# Web Server

Implement a simple HTTP web server and host your personal web page with the server. The web server should be able to respond to the page requests initiated from a web browser (such as Chrome or Firefox) as a client. The server serves pages stored locally (i.e., on the same computer in which the server is running), including your personal web page.

## 1. Objectives

- To further improve programing skills in C.
- To learn how to handle network connections in Linux.
- To gain deeper understanding on the client-server model and how servers function.
- To gain deeper understanding on HTTP protocol.
- To learn HMTL and get familiar with the tags in HTML.

## 2. Overview

You are to do this project BY YOURSELF.

The web server you implement will be similar to, but much simpler than, normal web servers, such as Apache or IIS. It only responses GET requests with static content (i.e., files that can be delivered to browsers without having to be generated, modified, or processed). It does not respond to other requests (e.g., POST). But, it needs to respond correctly to the invalid requests from browsers (e.g., requests for non-existent content) by generating corresponding messages (e.g., "HTTP 404 - Not Found" error message). It also needs to support both non-persistent connections and persistent connections.

The program should accept a port number when it is run, and should print out a usage message (https://en.wikipedia.org/wiki/Usage_message) to instruct the user to do so if the port number is not given in the command line. Suppose the executable file of your program is `webserver`. The command line to run the program is like the following

```
./server 13895
```

In the command line above, 13895 is a port number.

Your web server needs to

- accepts connections (TCP) from web browsers;
- reads in the packets that the browsers sends;
- detects the end of a request by checking the content in the packets;
- parses the request to determine the type of HTTP connection (persistent vs. non-persistent) and the content to be served;
- prepares and sends a response to the web browser. If the request is invalid, the response should contain the corresponding error message. When the response is received, the browser should be able to display the content or the error message on the screen.

Refer to the lecture notes on network programming for the functions that you can use in your program for network communication with web browsers. Refer to the lecture notes on web and

HTTP to have some basic ideas on how a web server interact with a web browser using HTTP protocol. To get more information, you may also refer to some online articles introducing the HTTP protocol. Since your program needs to parse HTTP requests and generate HTTP responses, you need to clearly know how raw HTTP requests and responses look like. Web-sniffer at http://web-sniffer.net/ is a useful tool for you to check the formats of raw HTTP requests and responses.

Create a personal web page. Save the html page as **index.html**. You should include at least the following contents on the page:

- Your name
- A photo of yourself in jpeg format. The photo needs to be saved in a file with .jpg extension.
- UCID
- NJIT ID number
- Major
- A photo of anything else (e.g., a pet, a guitar, a computer, or your garden, etc). The photo needs to be saved as a PNG picture with .png extension.

You need to organize all the contents above on the same page. So, when you open the page (i.e., index.html) with a browser, you can see all these contents.

You may also include other contents on the page or other pages pointed by the hyperlinks in index.html. But this is not required and will not be tested by the grader.

## 3. Programming Instructions

When the program is run, it creates a socket to listen for incoming TCP connections based on the port specified by the command line. Then, in a loop, it accepts new connections. Refer to the example server code in the slides for network programming part. You should use a backlog of 10 in `listen()` function call. Make sure that your program checks the error in the `bind()` call. The program should print out an error message when this call fails.

Your web server should be multi-threaded. For each new connection, it creates a worker thread for the connection. The worker thread is to receive request(s) from the connection, parse the request(s), generate response(s), and send the response(s) back to the client through the connection. Since a new socket is created for the connection (returned from accept() call), the worker thread needs to know the file descriptor for the socket, in order to talk to the client.

Most of the work is done in worker threads. Thus, most of your code is actually in the function(s) for the worker threads.

