HW1 Report, Rudy: A Small Web Server

David Fischer

September 3, 2025

1 Introduction

The Hypertext Transfer Protocol is one of the cornerstones of the modern Internet, serving as a primary communication protocol that enables distributed systems to function on the World Wide Web.

As part of this report a HTTP/1.1 server partially implementing the RFC 2616[1] specification was created in the Erlang programming language. The server is capable of correctly parsing and responding to HTTP requests, handling multiple clients concurrently, and acting as a file server with functional encoding.

2 Main problems and solutions

As the HTTP/1.1 specification is quite broad building up scope creep beyond the goals of the original assignment was a concern. Work was limited to objectives mentioned or implied in the assignment description. At time of submission the server supports the following feature set:

- Listening on a specified port and delegating requests to one or many handler processes
- Parsing HTTP requests (URI, request headers, and body)
- Responding with one of three HTTP status codes: 200 OK, 404 Not Found, or 500 Internal Server Error
- On *POST*, acting like an echo server
- On *GET*, serving static files from the working directory and its subdirectories with a limited number of supported media types
- Dynamically responding with interactive directory trees
- Respecting the Accept-Encoding request header, specifically gzip

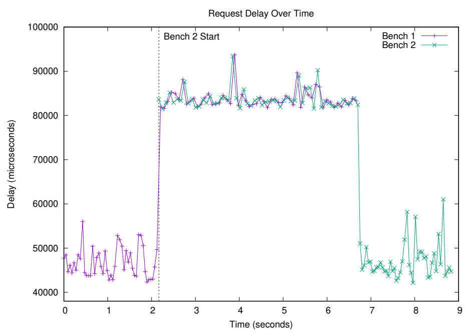
[1] `https://www.ietf.org/rfc/rfc2616.txt`

3 Evaluation

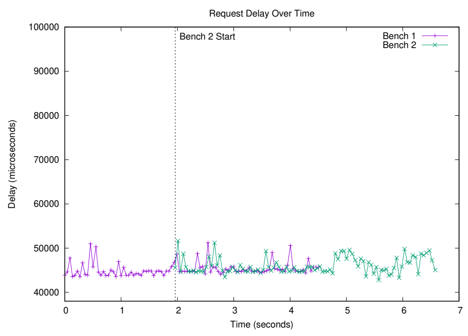
3.1 Throughput

Testing the baseline implementation of *Rudy*, including the artificial 40ms delay, with the given benchmark program measures 100 requests in 4.916 seconds, or ≈20 requests per second. Since the benchmark runs sequentially, the request parsing overhead and response delay can be calculated as 9,16ms meaning the artificial delay is over four times the length of what a regular request would be.

Additionally, while the artificial delay isn't noticeable with a single instance of the benchmark running, starting a second run from another machine doubles the request time during the overlap as can be seen in Figure 1a.



(a) Single Process Handler



(b) Multi-Process Handler

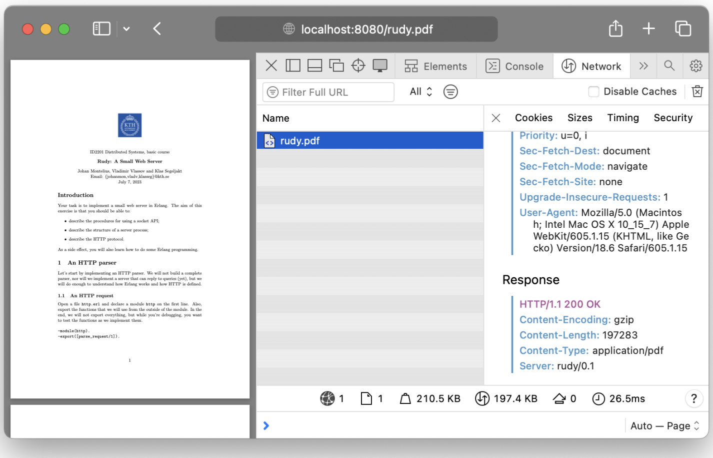
Figure 1: Comparison of Single and Multi-Process Handlers

This outcome is expected, as a single handler process will always be blocked while already responding to a request, during which all new requests are waiting to be picked up. To solve this and increase throughput, Erlang's support for having multiple processes listen to the same socket was utilized. Figure 1b shows the same two sequential benchmarks running against an instance of *Rudy* with two handler processes running, resulting in no spike in response delay.

3.2 Delivering Files

The implementation of *Rudy* supports serving files from a limited selection of MIME types with proper response headers as shown in Figure 2. Additionally, most modern browsers will ask for compressed content to be returned, which was implemented by parsing the request headers and using the zlib module[2] to respond with gzip files.

[2]`https://www.erlang.org/doc/apps/erts/zlib.html`



```
Priority: u=0, i
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintos
    h; Intel Mac OS X 10_15_7) Apple
    WebKit/605.1.15 (KHTML, like Ge
    cko) Version/18.6 Safari/605.1.15

Response

HTTP/1.1 200 OK
Content-Encoding: gzip
Content-Length: 197283
Content-Type: application/pdf
Server: rudy/0.1
```

Figure 2: *Rudy* serving a gzipped PDF file to a browser

4 Conclusions

This assignment provided a helpful review on the underlying processes that enable querying web resources. Additionally, it served as an excellent introduction to functional programming, coming from an imperative background, the initial learning curve was definitely reduced by starting out with Erlang's TCP/IP socket API. Implementing concurrent request handling was pivotal in understanding the language's process-oriented approach.