Lab 5: Building a HTTP Server

CS252

[Frequently asked questions](#)

[Your server should look like this: This is the http-root-dir as served by the CS HTTP Server.](#)

[Grading Form Part 1](#)

[Grading Form Part 2](#)

Purpose of the Lab

The objective of this lab is to implement a HTTP server that will allow a HTTP client (a web browser like FireFox or Internet Explorer) to connect to it and download files.

(又

协议

HTTP Protocol Overview

A HTTP client issues a `GET' request to a server in order to retrieve a file. The general syntax of such a request is given below :
GET <sp> <Document Requested> <sp> HTTP/1.0 <crlf>
{<Other Header Information> <crlf>}*
<crlf>
where :

- <sp> stands for a whitespace character and,
- <crlf> stands for a carriage return-linefeed pair. i.e. a carriage return (ascii character 13) followed by a linefeed (ascii character 10).
- `<crlf><crlf> is also represented as "\r\n\r\n".p`
- <Document Requested> gives us the name of the file requested by the client. As mentioned in the previous lab, this could be just a backslash (/) if the client is requesting the default file on the server.
- {<Other Header Information> <crlf>}* contains useful (but not critical) information sent by a client. Note that this part can be composed of several lines each separated by a <crlf>.
- `* - kleene star // regular expressions`

Finally, observe that the client ends the request with two **carriage return linefeed character** pair: <crlf><crlf>
The function of a HTTP server is to parse the above request from a client, identify the file being requested and send the file across to the client. However, before sending the actual document, the HTTP server must send a response header to the client. The following shows a typical response from a HTTP server when the requested file is found on the server:
HTTP/1.1 <sp> 200 <sp> Document <sp> follows <crlf>
Server: <sp> <Server-Type> <crlf>
Content-type: <sp> <Document-Type> <crlf>
{<Other Header Information> <crlf>}*
<crlf>
<Document Data>

sever reply

where :

- <Server-Type> identifies the manufacturer/version of the server. For this lab, you can set this to CS 252 lab5.
- <Document-Type> indicates to the client, the type of document being sent. This should be "text/html" for an html document, "image/gif" for a gif file, "text/plain" for plain text, etc.
- {<Other Header Information><crlf>}* as before, contains some additional useful header information for the client to use.
- <Document Data> is the actual document requested. Observe that this is separated from the response headers by two carriage return - line-feed pairs.

If the requested file cannot be found on the server, the server must send a response header indicating the error. The following shows a typical response:
HTTP/1.1 <sp> 404 File Not Found <crlf>
Server: <sp> <Server-Type> <crlf>
Content-type: <sp> <Document-Type> <crlf>
<crlf>
<Error Message>

文件不存在
sever reply

where :

- <Document-Type> indicates the type of document (i.e. error message in this case) being sent. Since you are going to send a plain text message, this should be set to text/plain.
- <Error Message> is a human readable description of the error in plain text/html format indicating the error (e.g. Could not find the specified URL. The server returned an error).

Procedure and Algorithm Details

Getting started

Login to a CS department machine (a lab machine or `data.cs.purdue.edu`), navigate to your preferred directory, and run

```
cd
cd cs252
git clone /homes/cs252/sourcecontrol/work/$USER/lab5-src.git
cd lab5-src
```

Then build the server by typing *make*. Run the server by typing *daytime-server* without arguments to get information about how to use the server. Run the server and read the sources to see how it is implemented. Some of the functionality of the HTTP server that you will implement is already available in this server.

Part 1:

Basic Server

You will implement an iterative HTTP server that implements the following basic algorithm:

- Open Passive Socket.
- Do Forever
    - ✓ Accept new TCP connection
    - Read request from TCP connection and parse it.
    - Frame the appropriate response header depending on whether the URL requested is found on the server or not.
    - Write the response header to TCP connection.
    - Write requested document document (initially you will respond with index.html that is found in htdocs/index.html) to TCP connection.
    - Close TCP connection

The server that you will implement at this stage will not be concurrent, i.e., it will not serve more than one client at a time (it queues the remaining requests while processing each request). You can base your implementation on the example server given in [1] . The server should work as specified in the overview above. Use daytime server as a reference for programming with sockets. Implement http server in "**myhttpd.cc**".

Basic HTTP Authentication

Background

In this part, you will add basic HTTP authentication to your server. Your HTTP server may have some bugs and may expose security problems. You don't want to expose this to the open Internet. One way to minimize this security risk is to implement basic HTTP authentication. You will implement the authentication scheme in RFC7617, aptly called Basic HTTP Authentication.

