CS 118 Project 1 Report：HTTP Web Server

Haiying Huang 804757410
Xilai Zhang 804796478

1. Project Description

In this project, we developed a web server that handles HTTP request from clients using socket programming in C language. Once a TCP connection to the client is established, our server will read the HTTP request, create a response message with appropriate headers and file data, and send the response to the client. The server will also dump the request message to the console. We have tested the server on the ubuntu virtual machine app over windows 10 and initiated web clients using Chrome browser on the same machine. For example, we can enter the following URL into the browser to request an HTML file from the local machine via port 2048:

`localhost:2048/ucla.html`

The client will receive content of the requested file and be able to display the HTML page on the browser.

2. Implementation and Difficulties

2.1. Implementation

This project can be divided into two major parts: 1) setting up the server and 2) handling HTTP requests. To set up the server, we create a socket, bind it to the host address and the specified port number, and put it into passive mode to wait for client connections. For simplicity purpose, our server handles client connections sequentially, one request at a time. As shown in the code snippet below, the server issues the *accept()* system call to extract the first pending connection from the queue, reads the request message from the socket, dumps the message to the *stdout*, and handles this request before moving on the next pending connection.

```
while (1)
{
        // skeleton of the main loop
        new_sockfd = accept(server_sockfd, ...);
        ...
        int nread;
        char buf[MAXBYTES];
        nread = read(new_sockfd, buf, MAXBYTES);
        ...
        write(STDOUT_FILENO, buf, nread);
        handle_request(new_sockfd, buf);
        close(new_sockfd);
}
```

To handle the HTTP request, we parse the headers from the request message, such as file name and HTTP version. Then, we search the current working directory for matched entries. If the requested file exists on the server, we will retrieve its information such as file-type, last-modified-time, and size, using the *stat()* syscall and generate a corresponding response message. For the message body, we will send the binary content of the file. If the requested does not exist on the server or we detect some error, we will send an HTTP response with error status code according to the RFC 1945 standards.

2.2. Difficulties

One major challenge we met is to search for the requested file. According to the specification, filename must be handles in case-insensitive fashion, and multiple file extensions must be supported. Moreover, file names might contain spaces. To implement the correct behavior, we write a function to test the equality of two strings by converting both strings to lowercases. To handle file extension, we split the string by delimiter dot, retrieve the extension, and look up the corresponding file-type, such as text/html or image/pdf. To handle spaces, we adopt the standard that spaces in URL would be mapped to '%20'. Therefore, we will replace all occurrences of substring '%20' into spaces before we compare the two file names.

Another major problem is debugging. When we test the first version of our servers, we notice that in addition to the content of requested file, some headers lines such as last-modified-time might also appear in the browser. It took us very long time to debug and detect the cause of this problem. When we generate the HTTP response message, we use *ctime()* syscall to make a formatted string of the current date and append '\r\n' to mark the end of the Date header. However, *ctime()* automatically appends a newline character to the date string so that the '\r\n' becomes an empty line, which is used to mark the end of the message headers. Therefore, subsequent headers, such as file-type, are also treated as the message body by the client. Our solution is to strip the last character from the date string returned by *ctime()*.

3. Compilation and Usage

We will briefly explain how to compile and run our program on a Linux machine. We have also included a simple Makefile for this purpose. First, you need to go to the directory containing our source files and issues *make* command, which will compile our source code using gcc.

```
[haiying@lnxsrv09 ~/cs118/project1]$ make
gcc -Wall -Wextra webserver.c -o webserver
```

To launch the server, issue command *./webserver portno* in the console. We have not hardcoded the port number so you need to specify an available port (e.g. 2048) as command line argument.

