Report 1: Rudy Web Server

Distributed Systems, Basic Course

Samouil Mosios

August 29, 2025

1 Introduction

This report describes the development of Rudy, a rudimentary HTTP server written in Erlang. This project served as an introduction to the Erlang ecosystem, including the language itself, the Erlang runtime and VM, modules and processes, and interprocess communication via messaging and TCP.

2 Main problems and solutions

The main problem with the initial implementation guided by the notes is the lack of concurrency. The benchmark program provided by the course is essentially a client that sequentially sends requests to the server and terminates once all requests are completed. Although the results may differ slightly for each separate benchmark execution, due to CPU, network, and other factors, the results in the table below are adequately representative.

The table below describes how long it took each of Rudy's implementations (single threaded and concurrent) to complete the benchmarks. This is measured in seconds, and the tests were executed from separate Erlang shells, which simulate the mentioned "clients". The number of requests for each client is represented by the variable N.

Additionally, to simulate the complexities that a server response might entail, such as server-side processing and database queries, a 40 ms delay is introduced to every request. The impact of this delay is examined below.

Delay	**Seconds for** $N = 100$	**Seconds for** $N = 200$
Without delay	0.04–0.06	0.06–0.08
40 ms delay	4.2	7.9

Table 1: Impact of the 40 ms delay on throughput

Throughput (R) is calculated using:

$$R = \frac{\text{Total Requests}}{\text{Total Seconds}} \quad \text{(in Requests/Second)}.$$

Before delay:

$$N = 100: \quad R = \frac{100}{0.06} \approx 1666.67 \text{ requests/s},$$

$$N = 200: \quad R = \frac{200}{0.08} = 2500 \text{ requests/s}.$$

After delay:

$$N = 100: \quad R = \frac{100}{4.2} \approx 23.8 \text{ requests/s } (\approx 70 \text{ times slower}),$$

$$N = 200: \quad R = \frac{200}{7.9} \approx 25.3 \text{ requests/s } (\approx 100 \text{ times slower}).$$

An important detail is that in the case of concurrent clients, the total time for each client was almost identical. Therefore the numbers in the table can be considered as an average, but with almost no fluctuation between individual clients.

Concurrency with N=100	**Single-threaded Rudy**	**Concurrent Rudy**
1 client	4.3	4.2
2 clients	7.9	4.2
3 clients	12.3	4.3

Table 2: Total time in seconds per Rudy implementation per concurrent connections.

Concurrency with N=200	**Single-threaded Rudy**	**Concurrent Rudy**
1 client	8.5	8.5
2 clients	16.1	8.5
3 clients	23.7	8.5

Table 3: Total time in seconds per Rudy implementation per concurrent connections.

Another interesting problem that occurred was in the delivery of files. When attempting to read files, the server behaved very unpredictably. There was a need to handle file errors gracefully, to prevent the server from shutting down abruptly. This was solved with error handling of the most common errors one might run into when attempting to read files: enoent (entity not found) and eaccess (forbidden). I extended the helper functions in the HTTP module to simplify the creation of these responses.

```
no_access(Body) ->
  "HTTP/1.1 403 Forbidden\r\n" ++ "\r\n" ++ Body.
not_found(Body) ->
  "HTTP/1.1 404 Not Found\r\n" ++ "\r\n" ++ Body.
```

3 Evaluation

There are a few points worth discussing regarding the observed results. First, in the single-thread implementation, the fact that the total time was increased for all clients in concurrent scenarios is very interesting. Let's examine a possible worst-case scenario. We have Client-A. and Client-B. They both begin sending requests to Single-Threaded Rudy (STR). A quite "dumb" way to handle this would be to handle all of Client-A's requests first, and then move on to Client-B's requests. Assuming STR took X amount of seconds to complete the requests, Client-A's benchmark should be completed in X seconds. Afterwards, Client-B's benchmark should also take ≈X seconds, and the overall benchmark for Client-B would result in ≈2X seconds.

For the example of N=100, we would expect Client-A to finish in ≈4 seconds, and Client-B in ≈8 seconds. However, **both** clients finished in ≈8 seconds, exposing an inefficiency, probably caused by Rudy trying to juggle requests between the two connections.

Another topic which could be investigated, is to what extent this approach can scale. Erlang processes are not operating system threads, in the same way that they exist in a language such as Java. This means that the scaling is not limited by the number of physical cores and threads on the host machine, as Erlang processes are extremely lightweight and the BEAM VM handles the load balancing and resource usage between them. This means that a single server could handle potentially tens, or hundreds of thousands of these processes.

4 Conclusions

This project is a very interesting adventure in the depths of Erlang, functional programming, and system design. It forced me to think like a systems engineer, not like an application developer, where the various system components are the "users". It also has provided me with a platform to develop highly scalable applications, given Erlang's design for scalability. There is definitely a lot more to be discovered, especially regarding system reliability and fault tolerance, which I am sure will be explored in future assignments.