In Basic HTTP Authentication, you will check for an `Authorization` header field in all HTTP requests. If the `Authorization` header field isn't present, you respond with a status code of `401 Unauthorized` with the following additional field: `WWW-Authenticate: Basic realm="<something>"` (you should change `<something>` to a realm ID of your choosing such as "The_Great_Realm_of_CS252"). When your browser receives this response, it knows to prompt you for a username and password. Your browser will encode this in Base64 in the following format: `username:password` that gets supplied in the `Authorization` header field. Your browser will repeat the request with the `Authorization` header. You should create your own username/password combination and encode it using a Base64 encoder online (such as [this one]) or on a CS lab machine with `cat mycredentials.txt | base64`.

Client Requ1.1
est:

```
GET /index.html HTTP/
```

Server Response:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="myhttpd-cs252"
```

Client browser prompts for username/password. User supplies `cs252` as the username and `password` as the password and the client encodes in the format of cs252:password in base 64.

Client Request:

```
GET /index.html HTTP/1.1
Authorization: Basic Y3MyNTI6cGFzc3dvcmQ=
```

Note: you should create your own username and password and **NOT** use cs252 and password. You shouldn't use your Purdue career account credentials either.

You will check that the request includes the line

"Authorization: Basic <User-password in base 64>"

and then respond. If the request does not include this line you will return an error. This Line will be included in all the subsequent requests so you do not need to type user password again.

After you add the basic HTTP authentication, you may serve other documents besides index.html.

Resources
[RFC7617]

Adding Concurrency

You will also add concurrency to the server. You will implement three concurrency modes. The concurrency mode will be passed as argument. The concurrency modes you will implement are the following:

-f : Create a new process for each request
In this mode your HTTP server will fork a child process when a request arrives. The child process will process this request while the parent process will wait for another incoming request. You will also have to prevent the accumulation of inactive zombie processes. You can base your implementation on the server given in [3]

-t : Create a new thread for each request
In this mode your HTTP server will create a new thread to process each request that arrives. The thread will go away when the request is completed.

-p: Pool of threads
In this mode your server will put first the master socket in listen mode and then it will create a pool of 5 threads where each thread will execute a procedure that has a while loop running forever which calls accept() and dispatches the request. The idea is to have an iterative server running in each thread. Having multiple threads calling accept() at the same time will work but it creates some overhead under Solaris (See [4]). To avoid having multiple threads calling accept() at the same time, use a MUTEX lock around the accept() call.
If you want a review of threads see [Introduction to Threads.]
The format of the command should be:

```
myhttpd [-f|-t|-p]  [<port>]
```

If no flags are passed the server will be an iterative server like in the Basic Server section. **If <port> is not passed, you will choose your own default port number**. Make sure it is larger than **1024** and less than **65536**. ✓

MAKE SURE THAT THERE IS A HELP FUNCTION AND YOUR CODE IS INDENTED AND EASY TO READ (there will be points for this).

To turn in Part 1:

1. *Login to a CS department machine*

2. *Navigate to your `lab5-src` directory*
3. *Run `make clean`*
4. *Run `make` to check that your shell builds correctly*
5. *Run `git tag -f part1`*
6. *Run `git push -f origin part1`*
7. *Run `git show part1`*
8. *The `show` command should show the diff from the most recent commit*

Part 2

Browsing Directories

In this stage you will add to your server the capacity to browse directories. If the <Document Requested> in the request is a directory, your HTTP server should return an HTML document with hyperlinks to the contents of the directory. Also, you should be able to recursively browse subdirectories contained in this directory. An example of how a directory should look like is indicated in http-root-dir. Check the man pages for *opendir* and *readdir*.
Also implement sorting by name, size, and modification time.
MAKE SURE THAT THERE IS A HELP FUNCTION AND YOUR CODE IS INDENTED AND EASY TO READ (there will be points for this)

CGI-BIN

In this stage you will implement cgi-bin **.** When a request like this one arrives:
GET <sp> /cgi-bin/<script>?{<var>=<val>&}*{<var>=<val>}<sp> HTTP/1.1 <crlf>
{<Other Header Information> <crlf>}*
<crlf>
*运行脚本
the child process that is processing the request will call *execv* on the program in *cgi-bin/<script>*.
There are two ways the variable-value pairs in {<var>=<val>&}*{<var>=<val>} are passed to the cgi-bin script: the GET method and the POST method. You will implement the GET method and for **extra points** you may implement the POST method.
In the GET method the strfping of variables {<var>=<val>&}*{<var>=<val>} is passed to the <script> program as an environment variable QUERY_STRING. It is up to the <script> program to decode this string. Also if this string of variables exists, you should set the REQUEST_METHOD environment variable to "GET". The output of <script> will be sent back to the client.
In summary your cgi-bin implementation should:

1. Fork child process
2. Set the environment variable REQUEST_METHOD=GET
3. Set the environment variable QUERY_STRING=(arguments after ?)
4. Redirect output of child process to slave socket.

5. Print the following header:
6. HTTP/1.1 200 Document follows crlf
   Server: Server-Type crlf
7. Execute script

The script or cgi program will print the content type and will generate an output that is sent to the browser.
For more information on how cgi-bin works see [the Apache documentation.]()
Note. You will need to recompile the cgi-bin modules in lab5-src/http-root-dir/cgi-src

