CS536 Lab1: Build Your Own Web Server over Sockets

Due: 23:59:59 PM, Feb 8, 2023
Total points: 50 points

Updated on Jan 28, 2023.
Please refer to this document if any conflict with previous clarifications at PSOs or Campuswire.

1 Goal

This is a programming lab. In this lab, you are going to develop client-server socket programming and build a concurrent Web server in C. Also, we will take the chance to learn a little bit more about how Web browser and web server work behind the scene.

2 Part A: Build Client-Server over Socket Programming (10 points)

- Set up server and client over Socket programming. The sample code and tutorial of socket programming are provided in the assignment (lab1.zip).

  Please compile the sample codes (using gcc or write a MAKEFILE) and generate two executive file `server` and client. Please open two terminals: one to run server and the other to run client. Basically, the server runs a server process over TCP while each client creates a client process to connect to this server process. Please run the server first. After that, please run a client and you will be prompted to input a message and then the message will be sent to the server after clicking "Enter". Upon receiving the message, the server will send the message back, which will be displayed on the client terminal. The server will show the message on its terminal, too.

- (5 points) You need to revise the provided sample codes to allow the client and the server to run on different machines which can be specified by the IP address and port number in the command line. You should run the code using the following commands:
  ./server [port#]
  ./client [ip_server_addr] [port#] [message]

  Please run the server first. The client will send [message] to the server and wait for the response message from the server. After the response message is received, the client will close the connection (after a while, explained next). The server will print out the received messages on its terminal in the following format and send the same message back to the client as the response message:
  message-from-client: [client_ip], [client_port]
  [message]
  where [client_ip] and [client_port] are the IP address and Port Number used by the client. Please start with a new line to display the received message, i.e., [message].

  The server will print out the following information on its terminal upon receiving the client's request to close the connection:
  close-client: [client_ip], [client_port]

TIP: the client program can choose to terminate the connection right after the message is received at this step. But we strongly suggest that the client program does not terminate the connection right after one message is received; Instead, it should stay connected longer for a while (say, 60s), which will makes it easier for you to run the next step to support multiple clients and the following parts to build a web client. You can use `while` or `for` loop with `sleep()` so that the client connection will last as least for 60 seconds before the client program terminate the TCP connection after the message is received.

You can begin by running both of server and client at the `localhost`. Namely, use `localhost` as [ip_server_addr]. After it works, you need to finally make the client and server run on two different machines at HAAS G050. Below are the machine names:

amber01.cs.purdue.edu
amber02.cs.purdue.edu
...
amber30.cs.purdue.edu

- (5 points) You need to revise the code to support multiple clients on the server using `Pthreads` and save it as `serverMul.c`. You can test the new server with at least two clients running on different machines or one machine with two different terminals.

  You should run the code using the following commands:
  ./serverMul [port#]
  ./client [ip_server_addr] [port#] [message1]
  ./client [ip_server_addr] [port#] [message2]

  We expect the server can print message1 and message2 on its terminal before two clients close their connections. Note you need to run the second client before the first client terminates its connection.

3 Part B: Develop Your Web Server over HTTP/1.1(25 points)

Please read Chapter 2 of the textbook and the lecture slides carefully. Please play with the in-class demo to see how the existing websites (web servers) respond to various GET messages. All these will help you to understand how HTTP and HTTP/1.1 works. You need to pick a port number for your web server. Instead of using port 80 or 8080 or 6789 for the listening socket, you should pick your own to avoid conflicts. Please do not use any port number between 0 and 1024.

This simple web server needs to implement the following features:

- When the client sends GET to request for one .html which exists, it should respond "*200 OK* "and return this .html file.
- When the client sends GET to request for one .html which **doesn't** exist, it should send the response "*404 Not found* ".
- When the client sends GET with the wrong format (e.g. URL String Syntax Error), it should send the response "*400 Bad Request* ".
- When the client sends GET with a different HTTP version, it should send the response "*505 HTTP Version Not Supported* ".

The server expects to print the following information on its terminal. Upon receiving a GET request from a client, the server will print out:

message-from-client: [client_ip], [client_port]
[firstline-of-request]

Upon sending a response message to a client, the server will print out:
message-to-client: [client_ip], [client_port]
[firstline-of-response]

[firstline-of-request] or [firstline-of-response] are the first line of the request/response, e.g, `GET /index.php HTTP/1.1` (firstline-of-request), HTTP /1.1 200 OK (firstline-of-response).

For the purpose of debugging, you are allowed to print more information on the terminal but please do not change the above printout (for example, no extra print out between the line of message-from-client: [client_ip], [client_port] and the line of [firstline-of-request].

Please save your server code as `server1.c`. Please make your server code support multiple clients at the same time as you did in part A.

Before you run the server code, please create a folder named `www` in the folder containing your server program and place all the web objects in the `www` folder. Include the relative path the `www` folder in your server's code and support the URL with and without `www`. That is, if you enter a URL `http://server/file:port\#` to get a webpage http://server/file, the server converts its URL into a local path as `./www/file`.

Several webpage files are provided in the zipped file to test with your web server. Of course, please feel free to create and use your own test webpage files and copy them into the `www` folder.

- text.html: a html file which contains text only.
- picture.html: a html file which contains text and a small picture (< 200KB).
- bigpicture.html a html file which contains text and a big picture (> 1MB).

In this Part, we will use a web browser to test your server code and it is not necessary to revise the client code. During your development, you can follow our in-class demo to send the GET messages to test and debug your codes. But you will find that the client code must be revised in Part C and it would be good to revise the client code in Part B (explained at the end of this part).

