

# Lab 4: HTTP Web Client and Server (15 Points)

## 1 Download Lab

Download the Lab 4 files from Brightspace. The source code will be inside the directory “Lab4/”. You must use the environment from Lab 0 to run and test your code. Next, open a terminal and “cd” into the “Lab4/” directory. Now you are ready to run the lab!

## 2 HTTP Web Page Downloader (4 Points)

There is a very useful program called “wget”. It is a command line tool that you can use to download a web page like this:

```
$ wget http://www.gnu.org/software/make/manual/make.html
```

This will download the make manual page, “make.html”, and save it in the current directory. “wget” can do much more (e.g., downloading a whole web site). See the manual for “wget” for more info.

For the first part of this lab, your task is to write a limited version of “wget”, which we will call “http\_client”, that can download a *single* file. You will do all your implementation inside the file “http\_client.c” inside the “http\_client” directory. To build and run the code, do the following:

First, go inside the “http\_client” directory,

```
$ cd http_client
```

Next, compile the code,

```
$ make
```

Finally, run the code,

```
$ ./http_client [host] [port number] [filepath]
```

For example, `./http_client www.gnu.org 80 /software/make/manual/make.html`

**You must build, run, and test your code on ecegrid using the environment from Lab 0. If your code does not run in that environment, you will not get any credit!**

In the above command, you give the components of the URL separately in the command line — (1) the server host, (2) the server port number (which will always be 80 for HTTP), and (3) the file path. The program will download the given file and save it in the current directory. So in the example above, it should produce “make.html” in the current directory. It should overwrite an existing file.

### Some useful hints:

1. The program should open a TCP socket connection to the host and port number specified in the command line, and then request the given file using HTTP/1.x protocol.  
(See <http://www.jmarshall.com/easy/http/> for the details of HTTP/1.x protocol).
2. An HTTP GET request looks like this:

```
GET /path/file.html HTTP/1.0\r\n
[zero or more headers]\r\n
[blank line]\r\n
```

Include the following header in your request:

```
Host: <the_host_name_you_are_connecting_to>:<port_number>
```

3. The response from the web server will look something like this:

```
HTTP/1.0 200 OK\r\n
Date: Fri, 31 Dec 2020 23:59:59 GMT\r\n
Content-Type: text/html\r\n
Content-Length: 1354\r\n
[blank line]\r\n
[file content]
```

There might be slight variations in the formats of responses from different servers. Go through the document in Step 1 (in particular, [this section](#)) to ensure that your parser is robust.

The code "200" in the first line indicates that the request was successful. If it's not "200", the program should **print the first line of the response to the terminal (stdout) and exit**.

You will need to extract the file name from file path (for example, extract `make.html` from file path `/software/make/manual/make.html`), create a new file with extracted file name in the current directory, and write the received file content into that file.

You should use the "Content-Length" value to figure when to stop receiving data from the web server and close the TCP connection. If the "Content-Length" field is not present in the response header, print the following error message to the terminal (stdout) and exit:

```
Error: could not download the requested file (file length unknown)
```

4. Some useful C library functions for parsing—"strchr()", "strrchr()", "strtok()", "strstr()"
5. Your program should be able to download any type of file, not just HTML files. Test your code by downloading all the different files on the web site <http://www.gnu.org/software/make/manual/>.

Use "write()" or "fwrite()" to write to the file in byte chunks. This is important to make your solution work for all different file formats, e.g., non-ASCII files such as pdf and image files. Functions like "fprintf()" might not work! **To verify correctness, also download the file using "wget" and make sure that it exactly matches the file downloaded by your program.**

### 3 HTTP Web Server

For the second part of the lab, your task is to write a HTTP web server using sockets interface. This task has two sub-tasks as described in Sections 3.1 and 3.2 respectively. You will implement both the sub-tasks in the file "http\_server.c" inside the "http\_server" directory.

To build the code, run:

```
$ cd http_server
$ make
```

**You must build, run, and test your code for both Sections 3.1 and 3.2 on ecegrid using the environment from Lab 0. If your code does not run in that environment, you will not get any credit!**

#### 3.1 Task 1: Serving static contents (7 Points)

In this part, you will write a web server that serves static content. The top level directory (called the "web root") for your HTML files will be the "Webpage/" directory provided with the lab. The web server will only serve contents inside the web root directory. For testing, you can add/remove/modify contents inside web root. We can also do the same while evaluating your submission.

To start the web server, run:

```
$ ./http_server [SERVPORT] [DBPORT]
```

The “SERVPORT” argument specifies the port number on which the web server would run, and “DBPORT” argument specifies the port number on which the database server would run (used in Section 3.2).

