CS 252: Systems Programming
Fall 2019
Lab 5: Web Server

Prof. Turkstra

Monday, November 11 11:58pm

1 Goals

In this lab, you will implement an HTTP server which allows an HTTP client (such as Mozilla Firefox or Google Chrome) to browse a website and receive different types of content. You will gain a high level understanding of Internet sockets, the HyperText Transfer Protocol (HTTP), SSL/TLS extensions to HTTP, and the Common Gateway Interface (CGI).

2 Deadlines

- The deadline for the checkpoint is **Monday, November 4 11:58pm**. No late submissions for the checkpoint will be accepted.
- The deadline for the final submission is **Monday, November 11 11:58pm**.

3 The Big Idea

You will implement relevant portions of the HTTP/1.1 specification (RFC 2616). Your server will not need to support any methods beyond GET, although there is extra credit available for supporting other methods (outlined in the extra credit section).

The lab template provides the basic foundation of your server in C and will allow you to focus more on the technical systems programming aspect of this lab, rather than needing to come up with a maintainable design for your server.

4 Getting Started

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

```
$ git clone ~cs252/repos/$USER/lab5
$ cd lab5
```

There is also a file example/daytime-server.c that has the example on sockets covered in class.

Build the server by typing `make`, and start it by running `./myhttpd <portno>` where `portno` is the port number you wish to bind to. You can see the various options that need to be implemented by running `./myhttpd -h`.

4.1 Security Notice

All CS machines are publicly accessible on the Internet. When developing a web server, it is wise to shield the server from any possible network traffic so that an attacker cannot take advantage of bugs in your server. **You should modify your server to bind to the loopback interface** for *at least* the first 3 tasks.

Upon doing this, if you are connected to data, your webserver will only accept connections coming from data itself. While all CS machines allow many users to be logged in simultaneously, it is a reasonably safe assumption that no malicious requests will be sent to your server. When you are comfortable that your server won't accidentally leak secret files, you can accept all connections from any interface so that you can develop via SSH and the web browser on your machine.

You can make your server bind to the loopback interface by modifying `src/tcp.c` to bind to `INADDR_LOOPBACK` instead of `INADDR_ANY`.

4.2 Resource Limits

In this lab, we set the limit for the amount of memory you use to 48 MiB, the maximum amount of CPU time to 5 minutes, and the maximum number of forked processes to be running to 50. We don't expect you to hit these limits, so if your server hits any of these limits, it's probably because you need to change the way you're doing something! Web servers like the one we are implementing should not be resource intensive.

5 The Assignment - Checkpoint

5.1 Background

HTTP, or Hypertext Transport Protocol, is a protocol for communicating over the Internet that is used by numerous applications. HTTP specifies the format and meaning of messages that are used to communicate between client applications and server applications. It is a textual format, and requests and responses can be thought of as structured blobs of text that humans can read. HTTP 1.0 follows a strict request-response model where the client opens a connection, sends a single request, and then receives a single response from the server before the server closes the connection and the transaction is complete.

Connections for HTTP are initiated using TCP, which is a high-level transmission protocol that has many features built in to handle dropped connections and faulty data transmission. You will not be implementing TCP in any capacity in this lab, it is merely the mechanism used to facilitate HTTP connections.

There are two types of HTTP messages: request messages and response messages. As you can imagine, a client sends a request message to an HTTP server to serve some request (e.g. the webpage `/index.html` or the picture `/img/PurdueSeal.png`). The request message contains details such as the version of HTTP being used, which resource is being requested, what you'd like to do with that resource, what sorts of responses the client accepts, and so on. The server is responsible for parsing the request message and creating a response message containing the results of the query.

