Web Server Project Writeup

Brian Kamau, Victor Kilel
Williams College

1 Introduction

We implemented a HTTP web server that supports both HTTP 1.0 and HTTP 1.1 requests. This program listens for connections on a socket (bound to a specific port on a host machine). Clients connect to this socket and use a simple text-based protocol to retrieve files from the server. In the next sections, we discuss the implementation in further details including the implementation design, features covered and instructions on how to run the program. We also provide an evaluation of its efficiency and challenges we encountered during implementation.

2 Architectural Overview

The server has been designed to support both `HTTP/1.0` and `HTTP/1.1` web protocols. The server immediately closes the connection once it receives a `HTTP/1.0` request but keeps the connection open if the request uses the `HTTP/1.1` protocol.

The server also supports concurrent requests via the `HTTP/1.1` protocol. There were three paths to take to accomplish concurrency:A multi-threaded approach, a multi- process approach or an event-driven approach. The authors of the program chose the multi-threading approach in the design of the program. This option was chosen for various reasons, a few of which we will mention. First, multi-threading is quick to create and requires few resources (it is lightweight), unlike multiprocessing which requires a significant amount of time and

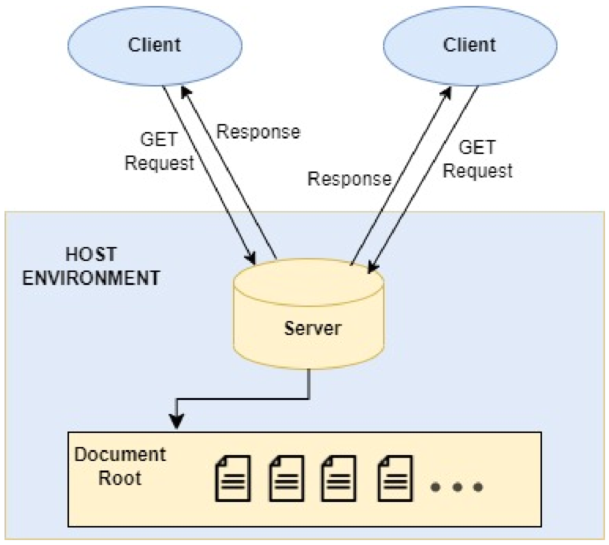

Figure 1: Illustrating the high level design of the web server

specific resources to create. Furthermore, multi-threading takes advantage of multiple CPU cores that are present in modern day computers. Perhaps, the most tempting of all, was that the code required to implement a multi-threaded approach is usually straightforward. This was especially crucial given the lack of prior knowledge in parallel processing of the authors and the short time frame of the project.
Aside from concurrency, a timeout was also set up for the server. That is, the amount of time the server can sit idly on a connection without receiving any request from a client, before it closes the connection. This figure was set to be 30 seconds.

The server handles the following status codes:

1. 200 : In this case, the request is executed successfully without any error and the requested resource is sent to the client,

2. 404 : This is an error for when the requested file is not found. We return a html-formatted string that displays a `File not found` message.

3. 403 : This occurs when the client tries to access a resource for which it has no permissions to. Again, we return a html-formatted string that displays a `Forbidden request` message.

4. 400 : This occurs when the client issues an improperly framed request. Once again, we return a html-formatted string that displays a `Bad Request` message.

It is important to note that the connection is always kept open if an error is encountered during the execution of a request, to give the client ample time to resend the request.

