COM SCI 118 Spring 2018
Computer Networking Fundamentals

Project 1: Web Server Implementation using BSD Sockets

Xiaopei Zhang(004309991), Hengyu Lou(005035476)

April 27, 2018

Abstract

In this project, we implemented a "Web Server" following the TCP establishment, connection and transmission procedures using the socket programming APIs provided by the C language. From this, we better understand the advantages as well as the limitations of the client-server model.

1 Overview

We used the demo client-server code provided in the specification as the skeleton code for our own "Web Server". Similar to the demo code, we require the users to run the executable with an input argument specifying which port to connect to the server. However, we also ask the users for a second input argument specifying which part they are interested in. For part A, our "Web Server" only dumps the HTTP request message to the console. And for part B, it dumps both the HTTP request message and the HTTP respond message (i.e, the HTTP header), and sends the requested file to the client if it exists on the server.

We used Mozilla Firefox as the client browser, which could connect to the server via a predefined port number and request a file. On the other end, to behave properly, the server should start before the client requests, bind to a socket, and listen for any incoming connection requests. Then the server should accept one request, parse the request message, and send the requested file back to the client if the file exists.

We declared and defined five helper functions, which are **parseRequest**, **findMatchingFile**, **getContentType**, **attachHeader** and **respondRequest**, to make our lives easier. Here in this report, we will only briefly introduce their major functionalities, and please refer to the well-commented code in "server.c" for more details.

- **parseRequest**: Deal with spaces and letter cases of the request message.
- **findMatchingFile**: Return the actual name of the requested file on the server.
- **getContentType**: Return the content type of the requested file.

- **attachHeader**: Print out the HTTP respond message header.
- **respondRequest**: Transmit the requested file to the client.

2 Difficulties and Solutions

2.1 Spaces

By printing out the HTTP request message, we knew that if the users' request contains spaces, each of them would be replaced by one "%20". But since we still need to preserve the spaces to find the requested file on the server, we should convert "%20"s back to spaces before passing the file name to our file seeker. This functionality is encapsulated in the **parseRequest** function.

Upon calling the **parseRequest** function, our server computes the file name length with all "%20"s replaced by spaces, allocates the right size of memory using the **malloc** function, and fills the modified file name in.

2.2 Letter Cases

To tackle the case-sensitive issues, in the **parseRequest** function, we made our server convert the name of the users' requested file into lowercase as soon as it receives that. For example, if a user types "http://localhost:10080/TeSt.TxT" in the address bar, our server will parse "test.txt" out of it.

Another special design is in the **findMatchingFile** function. To find the requested file on the server, we used the **strcasecmp** function to compare the processed users' request (i.e, the lowercase file name) with the name of each file in the server's main directory. The **strcasecmp** function overlooks the arguments' cases so that we do not have to compare two file names character-wise.

2.3 File Extensions

File extensions can be very tricky. Initially when we implemented the **getContentType** function, we used the **strstr** function to find whether certain file extensions, such as ".html", ".gif" and ".txt", exist in the users' request. Later on, we realized that some files might be named like they have more than one extension, say "text.html.txt". Essentially this is a text file, but our program will recognize it as an HTML page and send back to the client browser with a wrong content type. Therefore, we used the **strrchr** function in <string.h> to detect only the last occurrence of "." in the users' request, and then determine the file extension.

3 Compiling and Running Instructions

To compile and run our program, the users should follow the steps in the linux terminal:

1. Change to the directory where the file "server.c" and the Makefile locate.
2. Type "make" in the command-line in order to generate the executable.

3. Run the executable with command: ./server arg1 arg2.

4. Open the client browser and type "http://localhost:<port>/<filename>" in the address bar.

In step 3, "arg1" stands for an unreserved port number to connect to the server and "arg2" specifies which part of the assignment (either part A or part B) the users are interested in. As in step 4, the users should replace the <port> field with the actual port number they provided in step 3, and replace the <filename> field with the requested file's name.

4 Sample Outputs

Suppose we have the following files in the main directory of our server:

- test: A binary data file.
- TesT.HtMl: An HTML page file.
- TEST.html.txt: A plain text file.
- t e s t.JPG: An image file.

In total, we run 6 tests to verify that our server behaves correctly and robustly. We only presents what the users type into the address bar, but remember we should always start the server prior to any tests.

1. *http://localhost:10080/test.gif*: File does not exist on the server. (Figure 1)
2. *http://localhost:10080/Test*: Transmit "test" to the client. (Figure 2)
3. *http://localhost:10080/test.jpg*: File does not exist on the server. (Figure 3)
4. *http://localhost:10080/test.html*: Display "TesT.HtMl" on the browser. (Figure 4)
5. *http://localhost:10080/test.html.txt*: Display "TEST.html.txt" on the browser. (Figure 5).
6. *http://localhost:10080/T E S T.JPG*: Display "t e s t.JPG" on the browser. (Figure 6).