### 3.1. Receiving a request

A worker thread needs to first read the entire HTTP request header. An HTTP request header ends with the four characters `\r\n\r\n`. Read the slides for the format of HTTP request

headers. You may also use web_sniffer to get the raw HTTP request headers. A sample request header for Chrome browser is obtained from web_sniffer and included below:

```
GET /~dingxn/index.html HTTP/1.1[CRLF]
Host: web.njit.edu[CRLF]
Connection: close[CRLF]
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; de-de)
AppleWebKit/523.10.3 (KHTML, like Gecko) Version/3.0.4
Safari/523.10[CRLF]
Accept-Encoding: gzip[CRLF]
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7[CRLF]
Cache-Control: no-cache[CRLF]
Accept-Language: de,en;q=0.7,en-us;q=0.3[CRLF]
Referer: http://web-sniffer.net/[CRLF]
[CRLF]
```

In the slides and the header above, `CR` is `'\r'` and `LF` is `'\n'`. Note that different requests may have different header fields (e.g., different numbers of fields and different header values). There is not a fixed length for the request headers. Your program should not assume that the entire header can be read with one call to recv() or read(). You need to keep receiving/reading data from the connection until you see `\r\n\r\n`. Your program needs to read the header into a buffer so that it can analyze the header later. The requests are usually short. Thus, it is safe for you to use a buffer of 2KB (you need to reuse the buffer for different requests).

### 3.2. Parsing a request

Since your web server only responses to GET requests, which do not have request bodies. The end of a request header is also the end of the request itself. Thus, when your program sees `\r\n\r\n`, it can start parsing. It needs to extract the following information from the header: 1) the type of request (e.g, GET in the example above, POST, or HEAD) and the URL of an object (e.g., a HTML page, an image file) from the first line of the header; and 2) the `Connection` field and the value in the field. If the `Connection` field is not included in the request header, your program should treat the request as if the following line was included in the header:

```
Connection: close[CRLF]
```

### 3.3.Responding to a request

### a) Generating and sending response header

To generate a response, your program should first determine the status. In your program, the status may be one of the following:

- HTTP 501 – Not Implemented:

If a request is not well-formatted and the required information cannot be extracted (e.g., type of request cannot be recognized or URL is missing), this response should be sent back to notify the client that the request cannot be handled by the server.

• HTTP 404 --- Not Found:

When the object being requested is not found or cannot be opened by your server, this response is returned to the client.

• HTTP 200 --- OK:

The object being requested is found on the machine running the server. The object is sent back to the client in the body of the response.

To determine whether it is a HTTP 404 response or a HTTP 200 response, you need to first locate the object. Note that the URL part in the request is not a complete pathname. So you cannot directly locate the object using the URL. Some translation is needed. First, all the files served by the server are saved in the `web` directory of your current directory or its subdirectories. The URL is relative to the `web` directory. Thus, if the URL is `/index.html`, it is to get file `web/index.html`; and if the URL is `/images/ppt.jpg`, it is to get file `web/images/ppt.jpg`. Second, if the URL is ended with a `'/'`, it is to get the `index.html` page under the corresponding directory. Thus, if the URL is `/`, it is to get file `web/index.html`; and if the URL is `/myfolder/`, it is to get file `web/myfolder/index.html`.

You need to get the corresponding pathname based on URL before you can locate the file. If the file does not exist, the status should be HTTP 404. If the file exists, try to open it. If the open call fails (e.g., due to permission reasons), the status should also be HTTP 404. If the open call succeeds, the status should be HTTP 200.

Depending on the status, your program include appropriate status code and status phrase in the status line of the response header. Refer to the HTTP response message part in the lecture slides.

You may use the following constants as the status phrases in the status lines:

```
const char *HTTP_200_STRING = "OK";
const char *HTTP_404_STRING = "Not Found";
const char *HTTP_501_STRING = "Not Implemented";
```

In addition to the status line, the response header must include the following header lines in each response:

• **`Connection`**

If the request contains a `Connection` field and the value of the field matches `Keep-Alive` (case insensitive, use `strcasecmp()` instead of `strcmp()`), you must include a `Connection: Keep-Alive` line in your response header. If the request packet does not contain `Connection: Keep-Alive` (either because `Connection` is not in the request or because the value is not `Keep-Alive`), you must include a `Connection: close` line in your response header.