5.2 Format

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> <URL> <sp> HTTP/1.1 <crlf>
(<headers> <crlf>)*
<crlf>
```

where:

- `<sp>` stands for a whitespace character
- `<crlf>` stands for a carriage return + line feed pair (ASCII character 13 followed by ASCII character 10).
- `<crlf><crlf>` is also represented as `\r\n\r\n`.
- `<URL>` gives us the name and location of the file requested by the client relative to a specified `DocumentRoot`. This could be just a forward slash (/) if the client is requesting the default index file on the server.
- `(<headers> <crlf>)*` contains additional information that can influence how the server behaves when responding. Note that this part can be composed of several lines each separated by a `<crlf>`.

Finally, observe that the client ends the request with two carriage return + line feed character pairs: `<crlf><crlf>`

The function of a HTTP server is to parse the above request from a client, identify the resource being requested, and send the resource to the client.

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 resource is found on the server:

```
HTTP/1.1 <sp> 200 <sp> OK <crlf>
Connection: <sp> close <crlf>
Content-Type: <sp> <document_type> <crlf>
Content-Length: <sp> <size> <crlf>
(<additional headers> <crlf>)*
<crlf>
<data>
```

where:

- `Connection: close` indicates that the connection will be closed upon completion of the response
- `<document_type>` indicates to the client the type of document being sent. Below are the document types that need to be implemented for this lab:
  - "text/plain" for plain text
  - "text/html" for HTML documents
  - "text/css" for CSS documents
  - "image/gif" for gif files
  - "image/png" for png files
  - "image/jpeg" for jpg/jpeg files
  - "image/svg+xml" for svg files
- `<size>` is the number of bytes that compose the delivered content
- `(<additional headers><crlf>)*` contains, as before, some additional useful information for the client to use.
- `<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>
Content-Type: <sp> <Document-Type> <crlf>
<crlf>
<Error Message>
```

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 format indicating the error (e.g. Could not find the specified URL. The server returned an error).

5.3 Basic Server

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

- Open a passive socket
- Do forever:
  - Accept a new TCP connection
  - Read a request from the TCP connection and parse it

  - Choose the appropriate response header depending on whether the URL requested is found on the server or not
  - Write the response header to the TCP connection
  - Write the requested document document (by default you should respond with index.html, located at htdocs/index.html) to the TCP connection
  - Close the TCP connection

The server that you will implement at this stage will not be concurrent, meaning that it will not serve more than one client at a time (it queues the remaining requests while processing each request). Use the aforementioned daytime server as a reference for programming with sockets. Implement your http server in "server.c", and "http_messages.c".

Note that in HTTP, all newlines should be "\r\n" (CRLF), not just "\n" (LF)!

- You should read RFC2616 Section 5 for information on how to parse request messages. The only method you are required to support is GET.
- You should also read RFC2616 Section 6 for information on how to form response messages.

5.4 Basic HTTP Authentication

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 should 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 and will supply them 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, which may be found online, or on a CS lab machine with the below command:

```
$ cat mycredentials.txt | base64
```

To illustrate this process, consider the following message sequence:

Client Request:

```
GET /index.html HTTP/1.1
```

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, which the client then encodes as cs252:password in base 64 (Y3MyNTI6cGFzc3dvcmQ).

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 can modify username and password in file auth.txt, and use function return_user_pwd_string (in server.c) to load it. When loaded, the string is stored in a global variable g_user_pass as well. 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.

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

5.5 Serving Static Files

A lot of web servers are really simple and have no dynamic content. These web servers use a directory hierarchy as their website structure. Your webserver will serve the http-root-dir/htdocs directory in the lab handout. When your web server gets a request such as `/index.html`, you will look for the file http-root-dir/htdocs/index.html and send that file. If you get a request for a file that doesn't exist, you should reply back with a Status-Code of 404. If you get a request for a directory, you should serve the index.html file in that directory (if that file is not present, then 404).

You do not need to respect the Accept field of the Request message. We do expect that you give a valid Content-Type in your response. You can call `get_content_type()` (defined in misc.h) to get this information for you. Be careful though! This function does not validate the filename and simply runs `file -biE <filename>` and gives you the output.

