Cuteserver: A web server written in C

Project for the Lecture ”Operating Systems” FS24

Rahel Kempf Ephraim Siegfried

June 8, 2024

Introduction

”Web servers are everywhere”, Shakespeare once said. This is especially true today. But ”Hark! Dost the common folk comprehend what these web servers be? And doth they grasp the manner of their workings?”.

In this project we aspire to understand the inner workings of web servers. It is of interest for us, as we both recently set up our own home servers and came in contact with different tools such as nginx. Building our own web server helped us be more competent in using those tools. It also helped us deepen our knowledge of OS topics such as socket programming, thread/process creation & management and inter-process communication.

This project resulted in Cuteserver, a simple web server written in C. It can handle various HTTP/1.1 requests from multiple clients concurrently. It is configurable and supports hosting content for multiple domains and is thus similar to a reverse proxy. It can be hosted easily with a prepared docker image. In the following chapters, we will discuss the steps taken and the challenges we faced in this project.

Background

In this section we explain some concepts we had to understand implementing our project.

HTTP Protocol

The Hypertext Transfer Protocol (HTTP) is an application layer protocol which defines how data can be exchanged over the Internet. It is generally sent over TCP, as it relies on a reliable transport protocol. As a client-server protocol, requests are initiated by the recipient (browser) and responses are served by the provider (web server). The structure of these requests and responses is defined by HTTP. [1] For our project, we mainly focused on version HTTP 1.1.

The general structure of a request is the following: The first line denotes the **Method**, **Path** and **Version of the Protocol**. The following lines are called **Headers** and contain information for the servers. An empty line signals the end of the headers, for some request types (such as POST) the **Request Body** follows. A header to point out would be the **Connection** header, which was introduced in version 1.1 to provide persistent connections by reusing a TCP connection for multiple request/responses instead of opening a new connection for each response. The response follows the same structue, just that the first line contains the Version of the Protocol First, then the Status Code and Status Message [2]

There are several Request Methods which signal what kind of action should be performed. [3] For example:
GET retrieve state representation of target resource. Parameters sent through URL
HEAD retrieve only metadata.
POST request that target resource processes the representation in the request body.

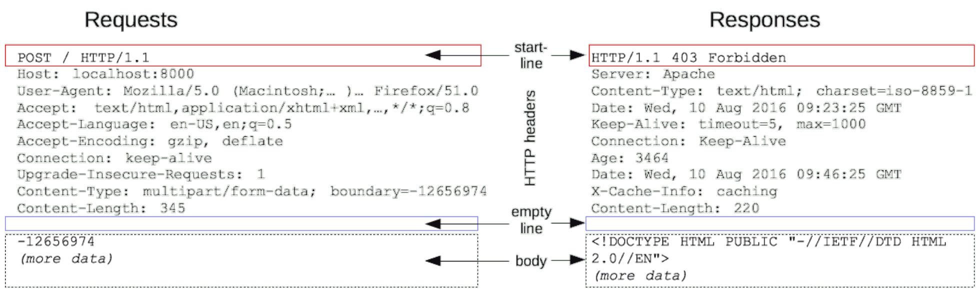

Figure 1: HTTP Requests and Responses

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
[2] See HTTP Codes https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
[3] https://datatracker.ietf.org/doc/html/rfc1945#section-8

Common Gateway Interface

The Common Gateway Interface (CGI) [4] is a technology we encountered during our research on handling HTTP POST requests. CGI is an interface that defines how the web server interacts with external programs. This enables the creation of dynamic content, such as generating user-specific webpages or processing form submissions, and facilitates communication between the web server and application backend services.

