Rudy: a small web server

Zainab Alsaadi

October 4, 2021

1 Introduction

This assignment is about building a web server, Rudy, and access it from other machines to understand the structure of a server process and use HTTP requests. In this assignment we learn to pass messages between programs that run on different machines, which is the core idea of distributed programming.

2 Main problems and solutions

For this assignment, the functional programming language Erlang had to be used. It was not difficult to follow the instructions of building the web server, but not having enough experience with functional programming could be a challenge for understanding the hole process. Some small encountered problems were i.a. forgetting to export functions before using them. Moreover, general understanding of recursion and function calls in Erlang where necessary for completing the rudy module.

3 Evaluation

We started first by implementing HTTP module. The functions were tested as they were implemented. Before doing that, it is important to understand how the HTTP protocol is constructed. In this case, our program works for version 1.0 or 1.1. We start with the implementation of a request as described in RFC 2616, ietf[1].

The method `request_line`is used to parse the http request. This is composed of `request_URI`and `http_version`methods. The URI is returned as a string and the version is for representing version information. Afterwards, the message header and body parsing methods were added. The program was tested with a request before moving on to the next step. running the program with the following request `"GET/index.htmlHTTP/1.1\r\nfoo34\r\n\r\nHello"` gives the following: {{get,"/index.html",v11},["foo 34"],"Hello"}

Next step was to implement rudy.erl, which consists of the methods `init\1, handler\1, reponse\1` and `reply\1`.These were filled with the missing lines to complete the functionality.

The added lines of code are commented in the code with '% :'. In order to implement these functions we need to understand the socket API procedures defined in gen_tcp library[2]. In the reply method, some Html code was added to return a proper reply. After adding the last parts, the program was tested by opening a browser and accessing the link `http://localhost:8080/foo`. The site could not be reached until the init function was called with the port number. The server terminates after one request. To let the server run until it is manually terminated, we let the function handler call itself recursively. It was also convenient to save the server as a registered process to make it possible for any process to find the server and send a message to it.

The following step was to use a benchmark program that can generate requests and measure the time it takes to receive the replies. the result showed that 100 requests could be made in $4239026ms \approx 4,2sec$, which means nearly 24 requests per second.

The delay that was added in the reply to simulate file-handling is shown in table 1 below.

Delay	Response time (μs)
40	4270120
400	4268544
4000	4282128

Table 1: Delay affect on the response time of the server

Running benchmarks on several machines/shells at the same time yields the results shown in 2.

Machine	Response time (μs)
Machine1	6890668
Machine2	8155839
Machine3	9047646

Table 2: accessing web server from several machines simultaneously

4 Increasing throughput

Concurrency was implemented by spawning a new process for each incoming request. The aim is to measure if this could increase the throughput of the server. In order to do this, we first need to modify the function that calls gen_tcp:accept, since here is where we accept an incoming request.

This means we start by modifying handler(). We spawn a process using spawn_link, which spawns a new process and creates a link between the calling process and the new process.

Redoing the experiment from 2, but this time with concurrency, We get the following results:

Machine	Response time (μs)
Machine1	4303038
Machine2	4290669
Machine3	4283653

Table 3: time to handle requests concurrently

5 Conclusions

We can see from 1 that artificial delay is not significant in the case of changing the delay time. That could mean that it disappear in the parsing overhead.
What makes delay time significant is, as we can see in 2 is accessing web server from several machines simultaneously.

From table 3, we see that it takes almost the same time to handle all three requests concurrently. Compared to sequential handling, that's an improvement.

References

[1] URL: `https://datatracker.ietf.org/doc/html/rfc2616`.

[2] URL: `https://erlang.org/doc/man/gen_tcp.html`.