You will need to be capable of sending the following status codes: 200, 404. When grading for 404, we will only check that the Status-Code was set correctly. You can provide any content message you wish.

Some notes:

- You will mostly be modifying `server.c` and `htdocs.c` for this part.
- If you receive a request for a document that is a directory *without* a trailing frontslash (e.g. `/dir1`, not `/dir1/`, then you should really server `/dir1/index.html`. That is, pretend as though requests for `/dir1` are really requests for `/dir1/index.html`.
- If a request has a trailing frontslash, you will handle this in the browsable directories part.

5.6 Concurrency

In this part, you will add concurrency to the server. You will implement three concurrency modes, distinguished by an input argument passed to the server. The concurrency modes you will implement are the following:

- **-f : Create a new process for each request**

  Fork mode (`run_forking_server()`): You should handle each request in a new process. To do this, after you accept a connection off of your socket acceptor, you should call `fork()` and execute your normal request/response logic from within the child. Don't forget about zombies!

- **-t : Create a new thread for each request**

  Thread-per-request mode (`run_threaded_server()`): You should handle each request in a new thread. To do this, after you accept a connection off of your socket acceptor, you should create a new thread and execute your normal request/response logic from within the child. You should consider using pthreads for this part.

- **-pNUM_THREADS : Pool of threads**

  Pool of threads (`run_thread_pool_server()`): You should handle each request using a pool of n workers. n is denoted by NUM_THREADS. Your program should be using n+1 threads during execution - the thread that your program starts in should be used to create the n workers and then wait for them to finish. Of course, since each worker is running in an infinite loop, they will only finish on an error or when the program exits.

- **-h : Print usage**

  This flag will print out the usage of myhttpd and myhttpds.

The format of the command to run your server should be:

```
myhttpd [-f|-t|-pNUM_THREADS] [-h] <port>
```

If no flags are passed, the server should act like an iterative server as created in the Basic Server section. **If port is not passed, choose your own default port number.** Make sure it is **larger than 1024 and less than 65536.**

5.7 Turning in the Checkpoint

The deadline for the checkpoint is Monday, November 4th, 2019 at 11:58 PM You must run the following commands in order for your submission to be valid:

```
$ make clean
$ make myhttpd
$ make submit_checkpoint
```

6 The Assignment - Final

6.1 Secure HTTP (HTTPS)

HTTP messages are sent across the Internet unencrypted, which means that anyone who can sniff traffic along a message's route can see anything transmitted between the client and server (think about that Base64 encoded password from earlier...). This is often undesirable.

Fortunately a more secure version of HTTP exists, called HTTPS. HTTPS uses Transport Layer Security (TLS) to communicate. TLS encrypts all traffic at the transport layer of the Internet model, which means that you, as an application programmer, only need to worry about adding high-level support for communicating via TLS. That is, the underlying socket has changed, but the HTTP protocol you've implemented is unconcerned about what the socket does when your server asks to read or write data.

Fortunately, we have an abstraction of tls_socket and tls_acceptor. You just need to implement the functions declared in tls.h

- int close_tls_socket(tls_socket *socket);
- int tls_write(tls_socket *socket, char *buf, size_t buf_len);
- int tls_read(tls_socket *socket, char *buf, size_t buf_len);
- tls_acceptor *create_tls_acceptor(int port);
- tls_socket *accept_tls_connection(tls_acceptor *acceptor);
- int close_tls_acceptor(tls_acceptor *acceptor);

To run your server with the TLS Socket, you should

```
$ make myhttpsd
$ ./myhttpds ...
```

When you navigate to your server with your Internet browser, be sure to include the `https://` prefix.

Note: After successfully porting the TLS "driver" to your web server, you will receive warning messages in your browser along the lines of "Connection is not Private." This is because your SSL certificate was not signed by an accepted certificate authority, such as Comodo, LetsEncrypt, GlobalSign, etc. You can manually install your certificate as a trusted certificate in your browser, or you may simply add an exception. In Chrome, you do this by clicking "Advanced" and "Proceed to ..."

