Rudy – a small web server

Óttar Guðmundsson
7 September, 2017

1 Introduction

The first project of the course was to make a small web server which was composed of a custom made HTTP parser and usage of the gen_tcp Socket API from Erlang.

Not only was the project useful for learning and working with the basics of Erlang programming but also necessary understand how a basic server worked and the functions behind it. I also gained knowledge about the structure of HTTP protocols and how they should be used as they are a very important part of distributed systems.

2 Main problems and solutions

2.1 HTTP parser

The assignment started with a simple, yet a little bit confusing HTTP parser. Even though the first part was given on the assignment paper, it took some time to go through the Erlang syntax step by step. After getting a better feeling for how the language worked regarding pattern matching and recursive functions I was ready to continue to the next part.

2.2 Rudy

Implementing the server file itself, **rudy.erl**, was easier. Our professor gave us a mock-up code with a description on how the functions should work where he left in marks on what needed to fill in to make it work. Just by reading the text, the rest was easy.

The code consisted of 4 main functions

init(Port) – initializes a listening socket and pass it to the handler. Uses gen_tcp:listen.

handler(Listen) – Listens to incoming request from the socket . Uses gen_tcp:accept.

request(Client) – Receives the request from the client. If it is ok, we parse it with out http class and deliver the request to the reply. Uses gen_tcp:Recv.

reply(Request) – Creates a reply by the sent request. Uses the http class to simply return the url to the clients window.

This server was pretty straightforward and wasn't really impressive because it could only serve one request at a time before terminating. This was solved by our professor by calling a new handler after each request had been dealt with completely. That way the server could keep on accepting requests until it would fail in any way or be shut down.

Running the benchmark later in the project (chapter 3 Evaluation) was also problematic. I spent maybe about 20 minutes trying to figure out why it wasn't working as expected. After having a look at the benchmark file, **test.erl**, I noticed that it couldn't access a function in the **http.erl** called get. The problem was that I hadn't included the function in the -export method on top of the module. As soon as I changed that and compiled again everything worked as expected. The problem was related to my Erlang knowledge, not the solution itself.

2.3 HTTP parser (Optional Task)

We were given the option of implementing an optional task for extra bonus. At first I thought about doing to 4.1 (Increasing Throughput) but I was unsure how to "assign" threads when they were free versus spawning multiple in the beginning. I picked 4.2 (HTTP parsing) which seemed like an interesting option. To solve this, a better look at the HTTP/1.1 documentation [1] was needed as well as looking the the gen_tcp API [2].

My implementation was simple. The *request* function was changed to take in an extra parameter which was an empty *string* by default. In the Request itself, the substring "/r/n/r/n" was looked up in the received string. If it wasn't found, the function called itself again recursively with the parameter string and received string concated together. The problem with this solution is that for each recursion the scan for a substring always gets slower, but this can be improved if needed. With this implementation the server was able to receive message as long as 33254 letters.

```
    Recv = gen_tcp:recv(Client,0),
    case Recv of
{ok, Str} ->
      BodyLength = string:str(Str,"Content-length:"),
      HeadEnd = string:str(Str,"\r\n\r\n"),
      case HeadEnd of
    0 ->
        request(Client, FirstString ++ Str);
    HeadEnd ->
        Request = http:parse_request(FirstString ++ Str),
```

Image 1: Changes to the code

After consulting with the teacher assistant about my solution, he pointed out that this was a right solution, but asked how I would change it if the *Content-length* header would be included.
My solution was that I would get the length from the header and by doing so the code knew how long the body would have to be. By doing that, the server would know exactly how much of the body it should read and then close the connection.