Challenges encountered

Like any other project, this project was not devoid of its challenges. We provide two specific challenges that were especially challenging over the course of this project and how they were overcome.
The first challenge encountered was getting the server to successfully parse the client request. To put this into context, when the server receives any client request, it stores it in a fixed sized buffer of 4MB. Further, the server must iterate through the buffer to ensure that any and all requests present in the buffer are executed and the appropriate message sent for each one of them. Our first attempt involved iterating through the buffer looking for two carriage returns[1], then modifying the buffer to exclude the request that was already found. However, since the buffer was declared to be of fixed size, it could not be modified. The workaround we settled on was creating a temporary buffer of the same size that we would copy our remaining requests to and then serve them. Of course, the downside to this was that the server was using more and more space than what the authors desired.
The second major challenge encountered was that of memory de- allocation. The server manually allocates memory to store a single request from the user. However, de-allocating that section of memory would often result in a segmentation fault[2]. We mapped out the cause of the problem to be shallow copying of some variables[3]To remedy this, we switched to deep copying. Of course, the downside to this was that the server was using more resources than desired.[4]

3 Evaluation

The authors tested the performance of the web server by running several different loads on it using the `ApacheBench` tool. This is is a bench marking tool that measures the performance of a web server by inundating it with HTTP requests and recording metrics for latency and success.
Our load involved a total of 100 requests, run on different values of concurrency $1 \leq c \leq 10$.
It is observed that on the first run, the `ApacheBench` tool runs successfully on whatever value of c is inputted. However, running the requests again and again on the server degrades it and eventually causes it to crash or just terminate.
There are a few guesses as to why this happens. One is that running a single `ab` command requires a lot of space. Continuing to run these commands may eventually deplete the space available. Another guess is that the `ab` command may leave lingering processes in the background and these ”zombie” processes may in turn clash with the subsequent `ab` commands. Finally, it may also be an issue with the code structure itself that causes the server to degradate over several runs of the `ab` command.

[1]In a real world client-server system, any request is appended by two newlines to signify the end of that request.

[2]Core Dump/Segmentation fault is a specific kind of error caused by accessing memory that “does not belong to you.”

[3]A shallow copy of an object is a copy whose properties share the same references (point to the same underlying values) as those of the source object from which the copy was made.

[4]A deep copy of an object is a copy whose properties do not share the same references (point to the same underlying values) as those of the source object from which the copy was made.

Given the short time frame of the project, we did not spend too much time on figuring out the cause of this issue. What is important to note though from the `ApacheBench` tool is that it indeed confirm that the server can serve concurrent requests on different threads without any faults.

4 Thought Questions

We address two questions in this section.

The first question would be how we would go about supporting `.htaccess`[5] in the server. The server listens for connections before accepting an incoming connection. The server could listen to an incoming connection check the `.htaccess` file to validate the IP Address of the client before accepting the connection. We could accept the connection, validate the IP address then close the connection but this method leaves the server vulnerable to sniffing attacks.

Like we said earlier, the server has been configured to support both `HTTP/1.0` and `HTTP/1.1` web protocols. The next question then addresses the different use cases between `HTTP/1.0` and `HTTP/1.1`. Recall that `HTTP/1.1` is a persistent connection, that is all connections are "kept alive" unless declared otherwise. All objects must also be retrieved in serial.Therefore one could imagine that in the general use case when a client connects to a server requesting for several resources, a persistent connection is far more optimal than `HTTP/1.0` , which closes the connection when the server sends a response. However, because `HTTP/1.1` serves requests serially, if one request cannot be loaded, it blocks all the other requests behind it.
Therefore,one use case where `HTTP/1.1` would definitely be disadvantageous would be prioritization. `HTTP/1.1` you cannot send the server several streams of data at once. Therefore, in such a case you would think of perhaps opening several `HTTP/1.0` connections on which you can send these priority requests.

[5]Web servers often use ".htaccess" files to restrict access to clients based on their IP address

5 Conclusion

Writing the code for the project was no easy task, especially given the fact that the authors had no prior experience with server programming. As stated earlier, the greatest limitations faced was basically memory management. However, solving the problems themselves helped reinforce an understanding of server programming.
All in all, the project itself was enjoyable. This is mainly attributed to the fact that the project was designed incrementally, meaning that whenever problems were encountered, we could fall back on the last working version of the project and proceed to rebuild from there.
Finally, the authors would like to acknowledge their Professor, Jeannie Albrecht and their teaching assistant,Jae Surh for their ever present guidance and help throughout the course of this project.