If you haven't added the certificate exception when you first connect to your server, Firefox will immediately close the connection—often before your server can write a response back to Firefox. Your code will likely crash because you receive a `SIGPIPE`. You must safely handle this signal.

6.1.1 Resources

We have provided a simple reference TLS server in the file **examples/tls-server.c**. You can build it with **make tls-server** and play around with it to learn how to use the OpenSSL library. The

Simple TLS Server example assumes that your private key and certificate file are called **cert.pen** and **key.pem**. And that you can bind to port 4433 - you will want to change this port number so that you can run the demo. You can generate these keys by running the following command and filling in the prompts:

```
$ openssl req -newkey rsa:4096 -nodes -sha512 -x509 -days 21 -
nodes -out cert.pem -keyout key.pem
$ chmod 700 *.pem
```

Note: For this task, you should primarily be understanding the flow of the provided Simple TLS Server and how those functions in socket.c control whether to use tcp functions or tls functions.

6.2 cgi-bin

For this part, you will implement CGI-like behavior on your server. The Common Gateway Interface allows a server to handle dynamic requests and forward them on to an arbitrary executable, typically inside of a specified folder like `cgi-bin`.

When a request like this one arrives:

```
GET <sp> /cgi-bin/<script>?{<var>=<val>&}*{<var>=<val>}<sp> HTTP/1.0 <crlf>
{<headers> <crlf>}*
<crlf>
```

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.

For the GET method, the string 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>` should be sent back to the client.

The general sequence of events for handling a cgi-bin request then is:

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/http-root-dir/cgi-src

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

Note: The output of CGI-bin scripts expects to be able to write directly to the socket stream. In the lab template provided, we form a response structure and write that to the socket. CGI-bin scripts also expects to be able to set some of the header fields since they have direct control of the output socket. You have a few options on how to tackle this problem. The first solution could be to parse the output of the cgi-bin script and detect any headers that were set in the output. Another solution could be to write a partial HTTP message over the socket (just HTTP/1.1 200 OK) and allow the CGI-bin script to write to the socket without the http_response structure involved at all.

6.3 Loadable Modules

For this part 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()`, your server should load that module into memory using `dlopen()`—if it has not been previously loaded. Your server should then transfer control to this module by first looking up the function

```
extern httprun(int ssock, char *query_string)
```

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
```

should cause your server to load the 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.

The hello.c module can be found in the examples folder. You should run "make" inside the examples directory to build it.

There is an example of how to use loadable modules, named `use-dlopen.c` in your Lab 5 examples directory as well.

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);
```

6.4 Statistics and Logging

HTTP servers deal with a lot of data, and you often want to see where things may have gone wrong or look at how users are interacting with your server. In this part, you will implement a logging mechanism in your server to take note of various parameters when you receive a request.

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

- The name of the student who wrote the project
- The time the server has been up (the amount of time myhttpd has been running)
- The number of requests since the server started
- The minimum service time and the URL request that took this time.
- The maximum service time and the URL request that took this time.

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`.

For logging, 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 URL requested
- The response code

The log should be stored in a file named `myhttpd.log` in your lab5 directory.

6.5 Directory Browsing

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 what a directory should look like is indicated in http-root-dir. Mimic the apperance and functionality of the directory browser found at that link. You may wish to read the man pages for `opendir()` and `readdir()`.

- You do not need to respect the **Accept_field** of the Request message. We do expect that you give a valid Content-Type in your response. You can call **char *get_content_type(char *filename)** (defined in misc.h) to get this information for you. Be careful though! This function does not validate the filename and simply calls **execl** and gives you the output.

