COM SCI 118 Spring 2019 Computer Network Fundamentals

Project 1: Web Server Implementation using BSD Sockets

704966744
Keyu Ji

704978214
Hermmy Wang

Introduction

In this project, we have written a Web server in C. The goal of this project is to develop a Web server that is able to dump the HTTP request messages to the console, transmits binary files of arbitrary sizes and shows an HTML page and other files correctly in the browser. The overall design is based on the Unix socket framework.

High-level description of the web server's design

We aim to develop a web server based on the TCP server socket programming. In order to make a TCP server, we refer to the following steps (https://www.tutorialspoint.com/unix_sockets/socket_server_example.htm) :first, we need to call `socket()` to create a socket file descriptor. Then we ask the user to provide a port number and convert the input string to an actual number. Then, we bind the host address with the socket file descriptor using the `bind()` system call. Now start listen for the clients. The maximum number of connections allowed is 5. In an infinite while loop, we continuously accept new socket file descriptors from the client. Fork the process Sending and receiving data between server and client is done by `read()` and `write()` system calls.

The client can initiate a file request through a URL. The server parses the input HTTP request header, figures out the file name, and attempts to open the file if it exists in the server's directory. In the case it does exist, the server sends the binary data back to the client and the file should be properly displayed on the client-side browser.

Difficulties and solutions

1. Insufficient buffer size
   At first, we tried to use `getchar()` to recognize the end of a HTTP request message instead of a fix-sized buffer. However, we soon realized that `getchar()` only reads from standard input instead of the socket file descriptor. Therefore, we choose a buffer size 1024 bytes to hold the HTTP request.

2. Case insensitivity
   Instead of directly using `fopen()` or `open()`, case_insensitive matching of file names requires iterating through all the files in the directory, ignoring the case. We adopt a method referred by https://www.experts-exchange.com/questions/21360429/fopen-make-filename-case-INSENSITIVE-take-both-Lower-and-Upper-Letters.html, which contains a directory entry structure that iterates all the files in a given directory.

3. ERROR on binding: Address already in use
   If the server was running with a specific port number and then was shut down, it is likely that the same port number will not be available during the following seconds. If we try to run the server

with the same port number, we get the error “Address already in use”. The time out range is about 20s~1min.

4. “favicon.ico”
   The server actually prints out two HTTP request headers. One for the user-requested file, the other one for “favicon.ico”. This icon can be found in the address bar to represent the web page. Therefore, the client requests both this icon and the specified file. Since the spec does not have a requirement for “favicon.ico”, we choose to ignore any implementation on it.

5. Testing with diff
   After we have received a binary file at the client side, we download the content to the local directory.
   `$ diff [path to the file in the server] [path to the file downloaded at the client]`

Manual for compiling and running the web server

1. Compiling

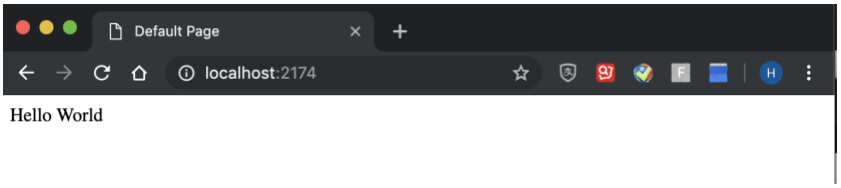
```
$ make
gcc -g3 -o server server.c

$ ./server [port number]
[cursor blinking...]

[^C] to exit
```

2. Running

(1) Print out the correct HTTP request message to the console

While the server is running, open Mozilla Firefox.
In the address bar, type in URL:
`http://localhost:[port number]`

You should be able to see the default HTML page on the client-side browser:



You should be able to see the received HTTP request header on the console:

```
wifi-131-179-34-105:proj1 hermmy$ ./server 2174
GET / HTTP/1.1
Host: localhost:2174
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (
KHTML, like Gecko) Chrome/73.0.3683.103 Safari/537.36
DNT: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/a
png,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,zh-CN;q=0.8,zh;q=0.7
```

(2) Transmit a small binary file

While the server is running, open Mozilla Firefox.
In the address bar, type in URL:

```
http://localhost:[port number]/[file_name]
```