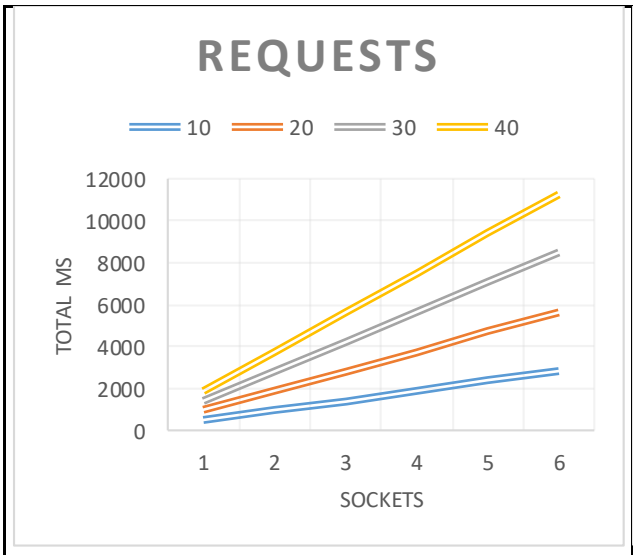
Another benefit of having the content header would be to measure the length of the body and compare it to the *Content-length* header. If it wouldn't match, the message should be discarded or resent because something got lost on the way or it was simply corrupted.

Image 2: a huge request displayed.

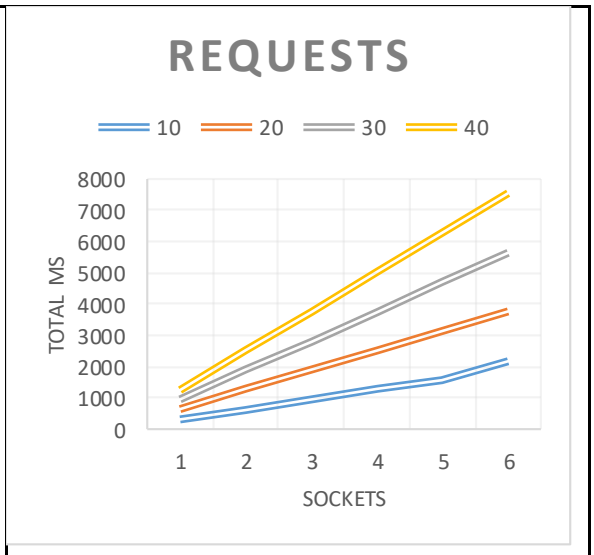
[1] https://www.ietf.org/rfc/rfc2616.txt
[2] http://erlang.org/doc/man/gen_tcp.html

3 Evaluation

Now some testing was needed to find out the server's performance with a given code, **test.erl** which was used to for a virtual benchmark. A 40 ms delay was added before the http parsing to simulate some action or calculation server side. Calling a simple given function to test the server, *test:bench(localhost,8080,1,10).* took *496 ms* to run which means that each request takes about 49.6 ms to complete. That gives us the assumption that the server can serve 20.161 request per second. This indicates that the delay we added to the *request* function does matter so the http parsing is not an issue and the tcp connection time is not a problem. After running the former benchmark, I decided to some more testing to see if the server would behave any different by adding more requests or, running the server with more sockets or adding to the delay.



Graph 1: Server tested with 40ms delay



Graph 2: Server tested with 20ms delay

It seemed that the result gives a linear pattern so the performance does improve linearly. The data for the tables can be found at the end of this report called *5 Data.*

Running several benchmarks test at the same time on the server with 40 ms delay gave me the following results.

Number of tests running concurrently	Benchmark time (ms)
1	1875
2	2547
3	3390

This is also a linear pattern as expected.

4 Conclusions

The project refreshed my memory and skills in functional programming. I've never written in Erlang before so understanding the *receive* function and other methods proved useful. I'm not really used to code with sockets and server on such a basic level but it is good to know what is hidden behind the technology. Doing the optional task was also encouraging to step out of the comfort zone and doing

5.Data

Benchmarking with 40 ms sleep time				
	10	20	30	40
1	496	984	1406	1875
2	936	1874	2811	3749
3	1406	2812	4234	5624
4	1890	3765	5641	7515
5	2358	4702	7061	9437
6	2828	5625	8485	11250

Table 1: Data for 40ms

Benchmarking with 20 ms sleep time				
	10	20	30	40
1	312	641	953	1250
2	624	1265	1874	2499
3	952	1889	2827	3749
4	1265	2515	3781	5015
5	1577	3140	4702	6264
6	2172	3766	5641	7547

Table 2: Data for 20ms