Report 1: Rudy the HTTP Server

Lorenzo Deflorian

September 10, 2025

1 Introduction

The main goal of this assignment was to implement a simple HTTP server in Erlang. The server needed to accept socket connections, parse HTTP requests, and send responses. Another important aspect was gaining a deeper understanding of Erlang processes and how to use them to handle multiple clients concurrently. Finally, I evaluated the server's performance by comparing the different implementation approaches I explored.

2 Main problems and solutions

The first task was to create a simple HTTP server that could handle requests sequentially. This was straightforward using the built-in `gen_tcp` module for socket connections. After implementing the basic server, I benchmarked it to measure performance for 100 sequential requests.

```
(client@DORORO)1> test:bench(localhost, 8080).
Time Elapsed 174.338 ms
ok
```

Normalizing the elapsed time by the number of requests gives an average latency of 1.743 ms per request. Hence, the server can handle about 573 requests per second.

The next step was to add a small processing delay (40 ms) to simulate a more realistic scenario and see if the overhead given by the parsing of the request is significant.

```
(client@DORORO)1> test:bench(localhost, 8080).
Time Elapsed 4408.464 ms
ok
```

The artificial delay increases latency to 44 ms per request, showing that server overhead is negligible compared to processing time.

To improve performance, we implemented a concurrent approach that creates a new process for each incoming connection, allowing multiple requests to be handled in parallel. This raises the question: how lightweight are Erlang processes? To answer this, I implemented a concurrent version of the server to measure overhead from process creation.

Metric	**Time (ms)**
Average process creation time	0.015
Maximum process creation time	0.171
Minimum process creation time	0.006

Table 1: Process creation time statistics

Table 1 shows the average process creation time is about 0.015 ms, which is quite low. Therefore, we can create many processes without significant overhead.

3 Evaluation

3.1 First concurrent implementation

Let's evaluate the server's performance with multiple clients connected at the same time. The server was launched with:

```
spawn_rudy:start(8080).
```

and the clients with (100 clients in total):

```
test:bench_many(100, "localhost", 8080).
```

I got the following results:

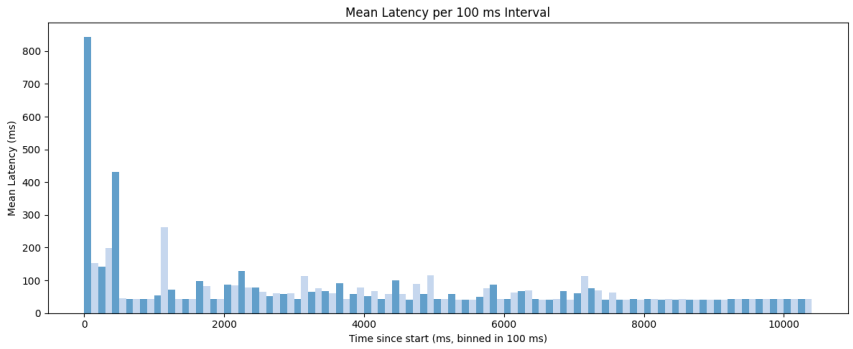

Figure 1: Latency Histogram

As shown, Erlang can handle many requests concurrently with good performance. The initial spike in latency is due to the first requests being processed; as the server warms up, latency stabilizes at a lower value.

Overall, even with a very simple implementation, Erlang proves to be a powerful tool for building concurrent servers.

3.2 Comparison between sequential and concurrent implementations

We next evaluate a more stable approach and compare it to the initial sequential implementation.

The `multi_rudy` implementation uses a pool of handlers to limit the number of concurrent processes and avoid overwhelming the system.

I benchmarked the sequential server using three clients and measured latency.

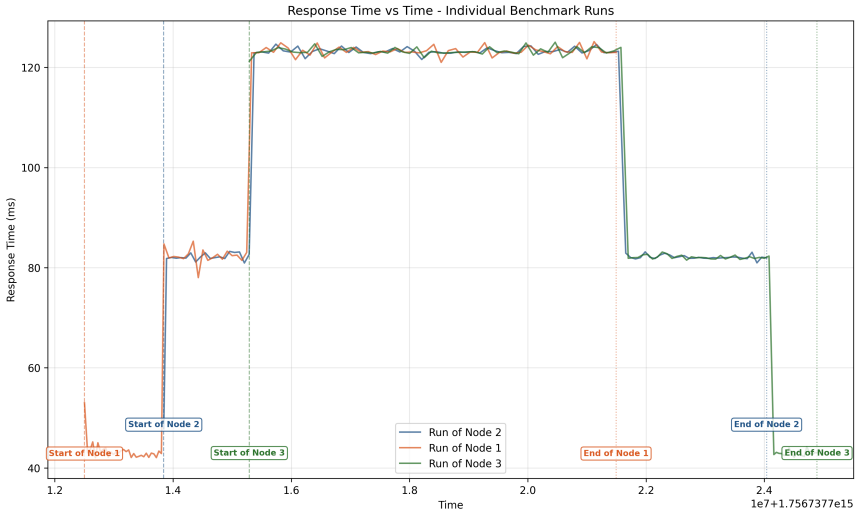

Figure 2: Latency Comparison (Sequential)

Figure 2 shows that latency increases roughly linearly with the number of connected clients, as expected. Each time a client finishes its batch of requests, latency drops because fewer clients are connected to the server.

We now evaluate how the concurrent version performs.

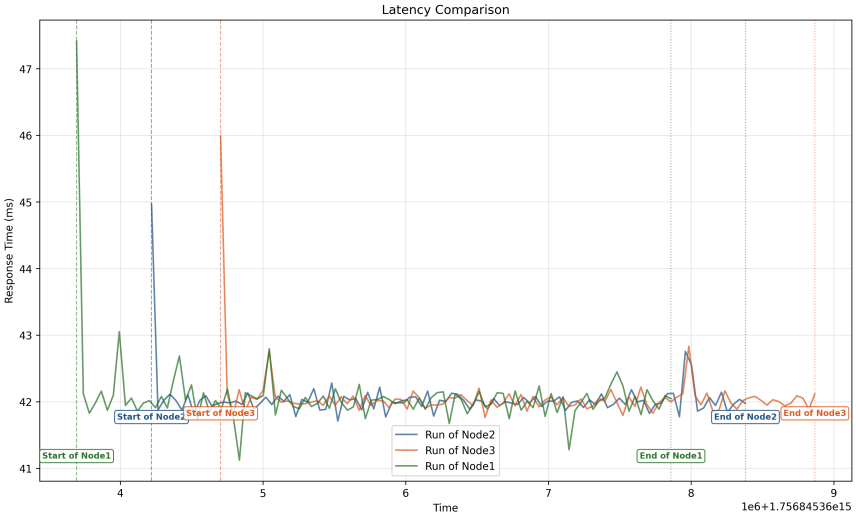

Figure 3: Latency Comparison (Concurrent)

The `multi_rudy` implementation performs significantly better than the sequential version: because each handler processes requests independently, overall latency is much lower. We have increased throughput!

4 Conclusions

This report explored the performance of a simple sequential server implementation in Erlang and compared it with a concurrent version using a pool of handlers. We found that the sequential implementation suffers increased latency as the number of connected clients grows, while the concurrent implementation handles multiple requests in parallel with much lower latency.

For further improvements, we could upgrade the pool management strategy to dynamically adjust the number of handlers based on current load. One manager process could monitor the busy handlers and spawn new ones as needed, or terminate idle ones to free resources.