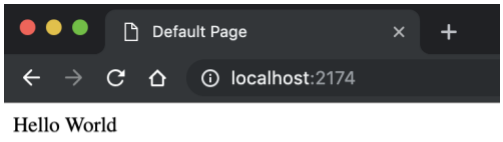
(3) Transmit a larger binary file

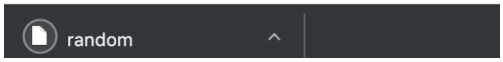
We have generated a larger random file (100MB) with

```
$ head -c 100000000 </dev/urandom >random
```

While the server is running, open Mozilla Firefox.
In the address bar, type in URL:

```
http://localhost:2174/Random
http://localhost:2174/random
```

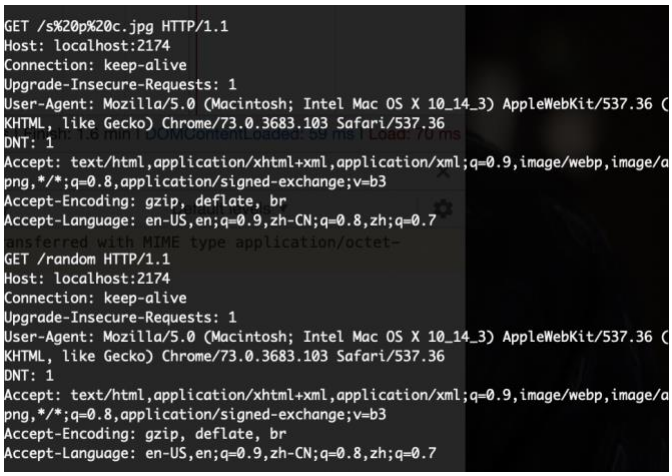




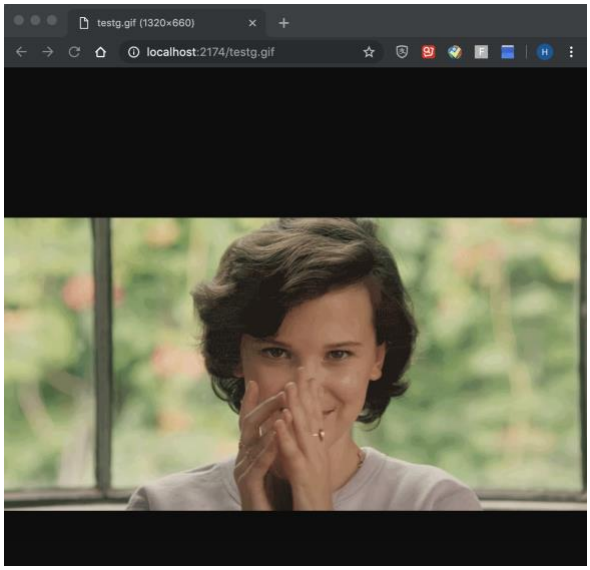
[Prompt to downloading]

Sample outputs

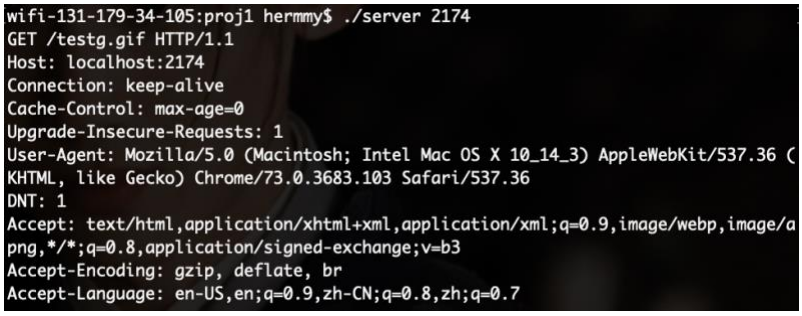
1. Part A: print out the received HTTP request header
   While the server is running, open Mozilla Firefox.
   In the address bar, type in URL:
   `http://localhost:2174/s%20p%20c.jpg`
   `http://localhost:2174/random`



2. Part B: successfully transmits binary file to the client
   While the server is running, open Mozilla Firefox.
   In the address bar, type in URL:
   `http://localhost:2174/testg.gif`



   Server receives HTTP request header:



Client response message:

`Inspect -> Network -> Headers -> Response Headers`