• **`Content-Length`**
This is the length of the request body. For a HTTP 404 response or a HTTP 501 response, it is the string length of the `HTTP_404_CONTENT` or `HTTP_501_CONTENT` constant below prepared for the response body. For HTTP 200 response, it is the length of the file.
• **`Content-Type`**
Your program needs to determine the content-type of the response. For HTTP 404 and HTTP 501 requests, the content type will always be `text/html`. For HTTP 200 requests, you need to examine the file name and find the extension. The content type is determined by the extension. For example, if the URL is `/index.html`, the extension is `html`. Thus, the content type is `text/html`. If the URL is `/photo.jpg`, the extension is `jpg`. Thus, the content type is `image/jpeg`. You may use the following constants in your program for the Content-Type line.

```
/* html or htm extension*/
char * chtml="Content-Type: text/html\r\n";
/* css extension */
char * ccss="Content-Type: text/css\r\n";
/* jpg extension */
char * cjpg="Content-Type: image/jpeg\r\n";
/* png extension */
char * cpng="Content-Type: image/png\r\n";
/* gif extension */
char * cgif="Content-Type: image/gif\r\n";
/* Any other extensions */
char * cplain="Content-Type: text/plain\r\n";
```

Note that, just like the request, every line in the HTTP header must be ended with `\r\n`. Before the response body, another two characters `\r\n` are needed to mark the end of the header. (There are four characters `\r\n\r\n`, taking into count the `\r\n` at the end of the last line of the header.)

Your program may put the header and `"\r\n"` into a buffer and send the content in the buffer in a send() call or write() call. Response headers are usually short. Thus, it is safe for you to use a buffer of 2KB. (You need to reuse the buffer for different responses).

**b) Sending response body**

Your program needs to send the response body after it has sent the response header and `"\r\n"`. For HTTP 404 and HTTP 501 responses, you may use the following constant strings (`HTTP_404_CONTENT` and `HTTP_501_CONTENT`) as response bodies.

```
const char *HTTP_404_CONTENT = "<html><head><title>404 Not Found</title></head><body><h1>404 Not Found</h1>The requested resource could not be found but may be available again in the future.<div style=\"color: #eeeeee; font-size: 8pt;\">Actually, it probably won't ever be available unless this is showing up because of a bug in your program. :(</div></html>";
```

```
const char *HTTP_501_CONTENT = "<html><head><title>501 Not Implemented</title></head><body><h1>501 Not Implemented</h1>The server either does not recognise the request method, or it lacks the ability to fulfill the request.</body></html>";
```

For HTTP 200 responses, the response body should be the entire content of the corresponding file.

### 3.4.Maintaining and closing the connection

If the request has a `Connection` field in its header and the value is `Keep-Alive`, your program needs to keep the connection to receive a new request (3.1) and respond to the request (3.2 and 3.3). So you need to put the code corresponding to 3.1~3.3 in a loop.

Otherwise, your program needs to finish the loop and close the connection (call `close()` against the socket file descriptor).

A worker thread finishes after it closes the connection.

## 4. Testing Instructions

You may want to test your program first on your local machine. But, make sure that you spend enough time on compiling and then testing your program on one of the *afs* servers (*afsN.njit.edu*, where *N* is an integer from 1 to 36), because our grader will also test your program on an afs server. To facilitate grading, your program should compile with *gcc* and without specifying the *-std* option in the *gcc* command line (on *afs* servers, *gnu90* is the default version for C standard).

### 4.1. Running your server

You can run your webserver program on either your local machine (e.g., a virtual machine) or an afs server. The program should be run with a port number above 1023 and below 60000.

```
./webserver 37864
```

To terminate the web server, you can press ctrl-c, or kill the server process using the kill command. It would be nice to handle the SIGINT signal by releasing the port. When your program terminates, you may find that the port may not be freed for up to a minute. Thus, you may need to change the port number if you restart your server right after terminating it.

Since ports are shared globally in a computer, when you run your program on an afs server, you need to select a port number that is not currently used by another user. You may use command `netstat -an --tcp | awk '/LISTEN/ {sub(".*:", "", $4); print $4}' | sort -nu` to check which ports are being used and select an unused one. The other way is to select a random port number. In your program, the `bind()` call will fail if someone else is already using that port. Thus, your program needs to print out an error message when this

happens. When you see the message, wait a minute and then try again. If `bind()` still fails, choose another random port number.

