CS339: Lab1 Write-Up

Zachary Romrell
Williams College
zr3@williams.edu

1 Introduction

The goal of lab 1 was to build a functional web server and to gain a better understanding of the basics of distributed computing. We were also required to work in the programming language C or C++ which introduced a handful of other technicalities and made you further consider how to allocate your memory in a way such that different clients' connections remained safe. Some of the main conceptual components I first had to understand before being able to dive into the code involved the simple client/server structure.

Abstractly a web server listens for connections on a socket bound to a specified port of a given machine. Clients can then connect to this socket and communicate with the server through the HTTP protocol. Throughout the lab we were only concerned with a local host connection, however, in the lectures we learned how a client from a different domain could attain the IP address of a server and connect. Simply the process of making a web server that is supported through a web browser was eye opening to me and gave me a better understanding of how the internet operates. Once a connection between a client and host is established there remains a handful of assumed functionality that the server must support for the client. There also exists questions like: how may a server support more than one client, how long should the server keep any given client's connection open for, and other questions around the possible design choices present when building a web server. In the following subsection [1.1] I give a high-level description of the HTTP protocol. In section [2] I will start to answer the questions posed above and further discuss some additional feature my web server contains. In section [3] I will talk about how my server performs against stress tests and answer some of the discussion questions provided in the lab handout. Finally, in section [4] I will give an overview of the project as a whole and reflect on my journey.

1.1 HTTP Protocol

It was quite a slow process to learn and discover all the functionality automatically assumed to be present within a client server connection. First off, servers must support GET commands with a specified file name followed by the HTTP connection type (1.0 or 1.1). Processing such GET commands required patient string parsing and also relied on built in protocols like the existence of two newline characters at the end of any received message. One component that I forgot to check for was to make sure that the host field in the received request was non-empty and corresponded to my server. Such technicalities and specific safe-checkings were all new to me and gave me further insight into how a malicious client could try to create difficulties for a server.

Within processing GET commands you must also know how to extract information like the file requested, desired connection type, and how to deal with incorrectly formatted requests. With any parsed filename one must check if the file exists and if the given permissions of the file allow for the client to access it. Checking for cases like the ones above also requires HTTP communication back to the client such as ”404 File Not Found” or ”403 File Forbidden”. Such HTTP implementation details are intuitive, however, were not things I focused on my first time around implementing my web server.

Another component in the HTTP protocol is allowing for the functionality of two slightly different connection types: HTTP/1.0 and HTTP/1.1. The main difference between HTTP/1.0 and HTTP/1.1 is the fact the HTTP/1.1 keeps the connection with the client open after processing the first request while 1.0 closes the connection immediately after. In section [2] I will further discuss how HTTP/1.1 fit into my chosen architecture type and further describe the simple heuristic that decides when to close a HTTP/1.1 connection.

Lastly, HTTP has a basic protocol for sending back the contents of a file. In order for a client to successfully process the requested file, they must receive a header that specifies that the file is available and has been read properly, 200 OK, the date and time in which the request was processed, Date: . . ., the length in bytes of the file contents, Content-Length: . . ., and the type of file being sent, Content-Type: . . .. One can include other headers as well, however, the headers above are necessary. Following receiving the header, the client is now ready to receive the contents of the file. Thus, the server can use the file descriptor of the opened file and read and send the contents to the file descriptor corresponding to the port the client is connected on.

All of the components above represent the basic high-level structure of the HTTP protocol. Any functioning web server is required to be support such characteristics. Next I will discuss the architecture type and some specific design choices and features I added to my web server.

2 Architectural and Design Overview

For this lab there were three main options in how you structured your web server. Since I was completely new to handling concurrent programs and wanted to work independently I decided to stick to the most simple architecture, a multi-threaded approach. I relied on the provided echo server in my implementation which supported a simple layout for accepting a connection to the server. Specifically

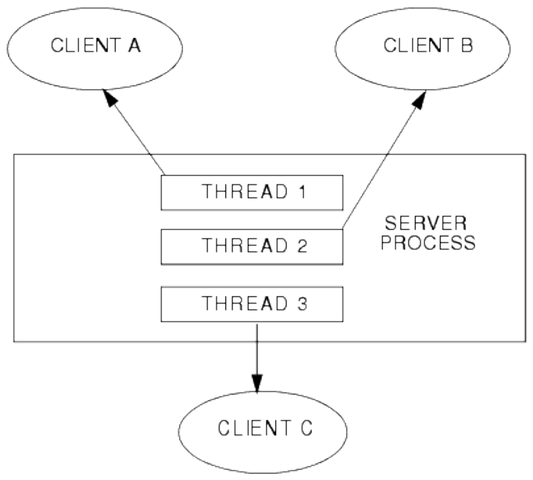

Figure 1: Threaded server example. Note that the Server Process would have connection to a local database for file requests. [1]

the server was designed to have one specific socket for handling new client requests and then spawned separate threads for handling new client requests. When creating a new thread the server also passed along the specific file descriptor corresponding to the port of communication with the client.