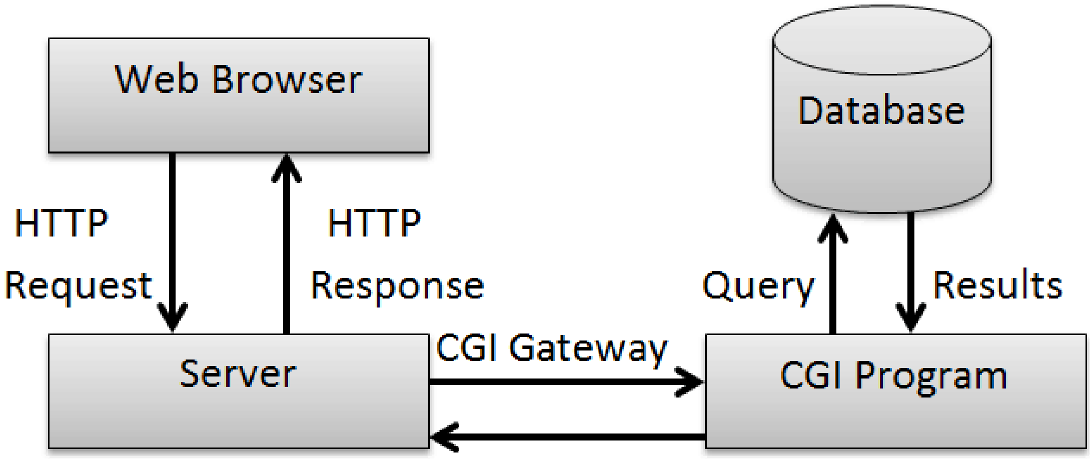

Figure 2: The Common Gateway Interface

When a request with a URL denoting a CGI script, like example.com/backend.cgi, enters the web server, the server locates the executable file called backend.cgi. It then spawns a process to execute the CGI script and communicates with this process via piped standard input/output and environment variables. After the CGI script sends its response to the web server, the server may verify the response and forward it to the client.

Although CGI is now quite outdated and less frequently used due to the inefficiency of spawning a new process for each request, enhanced versions such as FastCGI or SCGI are widely used. These versions improve performance by spawning CGI scripts only once. In our project we implemented the original CGI specification from 1997.

Implementation

Development Environment

The development of Cuteserver was primarily accomplished through pair programming. We held regular meetings where we shared our terminals using a shell sharing tool called **sshx** [5]. This tool was particularly effective for collaboration since we both used **Neovim** [6] as our code editor.

Initially, we used **Makefiles** to build the source code, but this approach was inefficient due to frequent adjustments. As the project grew more complex, we decided to use **CMake** for more efficient project building and file generation.

Our implementation strategy for the web server was iterative, starting with small, manageable tasks and progressively tackling more complex ones. We regularly tested the code by running it with different inputs to ensure functionality.

[4] https://datatracker.ietf.org/doc/html/rfc3875
[5] https://sshx.io/
[6] https://neovim.io/

Project Structure

We structured our code into modules, each handling a task or describing a logical "unit". In the **main** module the server socket is set up and incoming TCP connections are handled through a threadpool. When a new request is received, a thread starts serving the incoming request. Our implementation supports Keep-Alive Headers. Per default a "keep-alive"-Connection is assumed, so multiple responses can be sent over one TCP connection. (See Results). To prevent too many idle connections, a timeout of 5 Seconds is set.

Static File Requests

Each incoming request is parsed into request line, headers and body, and saved in a struct called request_info. We differentiate between static (GET or HEAD) and dynamic (GET or POST) requests. For static requests, we access the file to receive metadata such as content length and type. For HEAD-Requests only the Headers are sent, for GET-Requests the file is also sent.

CGI

Dynamic Request are handled with CGI. This means the incoming request is parsed, then passed to the **CGI handler** which creates a new process running the corresponding cgi script. The CGI Protocol defines "meta-variables" [7] which have to be passed from server to CGI script using environment variables.

The CGI process is started using the **execve** function, allowing us to pass the environment variables as an argument. Additional data has to be passed to the child process through standard in/output. Communication between cgi process and server (parent process) is enabled through pipes. A pair of pipe descriptors is created before the child process is forked. Then, the child's descriptors are redirected to standard input and standard output [8] (See Figure 3).

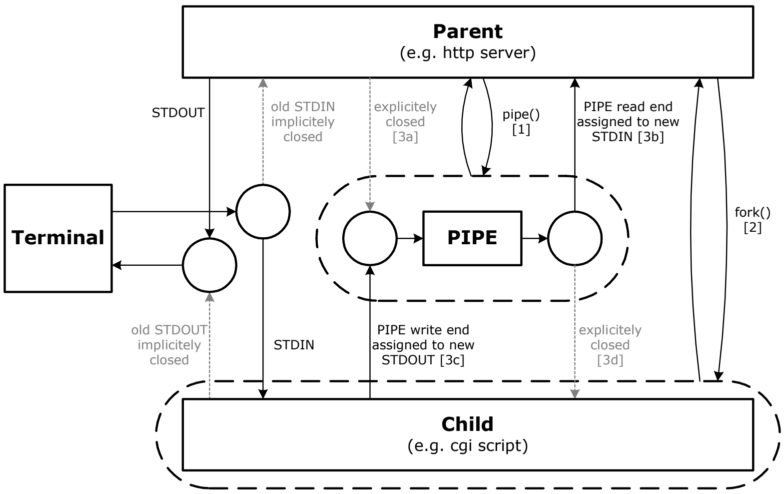

Figure 3: Pipe Redirection

Server Configuration

From the beginning, we aimed for our web server to support hosting multiple resources connected to different domain names. To achieve this multidomain support, we needed to make the web server configurable. To ensure user-friendly configuration, we used **TOML** [9] files. The web server parses a configuration file listing all resources and their locations for the server to serve. Additionally, users can specify resource-independent settings, such as the number of active worker threads and log storage locations. We also provide a command line interface where users can specify the TCP address and the path to the configuration file.