You need to test your server with a web browser:

- Please run your server. You should run the code using the following commands:
  ./server1 [port#] In each test case, please copy the messages displayed at the server's terminal into your lab report (please only copy the messages required and specified in this lab).

- (15 points, 5 points per one test html file: text.html, picture.html, bigpicture.html). Connect to your server from a web browser with the URL of http://<machine name>:<port number>/<html file name> and see if it works. For example, you can try `http://localhost:8888/text.html` or `http://server-ip-address:8888/text.html` if your web server uses the port number of 8888 and has a file called text.html in its root folder (www) for all the web contents. Please make sure the server should work with the existing browser (Chrome, Firefox, etc) to get the test webpages.

- (5 points) Test with other HTTP responses (404, 400, and 505). For example, you fetch a file which does not exist in the www folder.

- (5 points) Run the above tests with two or more web browsers.

Do I have to revise the client code in Part B? You do not have to make any change in the client's code to get full credit for PART B, we only test your server1. However, for Part C, you should implement a client program that supports HTTP/2.0. (Otherwise, you have to implement more in server2.c in order to test your HTTP/2 server using a web browser). That is, we will use client2.c (a naive client program) to test your HTTP/2 server in Part C. HTTP/2.0 requires a HTTP persistent connection mechanism allowing the

client to send/receive multiple request/responses to get multiple objects in a single TCP connection. So you might find that it is better to revise the client code to support a HTTP persistent connection and save your client code as `client1.c`. In particular, you should run the code using the following commands:
./client1 [URL] (namely, ./client1 [http://server-ip:port#/path])
Here, [URL] is the same as the one used by a web browser, http://server-ip:port#/path The client code must extract the server IP address, port# and path. The client will first fetch the base HTML file for the given URL and parse the received HTML file to check if there are more web objects to get. If yes, the client will send multiple GET requests in the same TCP connections and wait for the responses. Please use a timer (say, at least 60 seconds) to postpone the closure of this TCP connection even after all the objects have been received. Please save the code as `client1.c`.

4 Part C: Develop Your Web Server over HTTP/2.0 (15 points)

Please learn how HTTP/2.0 works to GET multiple objects in one webpage. Please modify the code of the client and server to support HTTP/2.0. Save your code as `client2.c` and `server2.c`. Please do remember to change HTTP/1.1 into HTTP/2 in all the request and response messages.

In your HTTP/2 implementation, please fix the transfer frame size to **40KB**. The client should print out the object and frame information in its terminal in the following format:
Object-Frame: [Object] [Frame_#]

Here, [Object] uses its URL or a relative path. Please print the results every 100 frames, that is, Frame_1, Frame_101, Frame_201, $\cdots$. In HTTP/2, you should see received frames of different objects intersecting with each other. For example,
Object-Frame: [Object1] Frame_1
Object-Frame: [Object2] Frame_1
Object-Frame: [Object3] Frame_1
Object-Frame: [Object1] Frame_101
Object-Frame: [Object2] Frame_101
Object-Frame: [Object1] Frame_201
Object-Frame: [Object1] Frame_301
...

Please test your programs with video.html, and compare the page loading process using HTTP/2.0 with the one using HTTP/1.1.

- (10 points) Deploy your server and your client both at localhost and compare how to GET video.html via HTTP/2.0 and HTTP/1.1. You should run the HTTP/2 code using the following commands:
  ./server2 [port#]
  ./client [URL] (e.g., http://localhost:8888/video.html)

  Please run HTTP/1.1 and HTTP/2.2 once at a time. Please show the order of objects and the time needed on the console. Please copy the print on the console at the client side into your lab report. Please summarize the main difference between HTTP/1.1 and HTPP/2.0 in the report.

- (5 points) Please run HTTP/1.1 and HTTP/2 server at one host (using different port numbers). Please deploy your clients at a different host and run the command to GET video.html almost at the same time. Do you see any changes from the above scenario with the server and clients at the same host? Please briefly explain why. Please include the print on the client console and answer the questions in your lab report.

Tip: Please start with small objects (5MB) during the development and debugging. You can replace video.html with your own test webpage with multiple objects. There are more than three objects. One huge object is larger than 12MB and a small object is smaller than 40KB.

5 Materials to turn in

You will submit your assignment at gradescope. You submission should zip all the files into “lab1_PUID.zip”, including the following files:

1. All your source code files (`client.c`, `serverMul.c`, `client1.c`, `server1.c`, `client2.c`, `server2.c`).
    - Please add appropriate comments to your code and make it easy to understand.
2. Your project report called `lab1.*` (txt, pdf or word format). In the report,
    - Please include your name and student ID at the top of the first page.
    - Include a brief manual (readme) in the report to explain how to compile and run your source code (if the grader can’t compile and run your source code by reading your manual, the project is considered not to be finished).
    - Include the answers and results as specified above.
    - Describe what difficulties you faced and how you solved them, if applicable.
3. Please DO NOT INCLUDE any file under “www” folder OR ANY .html files IN YOU SUBMISSION. You will lose at least 5 points without following the submission guideline.
4. Never submit .rar or other types of compressed file. You must submit .zip file. Otherwise, you will lose some points.

6 More Tips and Support

1. Please do not change the file names, otherwise you will lose some points.
2. For all the parts, you will need to support variable ports. The ports are given in the command, and you should not hard code it for your final submission version.
3. If connecting from off-campus, you will first need to connect to the Purdue VPN -https://webvpn.purdue.edu/ and then access the Linux machines.
4. If you want to use any late day, please send an email to cs536-ta@cs.purdue.edu and tell us.

More questions about the assignment should be posted on Campuswire or asked during PSOs.