However, this process of threading could create some issues in memory on the stack for different clients’ connections. Perhaps one client is requesting a very large file that requires more space on the stack than allowed for that given thread. Since the threads’ memory will be adjacent to each other on the stack one clients’ requests could impact the memory and ability for the server to respond to another client. Thus, within the thread one must dynamically allocate memory to the heap using malloc(). Unfortunately, with my foggy memory of CS237 I did not use malloc for my variables in the threads and thus, my web server is susceptible to such a memory error. Luckily, the act of messing this up has further ingrained this concept into my mind and such a fix is easy to do in code. Despite this low-level error I did not see any effects in my ab testing most likely due to the small size of the files requested and sent, see more in section [3]

With the architecture type of threads, supporting HTTP/1.1 was quite trivial. Within the given thread you could loop through the code and continually read requests until the time-to-live period has expired or the last processed request has a connection type of HTTP/1.0. Setting up the time-to-live component of the thread was not difficult since sockets have specific flags for such cases. Simply with the setsockopt() command one could set the flag corresponding to the receive time to a specific value. However, it isn't obvious how one should determine such a time-to-live value. One might intuitively expect as more clients connect to a web server, since the server is being worked harder, the corresponding time-to-live values of the new clients should decrease. Such a heuristic is reasonable and is exactly the type of approach I implemented when determining the time-to-live value for a new client's thread. In order to represent the time-to-live in terms of the number of current clients one must keep tally of the number of current clients connected. Using pthread_mutex_t one can create a global variable for the thread count and lock and increment the variable for each additional client/thread and lock and decrement the variable whenever a client disconnects/thread closes. The corresponding heuristic I implemented is described in the following formula where TTL_n = the time-to-live of the nth client connected:

$$TTL_n = 5 + \frac{50}{n}$$

Thus, the first client that connects will be allocated a time to live of 55 seconds. The second client will have time to live 30 and so forth. For the better or the worse this time to live value is independent of the specified connection type which will be irrelevant to HTTP/1.0 and potentially restrict HTTP/1.1's connection time if a couple HTTP/1.0 connections aren't done reading/sending a file's contents when the 1.1 request is processed. However, this may be desirable if the HTTP/1.0 requests are requesting very large files that are utilizing a lot of resources on the servers end.

An additional feature I provide is allowing the web server to process a shortcut GET command. Once you have parsed the filename field there is a convention that you can check for. It is known that "GET / connection.type" typically corresponds to "GET /index.html connection.type", which is thought of to be the base file in any connection. Thus, with a simple string comparison one can update the specified filename in the case of this shortcut. It is also useful to contain a quick check to make sure the requested file name does not contain any "../" character sequence as such a sequence would move the client out of the current directory and possibly into a new directory they shouldn't have access to.

It is also nice to feature to send feed back if the original client request is ill-formatted, 400 Invalid Request, or as a server you don't support such a request, 501 Not Implemented. In the else cases of the GET request I also check for some common HTTP requests like: "HEAD /", "PUT /", "PUSH /", and "TRACE /" and associated such requests with a 501 Not Implemented error. As a web server it is better to account for all the possible errors a client can throw at you and allow for effective communication if such a request cannot be processed.

3 Evaluation

After executing the make file one can run my server with the following command: webserver -document_root ROOT -port NUM with a specified document root and port number. Once running the server you can connect to it via telnet or firefox in the traditional way you would connect to a localhost or machine on the same network. The GET request is formatted in the traditional way, "GET /filename connection.type".

I manually tested my permission checks and error headers and included a small txt message body for each of my error messages so they are displayed in the web browser. I decided not make separate files for the message bodies and just manually send a string. When running my webserver versus ab stress testing it performed fairly well. When requesting a locally downloaded version of the classic sysnet.cs.williams.edu 80 web page with the following stress test parameters ab -n 10,000 -c 100 my server had a 100% completion rate. When increasing the number of requests to 100,000 keeping the same concurrency I still received a 100% success rate. Increasing the concurrency to 1,000 didn't effect my testing when I kept -n equal to

10,000. However, when I tried ab -n 100,000 -c 1,000 on average I would received around 250 failed requests. Overall, my web server performed fairly well against a high volume of requests and high concurrency for relatively small file requests. However, as expressed earlier in this report due to not using the heap for my threads' memory I would expect to run into serious memory errors when attempting to stress test with larger file sizes. Luckily, this is an easy fix and in the future when using C I will better understand the advantages and times in which you should use malloc(). Overall, I think that my web server behaves fairly optimally and is organized in a respectable manner. One slight optimization I could see myself doing if I had more time is passing along the file extension versus the entire filename when calling the send_header function.

I will also provide a response to the following discussion questions.

If I had to support ".htaccess", I would first prior to creating a thread for a new client check if the client's address is among one of the IP addresses listed in ".htaccess". I would also order the ".htaccess" file by similar IP addresses locations followed by newlines. Thus, as the program searches through the list it can skip to the next newline if the following locations mismatch. It would also be efficient to create a hash map of the prohibited IP addresses since retrievals are O(1).

HTTP/1.0 would perform better when the client is making a few number of requests for larger files. This is the case because the cost of opening and closing the socket for each large request is quite small relative to the request size. However, if one wanted to retrieve more than one smaller files it would make the most sense to use HTTP/1.1 since the cost of opening and closing a socket is greater in proportion to the request sizes. It would also make sense that HTTP/1.0 would outperform HTTP/1.1 when the bandwidth and latency is restricted in a client server connection. If the bandwidth and latency is constantly changing it is likely that the HTTP/1.1 connection will be quite unstable, however, it still may outperform HTTP/1.0 for certain file sizes. Overall, the decision comes down to the proportion of the overhead of opening and closing the socket to the specific file sizes.

4 Conclusion

In conclusion I greatly enjoyed building this web server and thought that this lab nicely complemented the concepts taught in lecture. Although figuring out all of the implementation details for the HTTP protocol and working in C presented some frustrating moments, I found that these difficulties only made me better understand the technicalities and nuances that go into implementing a web server. Throughout this lab I not only learned about all the necessary components in a client server connection, however, I also better understand the technology that I use everyday and what a web server actually does. This lab has further opened up my world of computer science and I now better understand how different components like web servers, web browsers, routers, and DNS servers interact with each other to provide the internet. I would like to close with a big thank you to Jeanie and Jae for helping me with my implementation and the strange C errors that I managed to create. Thank you.

References

[1] *Threaded Server Architecture Image*.