[7] https://datatracker.ietf.org/doc/html/rfc3875section-4.1

[8] http://www.fmc-modeling.org/category/projects/apache/amp/A_4_Pipes.html, and http://www.unixwiz.net/techtips/remap-pipe-fds.html

[9] https://toml.io/en/

Example Application

We had several web applications to test our web server. Initially, we used simple HTML files and gradually added more complex ones. Eventually, we developed a more sophisticated application using the React framework for the front end. This application is a simple chat app that makes POST requests when users submit chat messages. These POST requests are handled by our backend service, which stores the chat messages in a JSON file and sends the JSON data to clients. We initially wrote this service using the Flask framework in Python to test if interpreted languages could work as CGI scripts. While this worked, we later translated the backend Python script to a C script to achieve faster response times by running compiled executables.

Containerization

We decided to containerize our application for two main reasons. Initially, we used chroot jails to sandbox our application, i.e., a mechanism that isolates a process and its children from the rest of the system by changing their apparent root directory to a specified path. This approach worked until we implemented CGI support, which required the web server to start new processes with the necessary libraries for the CGI scripts. Manually copying those dependencies into the chroot jail would have been too laborious. Docker containers offered an easier solution by securely sandboxing execution and including dependencies. Additionally, Dockerfiles enabled us to automate the building of the web server and the example application, making it easy to use and run. We used multistage builds, first creating containers for building and then copying only the necessary files to the final containers. Multistage build reduced the container size from 1.2 GB to approximately 200 MB. This process enhanced our understanding of containerization.

Results

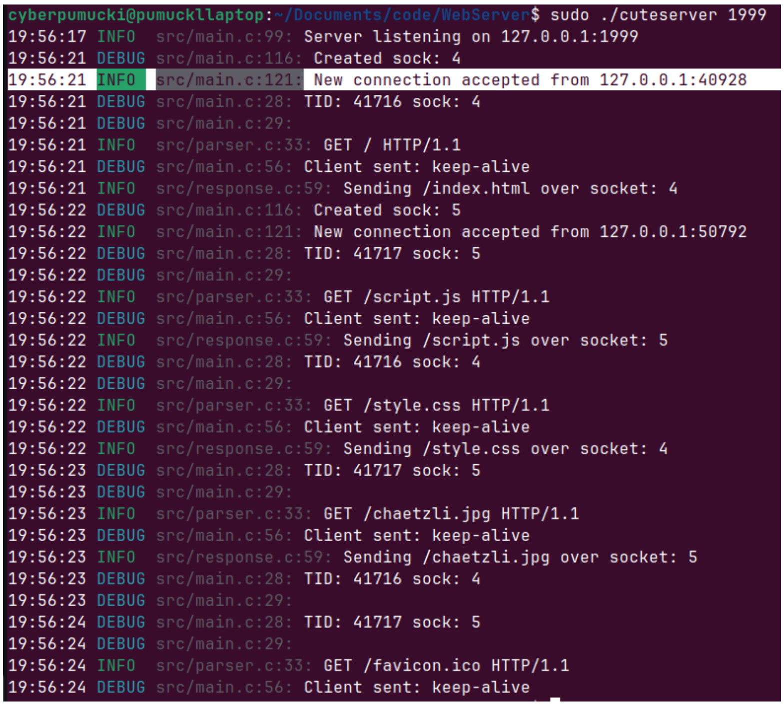

Figure 4: Keep-Alive: Multiple Requests handled over one Connection

Conclusion

We were successful in our project and comparing to our original plan we implemented everything but websockets. What we miscalculated a bit was the POST-Request handling (as we didn't know about CGI before). This took more time than we anticipated.

Project Planning in General: we never looked at the JIRA Board during the project. so maybe next time find a different way to plan.

Lessons learned

Future Outlook

fast CGI Implementing fast CGI to enhance performance and keep up with standards
Security
Websockets

References

Libraries

Hashmap https://github.com/tezc/sc
Logs https://github.com/rxi/log.c
Threadpool https://github.com/Pithikos/C-Thread-Pool
TOML-Parser https://github.com/cktan/tomlc99

Declaration of Independent Authorship

We attest with our individual signatures that we have written this report independently and without outside help. We also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.
Additionally, we affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used.
This report may be checked for plagiarism and use of AI-supported technology using the appropriate software. We understand that unethical conduct may lead to a grade of 1 or ”fail” or expulsion from the study program.