### Choosing the SERVPORT and DBPORT values:

To avoid port number clashes between different lab groups running their code at the same time on the same ecegrid machine, please follow the following convention for choosing the port numbers:

1. Choose “SERVPORT” value as **8000 + Group Number**

So, if your group number is 38, choose “SERVPORT” value 8038

If your group has two members, and both of you want to run the code at the same time, then choose the “SERVPORT” values as **8000 + Group Number** and **9000 + Group Number** respectively.

2. Choose “DBPORT” value as **53000 + Group Number**

So, if your group number is 38, choose “DBPORT” value 53038

If your group has two members, and both of you want to run the code at the same time, then choose the “DBPORT” values as **53000 + Group Number** and **54000 + Group Number** respectively.

The content served by the web server should be accessible through a web browser running on the ecegrid machine by typing the following URL (assuming the web server is running on port 8888):

```
http://localhost:8888/path/to/content/relative/to/web/root
```

For example, URL `http://localhost:8888/index.html` should display file “Webpage/index.html”.

Writing a web server is not a trivial task. Here is the list of what is expected and what is not expected from your web server:

1. The web server will be **iterative**, i.e., it will serve client requests one request at a time. The server should close the TCP socket (returned by the “accept()” call) after serving each request. In practice, most web servers are **concurrent**, i.e., they could serve multiple client requests in parallel using multithreading or multiprocessing (e.g., using “fork()”).
2. The web server will only support the GET method. If a browser sends other methods (POST, HEAD, PUT, for example), the server responds with status code 501. Here is a possible response:

```
HTTP/1.0 501 Not Implemented\r\n
[blank line]\r\n
<html><body><h1>501 Not Implemented</h1></body></html>
```

Note that server adds a little HTML body for the status code and the message. Without this, the browser will display a blank page. This should be done for **all** status codes except 200.

3. Our server will be strictly HTTP/1.0 server. That is, all responses will say “HTTP/1.0”, and all successful responses will include status code “200 OK”.

The server will accept GET requests that are either HTTP/1.0 or HTTP/1.1 (most browsers these days send HTTP/1.1 requests). But it will always respond with HTTP/1.0. The server should reject any other protocol and/or version, responding with 501 status code.

4. The server should also check that the request URI (the part that comes after GET) starts with “/”. If not, it should respond with “400 Bad Request”.

5. In addition, the server should make sure that the request URI does not contain `"/../"` and it does not end with `"/.."` because allowing `"/.."` in the request URI is a big security risk—the client will be able to fetch a file outside the web root. If true, respond with `"400 Bad Request"`.

**Note:** Most modern browsers automatically check for bad URL requests in points 4 and 5, and appropriately format the URL before sending it to the server. So, to test points 4 and 5, you can use your `"http_client"` instead of the browser (e.g., `./http_client localhost 8888 /../`).

6. The server must log each request to terminal (stdout) like this:

```
128.59.22.109 "GET /index.html HTTP/1.1" 200 OK
```

It should show the client IP address, the entire request line, and the status code and reason phrase that the server just sent to the browser (Figure 1).

```
vagrant@ubuntu:/vagrant/assignment4-sol/http_server$ ./http_server
10.0.2.2 "GET / HTTP/1.1" 200 OK
10.0.2.2 "GET /javascripts/scale.fix.js HTTP/1.1" 200 OK
10.0.2.2 "GET /pic.jpg HTTP/1.1" 200 OK
10.0.2.2 "GET / HTTP/1.1" 200 OK
10.0.2.2 "GET /pic.jpg HTTP/1.1" 200 OK
10.0.2.2 "GET /stylesheets/styles.css HTTP/1.1" 200 OK
10.0.2.2 "GET /stylesheets/pygment_trac.css HTTP/1.1" 200 OK
10.0.2.2 "GET /javascripts/scale.fix.js HTTP/1.1" 200 OK
10.0.2.2 "GET /stylesheets/styles.css HTTP/1.1" 200 OK
10.0.2.2 "GET /stylesheets/pygment_trac.css HTTP/1.1" 200 OK
10.0.2.2 "GET /?key=cute+cat HTTP/1.1" 200 OK
10.0.2.2 "GET /?key=fat+cat HTTP/1.1" 404 Not Found
10.0.2.2 "GET /favicon.ico HTTP/1.1" 404 Not Found
10.0.2.2 "GET /?key=cute+cat HTTP/1.1" 408 Request Timeout
10.0.2.2 "GET /factlist1.html HTTP/1.1" 200 OK
10.0.2.2 "GET /stylesheets/styles.css HTTP/1.1" 200 OK
10.0.2.2 "GET /stylesheets/pygment_trac.css HTTP/1.1" 200 OK
10.0.2.2 "GET /pic.jpg HTTP/1.1" 200 OK
10.0.2.2 "GET /javascripts/scale.fix.js HTTP/1.1" 200 OK
```

Figure 1: Sample terminal (stdout) output.

