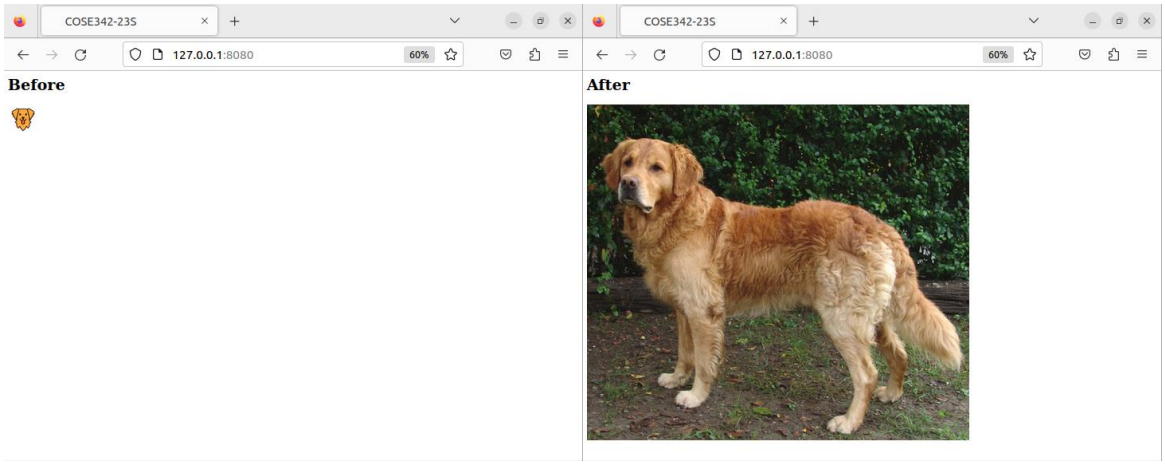

Figure: The initial web page (left). After one second, the image is changed to the larger one (right).

Supporting some HTTP message header fields such as Content-Length, Content-Type, and Content-Encoding is required to load correctly with typical web browsers. If your web server works with the built-in Firefox in the VM without any problems (check the example scenario in the next section), we will not further evaluate the implementation regarding HTTP message header fields. Specifically, supporting more encodings, such as gzip or compress is optional.

We do not cover transport layer security (TLS) over HTTP/1.1 in this project.

C. Multi-client support with I/O multiplexing

Your web server should be able to handle multiple clients' requests simultaneously using I/O multiplexing. In other words, while one client has a connection with your server, the server should be able to handle other requests from other clients. Check the man pages of `select()`, `poll()`, or `epoll()` and **choose one** to support multiple clients with a single control flow. Do not implement it with multi-processing or multi-threading. Support for multiple clients must be implemented with I/O multiplexing.

The server program accepts two command-line arguments. The first argument is a port number your server listens to, and the second argument is a serving directory's path to serve from clients' retrieval. Requests for resources outside of the directory and cases for resources that do not exist, should be addressed correctly (Section IV. B). For example, your server should be run with the following command:

```
$ ./server 8080 resources
```

This command runs your server to listen on port 8080 and to serve `resources` directory and its subdirectories for clients' requests. One exception is the request for `/`. If a client requests `/`, the server responds with `index.html` in the root of the specified directory. In the above example, `resources/index.html` is sent.

Here is an example scenario your web server should handle:

(1) Firefox sends an HTTP request to your web server to fetch `index.html`. That means you enter `http://127.0.0.1:8080/` in the address bar of Firefox to reach your server running on localhost.

(2) The server running in a different process receives the request and responds with the content of `index.html` in the root of the serving directory. Of course, your server should be running before Firefox tries to access it in Step 1.

(3) After Firefox receives `index.html` and parses it, Firefox sends further requests for resources referred to in `index.html` (*i.e.,* `script.js`, `gr-small.png`, and `gr-large.jpg`) since they are required to load the page completely. Subsequently, your web server responds to requests for them, completing the page load in Firefox. During this page load, the TCP connection should be maintained to support HTTP persistent connection.

The resources for this scenario are provided as well, and the figure above describes the correct visual progress of it. **If your server can support this scenario, the basic requirement for serving a client is satisfied.** However, other scenarios with different identifiers or larger files will be tested in the evaluation. Note that other content types and encodings are not considered, as discussed in Section IV.B. Make sure your server can serve more general scenarios than the only scenario above so that running with different paths or files is possible.

D. Description part

Your project report should include all the following materials: (1) how to build your source code; (2) implementation details (especially about how your web server handles multiple clients); (3) the execution results of the example scenario; and (4) the description of the following functions: `socket()`, `connect()`, `bind()`, `listen()`, `accept()`, `send()`, `receive()`, `close()`, `shutdown()`, `select()`, and `poll()`.

V. Evaluation & Deliverable

A. **No excuse will be accepted for late submission!** Any submission after the due date will receive no points.

B. Do not copy the answers of other COSE342 students. **When there is a suspicion that a submission of one student is similar to that of another student(s), the students have obligation to be interviewed for questions related to their work.** There is a demerit mark (severe minus point), if you copy the materials from other students' definitely.

C. You can refer to any information on the Internet (including generative AI), but you should verify the source's authenticity before consulting it. **Uncertain or erroneous information in your work will affect your grade.** In other words, the submission should be written in your own words, and you are responsible for the contents.

D. Submit your project report (the description part) in **PDF format,** not others (*e.g.,* docx. hwp, pages, *etc.*). The filename should be "[COSE342-TP] <student-id>-<name>.pdf"

E. How to submit your work

(1) Compress all your materials, including the source code, build script, and project report, into one file. Do not include the binary files or unnecessary source code (*e.g.,* HTTP client source for testing).

(2) The name of the compressed file should be "[COSE342-TP] <student-id>-<name>.xxx". Its extension might be .zip, .tar,. 7z, etc.

(3) Upload the file on the assignment submission menu in Blackboard.

F. If there is a problem on normal submission because of a legitimate absence on the class which is officially recognized (*e.g.,* participating in a competition or attending a conference), **send an e-mail with attaching the certificates for absence to chief TA Sunjae Kim (sjkim94@korea.ac.kr) before the due date.** In that case, please let your e-mail subject follow the format of "[COSE342-TP] <student-id>-<name>" (*e.g.,* [COSE342-TP] 2021320999-홍길동).