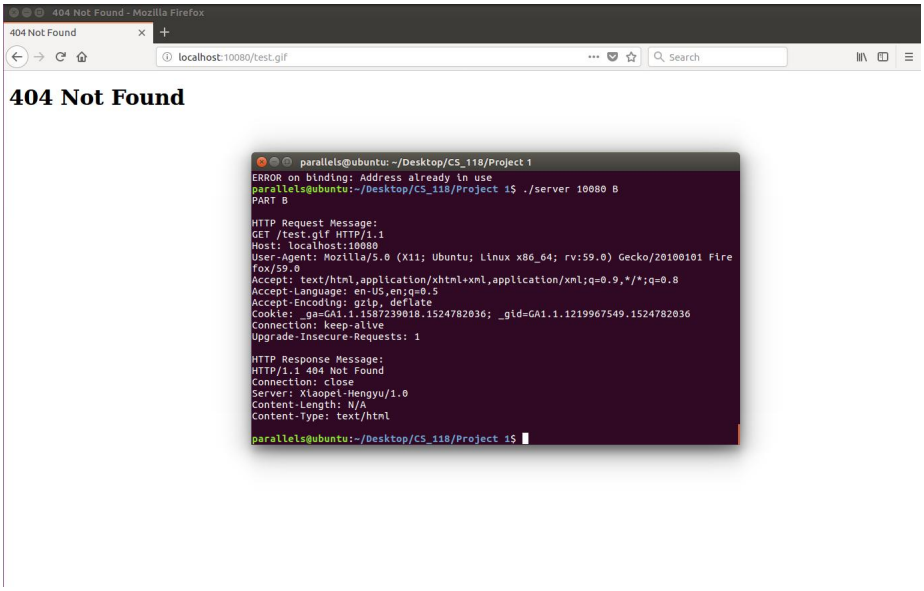

Figure 1: Output of *http://localhost:10080/test.gif*.

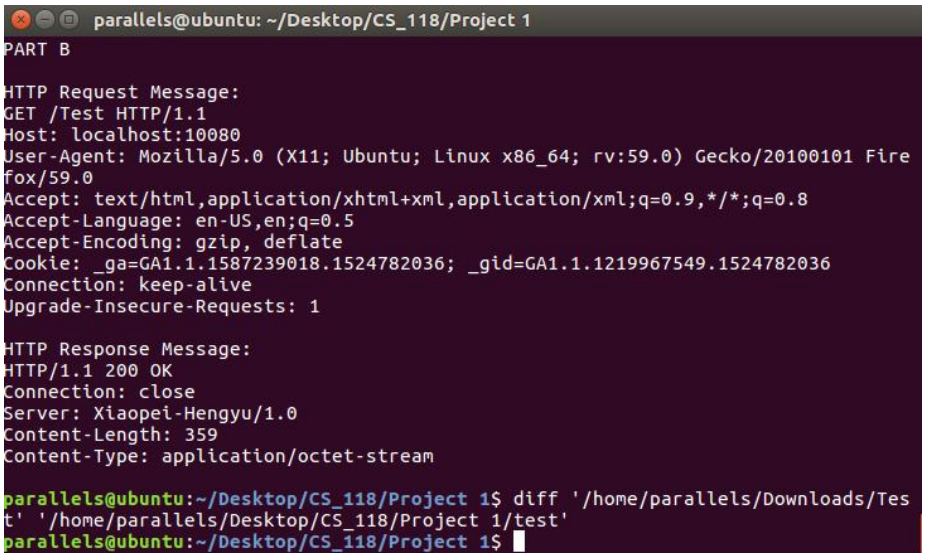

Figure 2: Output of *http://localhost:10080/Test*.

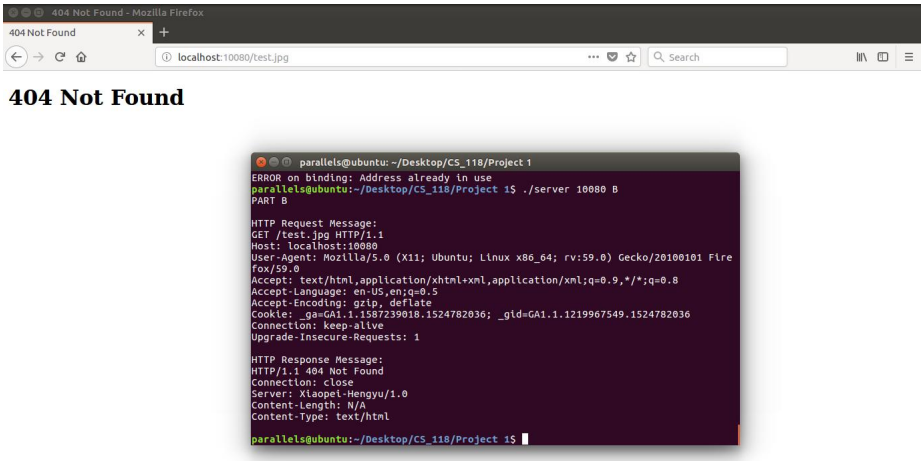

Figure 3: Output of *http://localhost:10080/test.jpg*.