You must log the requests in the exact format as shown in Figure 1. You must **not** print anything else to the terminal. **Violations of these guidelines would result in a 10% grade penalty.**

7. If the request URI ends with `"/"`, the server should treat it as if there were `"index.html"` appended to it. For example, given

```
http://localhost:8888/
```

the server will act as if it had been given

```
http://localhost:8888/index.html
```

8. If the request URI is a directory, but does not have a `"/"` at the end, then you should append `"index.html"` to it.

Use `"stat()"` function to determine if a path is a directory or a file.

9. The server sends `"404 Not Found"` if it is unable to open the requested file.

10. For reading the file, use `"fread()"` or `"read()"`. You should read the file in chunks and send it to the client as you read each chunk. **Do not use `"strlen()"` to figure the length of buffer that you are sending to the client**—`"strlen()"` will stop counting as soon as it encounters the first NULL character. Instead use the return value from `"fread()"/"read()"`. The recommended chunk size is 4096 bytes, which is the optimal buffer size for disk I/O for many OS/hardware.

### 3.2 Task 2: Serving dynamic contents (4 Points)

In this part, you will add a database service to your web server. Web servers often have to contact a database to serve certain client requests. In this lab, clients can request a cat picture by entering a search string in the textbox displayed on the web page. On getting such a request, the web server will contact the database of cat pictures, and respond with the cat picture requested by the client.

To start the database server, run:

```
$ ./db_server [DBPORT]
```

Next, start the web server in a different terminal,

```
$ ./http_server [SERVPORT] [DBPORT]
```

The “SERVPORT” and “DBPORT” values must be chosen as described in Section 3.1.

#### Some useful hints:

1. If the current URL in your browser is `http://localhost:8888/`, and you enter the search string “cute cat” in the textbox and submit, the URL in the browser will now point to,  
`http://localhost:8888/?key=cute+cat`  
 and the web server will receive the request URI “/?key=cute+cat”. You should extract the search string “cute cat” from the URI and send it to the database server.
2. The web server will communicate with the database server over a UDP socket (“SOCK\_DGRAM”). The database server’s IP address and port number are defined in the macros “DBADDR” and “DBPORT” respectively inside the file “http\_server.c”.
3. On receiving a search string, the database server will append “.jpg” to the search string, and search for the file with that name inside the directory “cat\_database”. If found, the database server will send the file to the web server in UDP packets of size 4K bytes each. The web server should relay the data received in those UDP packets to the client over the TCP connection. Once the entire file has been sent, the database server will send a final UDP packet containing the string “DONE”. The web server should stop receiving once it receives this final packet. Note that the web server should not relay the contents of this final packet to the client.
4. If the file is not found in the database, the database server will respond with a UDP packet containing the string “File Not Found”. On receiving this packet, the web server should respond to the client with “404 Not Found”.
5. If the database server is not responding, the web server should not wait indefinitely for the response. Instead, it should timeout after some time interval (e.g., 5 seconds), and respond to the client with “408 Request Timeout”.

To test this functionality, simply do not start the database server. Then all client requests for cat pictures should timeout.

One way to implement timeout is to make the UDP socket non-blocking, and use the “select()” system call to determine when there is some data to be read. The “select()” system call has an argument of type “struct timeval”, which can be set to the timeout value. If no data is received by the socket within the timeout interval, “select()” will return 0.

6. Make sure that the logging to the terminal (Figure 1) that you implemented in Section 3.1 also works for cat picture requests, as illustrated in Figure 2.

```

10.0.2.2 "GET /?key=cute+cat HTTP/1.1" 200 OK
10.0.2.2 "GET /?key=fat+cat HTTP/1.1" 404 Not Found
10.0.2.2 "GET /favicon.ico HTTP/1.1" 404 Not Found
10.0.2.2 "GET /?key=cute+cat HTTP/1.1" 408 Request Timeout

```

Figure 2: Sample terminal (stdout) output for cat picture requests.

## 4 Frequently Asked Question

### 4.1 How to get rid of error “bind: address already in use” while running http\_server?

Wait for a few seconds, and it will go away. Or, run the following command:

```
kill $(lsof -i :8888 | awk 'NR>1 {print $2}')
```

## 5 Submission

You are required to submit two files “http\_client.c” and “http\_server.c” on Brightspace.

**Do not submit a compressed (e.g., .zip) file.**



## Sample Browser Outputs