```
cd lab5-src/http-root-dir/cgi-src
grep getline *
----- Replace all the occurrences of "getline" to "mygetline"
make clean
make
```

Loadable Modules

In this stage you will implement loadable modules to be able to extend your server. When the name of a cgi-bin script ends with .so, instead of calling exec for this file your server will load that module into memory using *dlopen()*, if it has not been previously loaded. Then your server will transfer the control to this module by first looking up the function *extern "C" httprun(int ssock, char * query_string)* in that module using *dlsym()* and then calling *httprun()* passing the slave socket and the query string as parameters. *httprun()* will write the response to the ssock slave socket using the parameters in querystring.
For example, a request of the form:
http://localhost:8080/cgi-bin/hello.so?a=b
will make your server load the loadable module *hello.so* into memory and then call the function *httprun()* in this module with *ssock* and *querystring* as parameters. It is up to the module to write the response to *ssock*. Your server needs to keep track of what modules have been already loaded to not call *dlopen()* multiple times for the same module.
There is an example of how to use loadable modules in your lab5-src directory included in your git repository.
Also, in this part, you will need to rewrite the script http-root-dir/cgi-src/jj.c into a loadable module and name it jj-mod.c.
Hint: Use the call fdopen to be able to use buffered and formatter calls such as *fprintf()* to write to the slave socket. For example, in the top of *httprun() in jj-mod.c* call

```
  FILE * fssock = fdopen( ssock, "r+");
```

Then you can use the following to print to the slave socket:

```
  fprintf (fssock, "tomato, and mayo.<P>%c",LF);
```

Remember to close ffsock at the end of httprun().

```
 fclose( fssock);
```

MAKE SURE THAT THERE IS A HELP FUNCTION AND YOUR CODE IS INDENTED AND EASY TO READ (there will be points for this)

Implementing the Statistics and Log pages

You will implement a page http://localhost:<port>/stats with the following:

- The names of the student who wrote the project
- The time the server has been up line①
- The number of requests since the server started line 1
- The minimum service time and the URL request that took this time. line 2
- The maximum service time and the URL request that took this time. line 3

The service-time is the time it takes to service a request since the request is accepted until the socket is closed. Use the function timer_gettime to measure the duration of the requests and link your program with -lrt.

详见网页版

Also implement a page http://localhost:<port>/logs that will display a list of all the requests so far including in each line:

- The source host of the request ?
- The directory requested

The log will be stored into a file that will be preserved across runs.

Turning in your project

1. You will present your projects to your lab instructor during lab time.
2. Make sure that your server uses the http-root-dir and it loads by default the index.html from this directory. Test the simple, complex test, browsing directories and cgi-bin's. Your PSO instructors will use this directory during the presentation.
3. Write a short README file that includes:
   a) Features in the handout that you have implemented
   b) Features in the handout that you have not implemented
   c) Extra features

Include this file in your server's directory lab5-src/

Submission

Add a `README` file to the lab5-src/ directory with the following:

1. Features specified in the handout that work.
2. Features specified in the handout that do not work.
3. Extra features you have implemented.

To turn in Part 2:

1. *Login to a CS department machine*
2. *Navigate to your `lab5-src` directory*
3. *Run `make clean`*
4. *Run `make` to check that your shell builds correctly*
5. *Run `git tag -f part2`*
6. *Run `git push -f origin part2`*
7. *Run `git show part2`*
8. *The `show` command should show the diff from the most recent commit*

Grading

10% Checkpoint
90% Final presentation

Resources

lab5 slides part1
lab5 slides part2