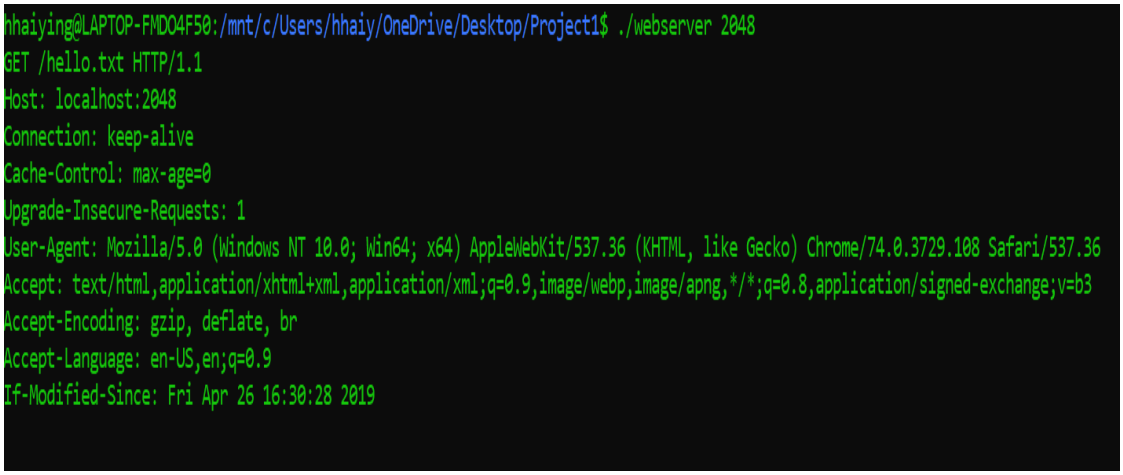
```
[haiying@lnxsrv09 ~/cs118/project1]$ ./webserver 2048
```

Once the server is on-host, you can use a browser such as Firefox as clients and send HTTP

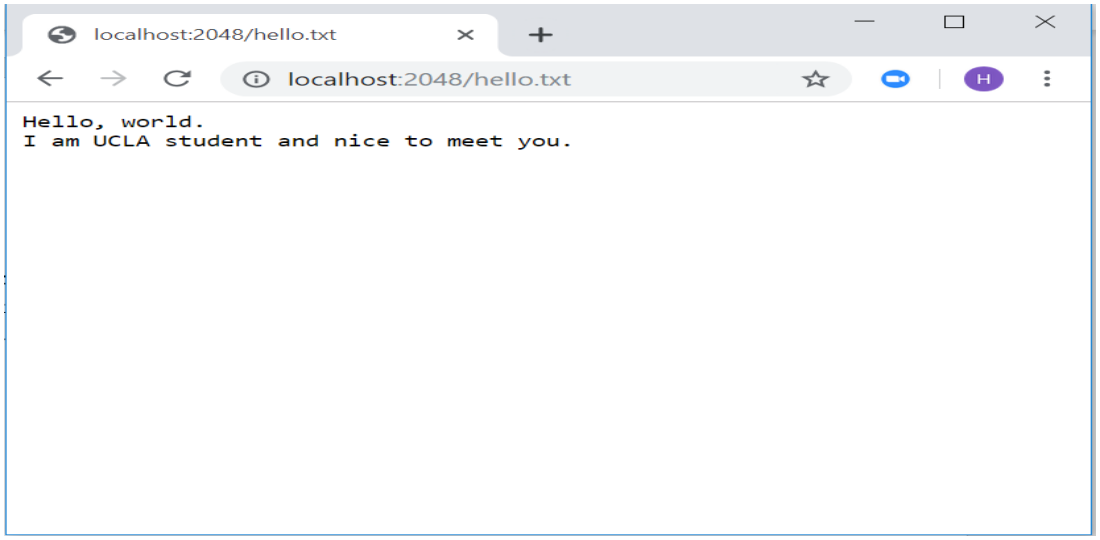
requests to our server as URL address *http://localhost:port/filename*. Notice that the port number must be consistent.

4. Sample Experiments

Now, we will demonstrate the outputs of some simple client-server experimental. Our server is launched on ubuntu virtual machine app over windows 10, and we use Google Chrome browser as our clients. First, we set up the server on port 2048 and send an HTTP request for a small text file hello.txt. Screenshots for the console output and browser are included below.