- You will need to be capable of sending the following status codes: 200, 404. When grading for 404, we will only check that the status code was set correctly. You can provide any content message you wish.
- The name of the current directory should be shown at the top of the page in the format "Index of <directory>"
- The name, last modified date, and file size should each be a seperate column (you do not need to worry about description).
- A "Parent Directory" link should always appear at the top of the list of files. Clicking this will take you to the parent directory of the current location. Clicking this button while in the highest explorable directory area will return to index.html.
- Icons should be implemented for the various file types. All the icons can be found in **http-root-dir/icons**.
  - "Parent Directory" link should use "back.gif" icon.
  - Directories should use "folder.gif"
  - Images of .gif, .png, .jpg, .jpeg, and .svg should use "image.gif"
  - All other files should use "text.gif"
- **Extra Credit:** Implement sorting by name, size, and modification time. The "Parent Directory" link should remain on top despite sorting.

Resources

opendir, closedir, and readdir

7 Extra Credit

You may notice that the logging features become somewhat complicated when multiple processes are involved (such as with the -f option). How do you coordinate writes to the log file between these processes? Successfully creating a logging feature that cleanly logs to the same file from multiple processes will earn 5 extra credit points. Hint: look up the mmap() function man page.

8 Turning in Final Version

The deadline for final version is Monday, November 11th, 2019 at 11:58 PM

```
$ make clean
$ make myhttpds
$ make submit_final
```

9 Testing

The primary methods for testing your server should be web browsers like Chrome and Firefox with the developer console. If Chrome and Firefox aren't giving you useful information, you can use tools such as telnet, netcat, or curl.

```
$ telnet localhost 4747
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET / HTTP/1.1


HTTP/1.1 200 OK
Connection: close
Content-Length: 12


Hello CS252!
Connection closed by foreign host.
$ printf "GET / HTTP/1.1\r\n\r\n" | netcat localhost 4747
HTTP/1.1 200 OK
Connection: close
Content-Length: 12


Hello CS252!
$ curl localhost:4747
Hello CS252!
```

ApacheBench (ab) is a useful tool for benchmarking and stress testing web servers. You can read more about it on Wikipedia or at Apache.

Note:

ApacheBench is **NOT** a good way to test if your basic implementation is correct. For that you should be testing with one of the methods above (curl, netcat, telnet, chrome, firefox). ApacheBench is good for checking that your server works under load after you know it serves correctly. In particular it can be useful for testing concurrency.

Create a file called auth.txt and place your username and password in with the format

```
<username>:<password>
```

This command requests index.html 1000 times with 25 concurrent requests from an http server on localhost running on port 8080. You will need to adjust these parameters to fit your server.

```
~cs252/bin/ab -B "auth.txt" -n 1000 -c 25 -r "http://127.0.0.1:8080/index.html"
```

Here is a general pattern for the command to run:

```
~cs252/bin/ab -B "auth.txt" -n ${n_reqs} -c ${n_conc} -r
"${proto}://${host}:${port}${path}"
```

Note:

- **Make sure to actually keep your user and password in the auth file and use the -B option over the -A option. If you do not others will be able to see them when listing processes.**
- ab doesn't work when using the hostname localhost so use it's IP instead (127.0.0.1) when benchmarking locally.
- If you add the cs252 bin to your path then you can just run ab. Add `export PATH=PATH:~cs252/bin` to your preferred shellrc (.bashrc, .zshrc, etc.)

HTTP response code list:

At a minimum, your solution should implement the following response codes:

- 200 . OK
- 400 . Bad Request (should be implemented as part of basic server)
- 401 . Unauthorized (Basic HTTP Authentication)
- 404 . Not Found
- 405 . Method Not Allowed (e.g., in repsonse to something other than a GET request)
- 500 . Internal Server Error (internal failures e.g. missing EOF on file read, pipe failure, fork failure, exec failure, etc)
- 505 . HTTP Version Not Supported (anything other than HTTP/1.1 or HTTP/1.0)