Report 1: Rudy - A small web server

Emil Ståhl

September 6, 2019

1 Introduction

This report covers the implementation of a small web server written in the functional programming language named Erlang. The code base consists of three main files which are http.erl, rudy.erl and test.erl. Most of the code used was available in the lab manual, the only thing added is network setup and communication functions as well as some minor refactoring.

Erlang was developed to offer stability, fault-tolerance and concurrency built in. The main purpose of this work is to explore these features as well as the syntax and structure of developing in Erlang. Due to this features, Erlang is a great language to use in order to learn about distributed systems and the challenges that comes with such a system.

2 Main problems and solutions

The problems experienced during this work were of different nature, including syntactical, compiling and understanding the purpose of each function.

The syntax of Erlang offers a steep learning curve, so it can be hard to grasp in the beginning but when you get the basics it goes quite fast to learn more complex code. However, most of the time spent on problems was due to difficulties compiling the program. This was not clearly described in the manual so it required some research and experimentation to get it up and working. Lastly, understanding the flow of the program required some thoughtful thinking and analysis. To comment each row is a great way to get a deeper understanding.

2.4 Implement concurrency

To implement concurrency and be able to handle several requests in parallell the used solution was to spawn a new process to handle each incoming request. This was added in a different file named *rudy_par.erl* in order to be able to test the different implementations separately.

So instead of calling the method *request* in the *handler* method, a new process is spawned to handle each requests. The code used for this is the following:

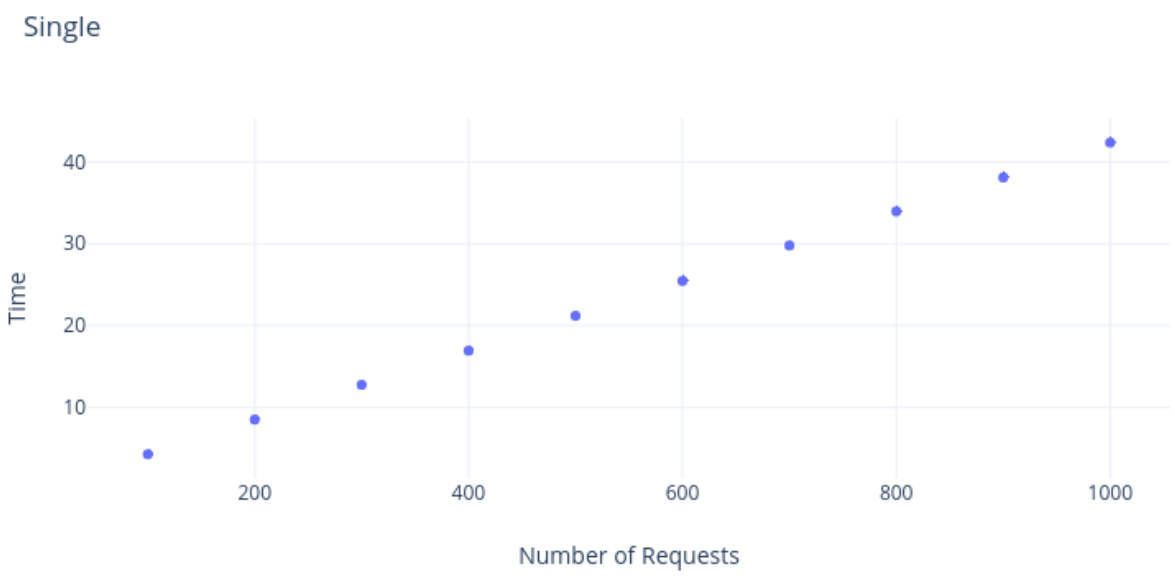
```
spawn_link(fun() -> request(X) end).
```

3 Evaluation

3.1 Sequential Single shell

When testing in a single shell to localhost on port 8080. We can see that the results is follows a linear structure starting from 4,2 seconds with 100 requests and stretching up to 42,3 seconds when performing a total of 1 000 requests.

Since we added 40 ms of fake work to each request, this leaves 4200 ms / 100 requests - 40 ms/request = 2 ms/request overhead. We can expect more overhead if testing over a real network.