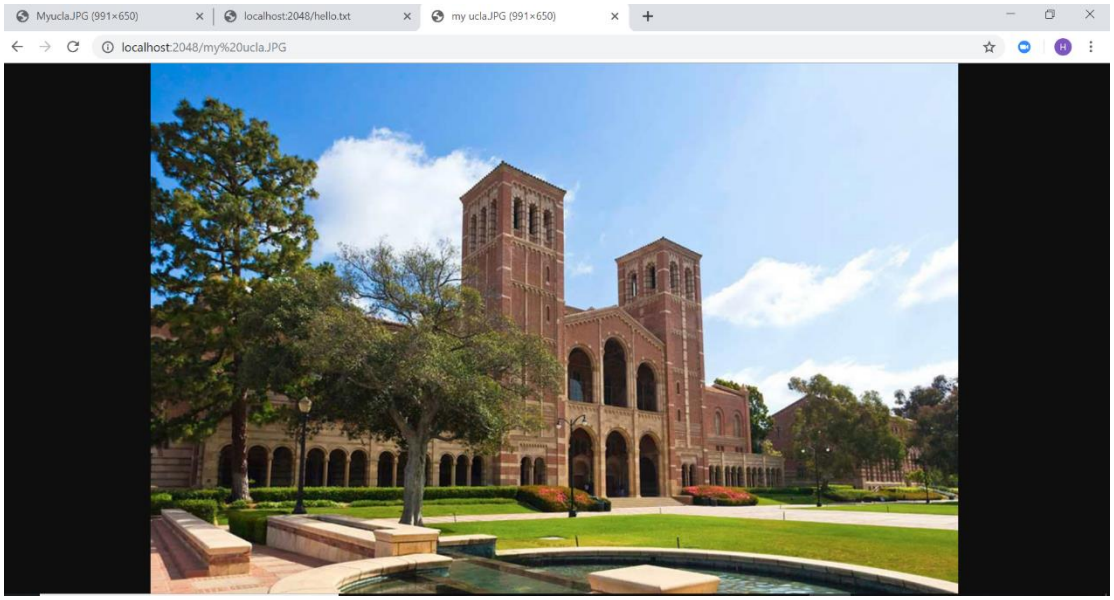
The request message has been echoed to *stdout*. As we can see, a GET command is issued to request file hello.txt and the client hopes to use persistent HTTP connection. Now, let us look at the browser.



Success! The content of hello.txt has been displayed, which suggests that data of the requested file has been transmitted and correctly decoded.

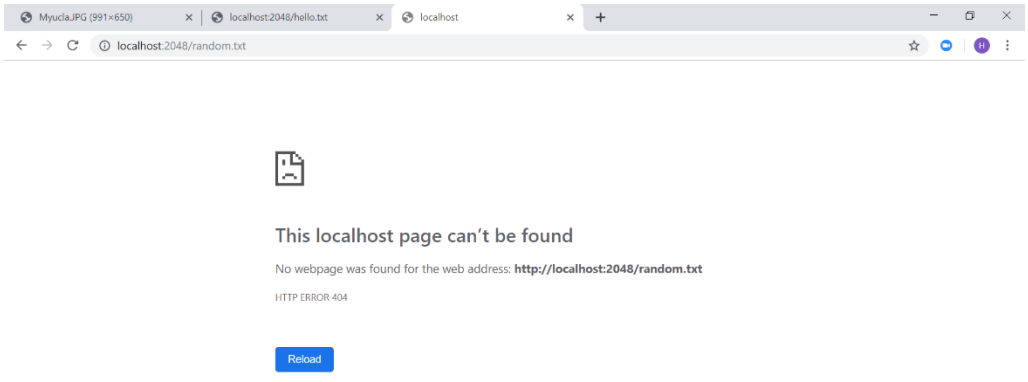
Next, we request a large image file and test the behavior of server to compare filenames with spaces in case-insensitive way. The file on the server side has name “my UCLA.jpg” but we enter “MY ucla.JPG” in URL address. As we can see, the browser has automatically replaced the space by “%20” in the request message.

```
GET /my%20ucla.JPG HTTP/1.1
Host: localhost:2048
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.108 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
```



Success! The image file has been displayed, which suggests that our server correctly handles spaces in file name and compares file name in case-insensitive way.

Finally, let us try to request a file “random.txt” that does not exist on the server.



The 404 error code is displayed, which suggests that our server correctly handles files that do not exist on the server.

5. Conclusion

In this project, we implemented a web server that handles HTTP request and serves the requested file to clients. Our server implements the RFC 1945 standard and supports multiple file types. We have tested our server on ubuntu virtual machine and included some sample output in this report.