### 4.2.Running the client

There are three ways as follows to run the client (i.e., the web browser).

**a) You run the web server and the browser on the same machine.**

You can simply use the following URL in the address bar of the browser. Note that in the URLs below, 37864 is the port number that the web server is listening. You need to replace it if you use another port number:

http://localhost:37864/index.html

If you see a webpage, your program successfully served an HTTP request! We will not be grading anything on the command line output of your web server. So feel free to use stdout and stderr for any debugging or status messages you'd like.

**b) You run the web server on an afs server and the browser on another machine.**

The machine running the browser needs to have a NJIT IP address. If you are not on campus, you need to use VPN, in order to connect to your web server. You may use the following URL in your web browser. Still, you need to replace afs15 with the actual host name of the afs server you are using, and replace 37864 with the actual port number your web server is listening.

http://afs15.njit.edu:37864/index.html

**c) Using telnet and lwp-request as web client.**

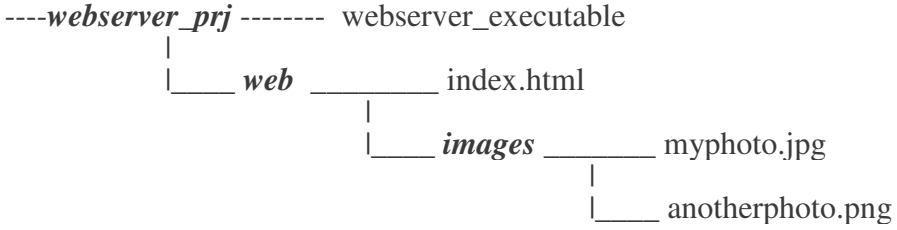
If nothing seems to be working, you may find telnet and another command-line tool lwp-request useful. Refer the lecture slides for the web server and HTTP part for how to use telnet to talk to a server. You can find the manual of lwp-request here (http://search.cpan.org/dist/libwww-perl/bin/lwp-request). You can use these tools to send requests to the server and show the responses. With these tools, you can control what fields should be included in the request headers. This can be helpful for testing and debugging the part parsing request headers. These tools also show responses for your to examine. Note that, for text-based files (e.g., html pages), you can let the tool show the responses on the screen; for binary files (e.g., jpeg images), you may want to redirect the output to a file.

The tool can also show both headers and bodies for both requests and responses. This makes the tool particularly useful for debugging.

lwp-request -U -e http://localhost:37864/index.html

### 4.3. Hosting web site

You need to put your personal web page and other files (e.g. photos) associated to the web page under a directory named ***web***. Then, you put the ***web*** directory and the executable of your web server under the same directory. You may decide whether you organize the files associated to the web page into the subdirectory of ***web***. But the html page must be named `index.html`, and `index.html` must be put into ***web*** directory, not any of its subdirectory. The following diagram shows an example organization of these files. Note that the names in bold font are folders.




We also provide the pages of a web site for you to test your program. You can download multi.zip from Moodle, which contains the web pages of a web site and other supporting files. Make sure that the files are saved in the ***web*** directory or a subdirectory of ***web*** before you accessing the pages.

## 5. Submission Instruction

1. You should include all your code for the web server in a single .c file. To ensure that we compile your C programs correctly, name your program in the pattern SECTION#_NJITID#_1.c. SECTION# is the three-digit section number of the CS288 section you registered (e.g., 001, 101, don't miss the leading 0s). NJITID# is the eight-digit NJIT ID (Not your UCID, Rutgers students also have NJIT IDs). So your file name is something like 001_00123456_1.c (DO NOT COPY THIS AS YOUR FILE NAME!).
2. Include the .c file and the ***web*** directory containing *index.html*, photos, and other objects for your personal web page into the same directory. Rename the directory ***webserver_prj***.
3. Packing the directory ***webserver_prj*** into a zip file ***webserver_prj.zip***. You can use zip command to create the zip file under Linux OS (don't forget the –r option).
4. Submit ***webserver_prj.zip***.
5. Do not include any executable files or shell scripts in the zip file.

