HTTP Webserver in C
Diary

Introduction

This diary forms part of the project to create a multithreaded webserver to handle HTTP/1.1 requests in C. It illustrates: the experiences I had; the challenges I overcame; and the changes I would make if I were to attempt this assignment again.

Experiences

To begin with, I decided to read through some sample code, written in C, provided for us. This code was for a client-server pair for sending messages; the client would establish a connection with the server, and messages sent from the client to the server were echoed back. Gaining an understanding of this code was vital in laying a foundation for my own project.

At this time, I made the decision that I would implement my HTTP webserver in C rather than Java. I realised that it would be more challenging but would ultimately be beneficial in improving my skills in programming at a low level. After reading through Ian's code, I decided that some practices I follow whilst programming had to be updated, especially around commenting. I often find that I don't provide sufficient explanation for sections of code, leaving me confused when I read over old code later on. I made sure to provide comments for each small section of code (3-5 lines) within my server, so I would both be able to track how I was progressing, but also to make the code understandable in the future.

With this, I began my implementation. I set up a git repository to maintain good version control over my project, allowing me to roll back to previous versions when mistakes were made. I decided not to try and adapt Ian's code, but attempt my own structure. I chose to do this to force me to learn more about C, since it is a ubiquitous language for developers working with networks and operating systems; I also have a personal interest in trying to understand the structure of the language, since I already have a good understanding of functional and object-orientated languages.

With my understanding of threads in C, my first task was to write a handler function that each new thread would run when created. This would parse requests from the client, locate the file(s), construct the response, send it to the client, then clean up all memory used. This proved a challenge, as the Hypertext Transfer Protocol is very specific about how messages must be sent, and incorporating that into the way C works as a programming language led to a lot of time debugging my solution.

Once I was happy with how my function worked following some testing in a separate file, simply calling the function directly rather than creating and running a thread with it, I ensured my file structure was adequate for serving web pages. I created index and error pages for the server to send the client dependent on the nature of their request. I also created a favicon.ico file, since I discovered that this was often requested by browsers.

To complete my implementation, I required a main function to handle the creation of the server socket, and initiate connections with the client(s). This was straightforward in comparison to the handler function used by the threads, and took a lot less time for me to understand. Following this, I began testing my solution to ensure it was robust. I had some issues with the server sometimes timing out when a client had been served, which my solution still has. My understanding of sockets and threads is still not as complete as I would like, but my solution serves files to multiple clients as stated in the exercise brief.

Challenges

The main challenge I found was parsing the request as send by the client. Upon separating out the request type (GET, POST, PUT etc.), file ('/' being index.html for example), and the protocol version (HTTP/1.1 is the only protocol version my server handles), I needed to perform comparisons on these. I used the `strcmp()`function for this, but found that strings extracted from the request were not matching properly. After spending some time looking into how strings work in C, I realised that all of the separate parts of the request did not have a null terminating character `'\0'` by default, so I had to go in and add these to every string to make them match correctly. This, to me, seems very clunky but appears to be the only way to correctly match substrings from larger messages.

Related to the above point, I realised that all strings have a length one larger than you would expect. Using the `strlen()` function to gauge the length of strings and files confirmed this, which is something I will always bear in mind when programming in C.

I also found it interesting that the `\r\n` pairing was required between every line of an HTTP response, with a double instance of this to make empty lines. The `\r` being designated as carriage return as on a typewriter was a fun, little quirk that reminded me how far programming has come since its early days.

Reflection

If I were to attempt this assignment again, there are a few things I would change. If I had managed my time better, I would have liked to test my solution on a variety of web browsers, and by connecting to my server when running a web browser on a different machine. Whilst being happy with my submission, I feel that my choice to start from scratch was foolish, especially when Ian had provided a client-server pair that was much more robust than my solution. If I had adapted this code to suit serving HTTP responses rather than just echoing a message back to the client, a lot of the work around sockets and threads would have been taken away. Whilst I do regret not taking this approach, I have both learnt more about the low-level interaction between servers and clients, as well as the process of refactoring well-tested code saving time and effort; there are few disciplines that encourage the sharing of knowledge as much as computer science.