Single shell

100	4.248385
200	8.493824
300	12.735432
400	16.914003
500	21.177075
600	25.431762
700	29.766177
800	33.946086
900	38.079024
1000	42.355775

3.2 Sequential Double shells

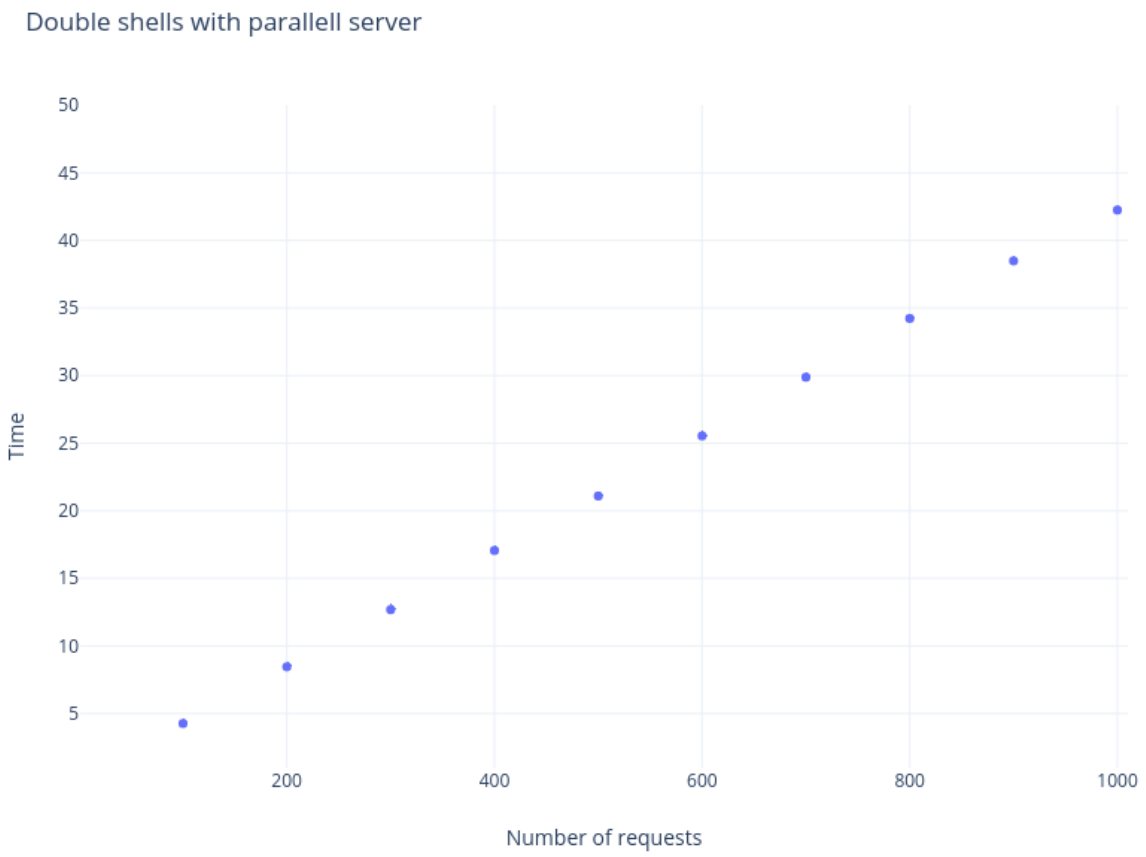
When testing in two different shells at the same time the results gets dramatically worse. The time it takes to do 100 requests now takes 7,8 seconds, while 1000 requests takes 81,7 seconds. This is roughly a 2x decrease in performance compared to the previous test with only one shell.

Double shells

100	7.883518
200	16.080910
300	24.269725
400	32.202469
500	40.743508
600	48.942772
700	57.141101
800	65.396381
900	73.581362
1000	81.742184

3.3 Parallell server

When testing in two different shells but with the parallell server, performance increases drastically compared to the sequential server. There is almost a 2x increase in performance. This is due to that a new process is spawned to handle each incoming request concurrently.



Double shells with parallell server

100	4.265608
200	8.457806
300	12.683237
400	17.062068
500	21.091839
600	25.538873
700	29.879968
800	34.224913
900	38.481922
1000	42.245030

3.4 Overview of results

We see that the results of the parallell server is almost exactly the same as the sequential server performed when tested in a single shell.

Overview of results

Requests	Sequential single shell	Sequential double shells	Parallell double shells
100	4.24s	7,88s	4.26s
500	21.17s	40.74s	21.09s
1000	42.35s	81.74s	42.24s

4 Conclusions

Although I have worked with functional programming before in Elixir, this was the first time I have done a bigger assignment in a functional language that actually does something useful. I have increased my ability to understand and write code in the Erlang syntax as well as using the built in libraries that handles sockets and TCP connections. I have also start using the IDE named Visual Studio Code instead of Eclipse which I used the last time I worked with functional programming. Using VS Code has brought huge improvements on how I work with my projects.

Regarding the results, the result of the benchmarks really shows the benefits of concurrent programming for increasing throughput by handling multiple clients for a service simultaneously. The parallel server approach does not have any benefits while only serving one user/shell. But when serving two users the performance is about 2x better than the sequential server.