**It's highly recommended that you download your zip file from Moodle, transfer it to an afs server, unzip it, and repeat the tests with the files from your zip file. Finalize the submission only after you have finished the tests with the files from the zip file.**

## 6. Grading

**Test 1 (10 points)**: The executable file of the web server can be successfully generated with *gcc* on an *afs* server.

It is not realistic for us to try different option combinations to get your programs to pass the compilation. We will only use *gcc* command lines without specifying the *-std* option. Make sure that the following commands can successfully compile your programs on an *afs* server.

```
gcc –o webserver  your_web_server_program.c
```

**Test 2 (5 points)**: The executable file of the web server can print an informative error message if the port number specified is not available.

Use the following command to test whether it can print out the error message:

```
./webserver `netstat –an –-tcp | awk '/LISTEN/ {sub(".*:", "", $4); print $4}' | sort –nu |tail –1`
```

**Test 3 (5 points)**: Your web server can be successfully launched with an unoccupied port number provided in the command line. It does not terminate or crash within 1 minute without any requests sent to it.

Refer to **4.1** on how to find an unoccupied port number to launch your web server. Do not send any requests to it. Your web server should not terminate or crash within 1 minute.

**Test 4 (10 points)**: Your personal web page (those files contained in the ***web*** directory packed in the zip file) can be displayed with the latest chrome browser and contain the required information.

Save the ***web*** directory on your local machine. Use a browser to open the index.html file in the directory. Your personal web page should show the required information (and other information you want to include).

**Test 5 (25 points)**: Your web server can successfully respond the GET requests for the index.html file that you created as your personal web page.

Suppose you run your web server on the afs15 server and the port number is 37864. On another machine with a NJIT IP (on campus or connected to NJIT VPN), use the following commands to get index.html.

```
lwp-request http://afs15.njit.edu:37864/index.html > /tmp/index1.html
lwp-request http://afs15.njit.edu:37864/ > /tmp/index2.html
```

File index1.html should be identical to the index.html you submitted in the zip file (20 points).
File index2.html should be identical to the index.html you submitted in the zip file (5 points).

**Test 6 (10 points)**: When a client requests a non-existent page, your web server can send back a HTTP 404 response to the client.

If you use lwp-request to fetch a non-existent page, lwp-request should be able to show the same content as that in the aforementioned constant `HTTP_404_CONTENT`.

```
lwp-request http://afs15.njit.edu:37864/not_a_page.html>./404page
```

Compare the content in file *404page* against the content in `HTTP_404_CONTENT`.

If you use a browser to fetch a non-existent page, the browser should be able to show a "**404 Not Found"** page.

**Test 7 (10 points)**: When a client sends an ill-formed request to your server, your web server can send back a HTTP 501 response to the client.

If you use lwp-request to send an ill-formed request, lwp-request should be able to show the same content as that in the aforementioned constant `HTTP_501_CONTENT.`

```
lwp-request http://afs15.njit.edu:37864/not...valid.html>./501page
```

Compare the content in file *501page* against the content in `HTTP_501_CONTENT`.

If you use a browser and the URL in the address bar is ill-formed (e.g., `http://afs15.njit.edu:37864/not...valid.html`), the browser should be able to show a "**501 Not Implemented" page.**

**Test 8 (15 points)**: A web browser can successfully connect to your web server and display your personal web page.

Refer to **4.2** for how to test the web server with a browser. You get partial credit if the web browser only show a part of the components in your web page --- 5 points for your photo (.jpg file), 5 points for another photo (.png file), and 5 points for all the other components.

**Test 9 (10 points)**: A web browser can successfully connect to your web server and display the pages from *multi.zip* downloaded from Moodle.

Unzip the files in *multi.zip* into ***web/multi*** directory (i.e., subdirectory ***multi*** under the ***web*** directory). A browser can display the *index.html* page using this URL (4 points)

```
http://afs15.njit.edu:37864/multi/index.html
```

or this URL (3 points)

```
http://afs15.njit.edu:37864/multi/index.html
```

Following the hyperlinks in the index.html page, the browser can show other pages in the website, including a blog page, a portfolio page, and a services page (3 points).

==========

Total: 100 points