Rudy - a small web server - Report HW1

Malte Berg

September 10, 2025

1 Introduction

In this homework assignment, the topics covered have been the following:

- Using sockets and a socket API for network communication
- Understanding the basic features of the HTTP protocol
- Programming in Erlang
- Concurrency in Erlang

The main topic related to this assignment would be the topic of *Networks and Interprocess Communication*, as the assignment shows how processes communicate with each other and over the network.

This is important, as this will serve as a good base for the remaining assignments, and is the building blocks for distributed programming. Without interprocess communication, the solutions become centralized.

2 Main problems and solutions

The first part of the assignment was to write the HTTP parser. The complete code for this was given in section 1 of the instructions.

After the parser was written, the next task was to write a program that would be an extremely rudimentary server, simply receiving a request, giving a reply and terminating.

Majority of the code was provided, with four places where code is missing to be filled in.

In task 2.1 the four places was filled with the following code:

- In the function `init/1`, the following: `handler(Listen),`
- In the function `handler/1`, the following: `request(Client),`

- In the function `request/1`, the following: `Request = http:parse_request(Str),`
- In the function `reply/1`, the following: `http:ok(URI).`

With these code snippets added, running the program and visiting the URL `http://localhost:8080/foo` in a web browser succesfully showed `/foo` as the body of the page.

In task 2.2 additional code for starting and stopping the server, as well as making it a server instead of a single serve. The function `handler/1` got an added recursive call to itself under the added code mentioned above so it called itself when it had served a client, to be able to continue serving.

For task 3, a benchmarking program was created, as well as a delay of 40 ms added in the reply function of the server simulating work being done.

After benchmarking, task 4 was started, specifically task 4.1 that regards multi-threading for increased throughput.

For this task, both the server code as well as the benchmarking code had to be rewritten to make testing the concurrent abilities possible. At first, the naive version described in the instructions were tested, spawning a new process for each request. While functional, to not be limited in the mentioned cases of multiple thousands of requests at once, the strategy of having a pool of workers were tested instead.

Struggles appeared when trying to create the pool of handlers from the `init` function, without having the port close prematurely. This problem was solved by adding a receive in the {`ok, Listen`} case in the init function that waits for a message to close the socket, with the following code:

```
receive
    stop ->
        gen_tcp:close(Listen)
end,
```

Along with this, the `stop` function was rewritten to close the socket in this manner, which sends the stop message to the process when called:

```
stop() ->
    whereis(rudy_conc) ! stop, ok.
```

To create the pool of handlers, the call to the `handler` function in the `init` function was replaced with a call to a new recursive function spawning a given amount of handlers. Together, these pieces of code made the server able to run multiple handlers listening to the same socket.

The benchmarking program was rewritten in a similar manner, spawning a given amount of runners, or clients, in a pool that will all connect to the server.

A problem appeared however, that the benchmark would instantly complete. This was similar to the problem that appeared in the server, having no "waiting" function that made sure all the processes were finished. This was fixed by creating a function `await_finished_runs/1` with the amount of clients as its argument. It's a recursive function that waits to receive a message `finished` from the clients that they have completed their run. The spawning of the runner had to be changed as well to the following:

```
spawn(fun() -> run(100, Host, Port), Proc ! finished end)
```

The part of the code that was added was `Proc ! finished` where Proc is an argument in the pool spawner that is called with self() as its value. This was done to be able to pass the message to the program process. This means the spawned process first does the run, and then signals being finished before ending.

Now, the benchmark computes how long it takes for all the processes to finish, which will help show whether there is an improvement in the throughput with a concurrent server.

3 Evaluation

Non-concurrent Server

Firstly, the amount of requests that could be served on the server with one process, with a delay, was between 23.5 to 24.5 requests per second.

When removing the 40 ms delay, the result shot up to values of between 2050-2400 requests per second.

This shows clearly that the delay adds a significant amount to the benchmarked time.

Looking at Table 1: When connecting with multiple machines, or in this case multiple processes on the same machine, the time on the benchmarks went up in quite a linear fashion, meaning the requests per second stayed approximately the same, with some normal fluctiation. With 10 simultaneous connections, a drop in the requests per second can be seen. This could be due to processing power being high and the machine not being able to handle the load for a prolonged time. A similar drop-off can be seen on the no-delay version of the program, with a speedup of the first 5 connections, probably accredited to the program warming up, and that most of the time for the measurements are spent for the socket connection rather than the work process, making the delay good to see the bigger picture.

Concurrent Server

Looking at Table 2 shows a clear difference in the amount of requests served per second when four handlers are working concurrently. As the amount of requests served per second goes up for each additional connection, and

# of Connections	With delay [req/s]	Without delay [req/s]
1	23.7	2115
2	24.4	4186
3	24.4	6723
4	24.1	7104
5	24.3	7614
10	19.44	939.6

Table 1: Benchmark of server with one handler

# of Connections	With delay [req/s]	Without delay [req/s]
1	23.8	2174
2	47.3	3932
3	71,3	6051
4	93,9	7458
5	97,3	7544
10	65,6	934.6

Table 2: Benchmark of server with four handlers in a pool

plateauing at one connection more than the amount of handlers, it is clear that the concurrent implementation of the server is functional.

4 Conclusions

This assignment was very useful, and gave great insight into how an Erlang program can be written to both function concurrently and sequentially. I have learned how to deal with problems that can occur when spawning processes, got a greater feeling for the syntax of Erlang, and managed to improve the given program without instructions with the help of the Erlang documentation, proving to be very informative.