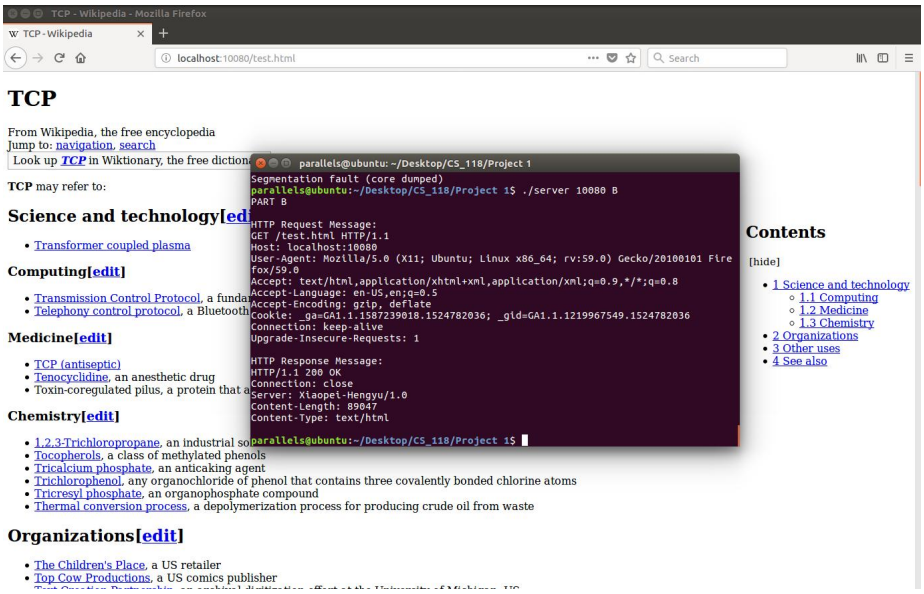

Figure 4: Output of *http://localhost:10080/test.html*.

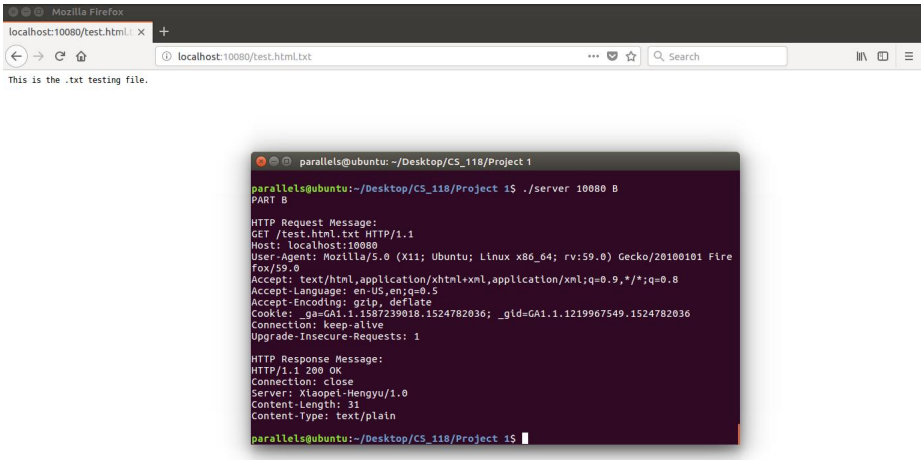

Figure 5: Output of *http://localhost:10080/test.html.txt*.

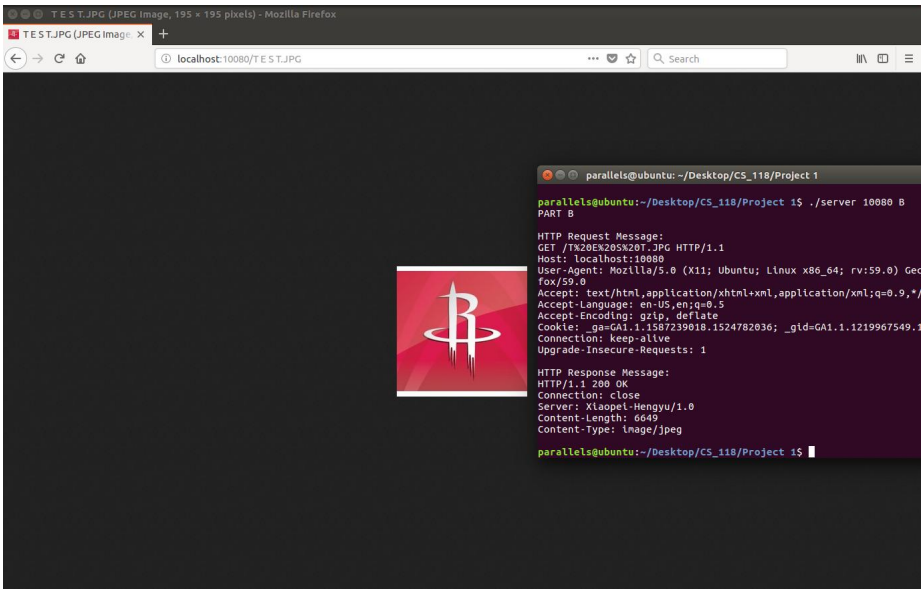

Figure 6: Output of *http://localhost:10080/T E S T.JPG*.