Homework Report 1: Rudy - A small web server

Joakim Öberg

September 2, 2020

1 Introduction

In this lab a small web server is implemented to handle simple HTTP requests and responses. The main goal is to better understand the structure of a server process, the HTTP protocol and get some hands on experience with a socket API. The programming language in use is Erlang.
The main files in this report is test.erl, rudy.erl and http.erl. For the optional task(increasing throughput) the files test_parallell.erl and rudy_parallell.erl are also used. Most of the code was given in the report.

2 Main problems and solutions

The lab was well described with instructions easy to follow. A functional language can be a bit tricky to read sometimes with all the base cases and hops between functions, but with a good understanding how the incomming argument for a function looks like and what one want to fulfill is gets a lot easier.

2.1 Increasing throughput(optional task)

The file rudy_parallell.erl is based on the file rudy4.erl that was given in the laboration.

2.1.1 Difference parallell and sequential server

The main diffecence in rudy_parallell.erl and rudy.erl is what the function init calls. In rudy.erl init/1 calls handler/1 that with the listening socket waits for a connection request to accept. The request is handled in request/1 and after that handler/1 is recursivly called to listen for a new connection request. This makes the program sequential.

```
handler(Listen) ->
    case gen_tcp:accept(Listen) of
```

```
        {ok, Client} ->
            request(Client),
            handler(Listen);

        {error, _Error} ->
            error
end.
```

In rudy_parallell.erl init/2 calls handlers/2 that takes the arguments listening socket and N(number of processes to run). For each N a new process is spawned and each of the processes are waiting for a connection request to accept in the same way as handler/1 in rudy.erl. Each of the processes run sequentially in the same way as in rudy.erl, but all of them are listening to the same listening socket.

```
handlers(Listen, N) ->
    case N of
0 ->
    ok;
N ->
    spawn(fun() -> handler(Listen, N) end),
    handlers(Listen, N-1)
    end.
```

3 Evaluation

The tests for both the sequential and the parallell implementation include 100, 500 and 1000 requests to the server. In **3.1** a sequential server is used, first tested from one single shell and then from two shells simultaniously. In **3.2** we use a parallell server and a test program test_parallell.erl that allows us to run clients cuncurrently. Each test is performed twice for a better time estimate and the execution times is measured in milliseconds.

3.1 Sequential server - Results

The program test.erl was used in this test. To start the server, *rudy:start(8080).* is executed in one shell. A client is executing *test:bench(Host,Port)* from another shell.

3.1.1 One client

When testing with one client the command *test:bench(localhost,8080).* is executed in one shell. Each test is run twice to get a better result.

One client		
Req	**#1 Time (ms)**	**#2 Time (ms)**
100	4 292	4 255
500	21 197	21 170
1000	42 488	42 404

The sequential server can with one client handle about 25 requests per second.

3.1.2 Two clients

When testing two clients the command *test:bench(localhost,8080).* is executed in two different shells, manually, as simultaniously as possible. The time from the two shells is both presented below, each test is run twice to get a better result.

Two clients				
Req	**Time (ms)**			
	#1 shell1	**#1 shell2**	**#2 shell1**	**#2 shell2**
100	7 891	7 906	7 984	7 959
500	40 994	40 968	41 172	41 150
1000	82 415	82 458	82 685	82 647

When two clients request a sequential server concurrently the server tries to execute them at the same time, making it execute the first client and the second client a little bit at a time resulting in that the execution time doubles for each of the clients, however the requests per second is still about 25.

3.2 Increasing throughput(optional task) - Results

To start the server we run *rudy_parallell:start(8080,2).* in one shell. To test the parallell server implementation we use the program test_parallell.erl and execute the command *test_parallell:bench(localhost,8080,#clients,Req).* in another shell. The tests executed, with the server running two processes, includes 100, 500 and 1000 requests from first 2 and then 4 cuncurrent running clients. Each test is run twice to get a better estimated time.

Parallell server - 2 running processes						
	Time (ms)					
Req	**#1,1client**	**#2,1client**	**#1,2client**	**#2,2client**	**#1,4client**	**#2,4client**
100	4 199	4 186	4 235	4 222	8 202	8 203
500	20 939	20 966	21 143	21 144	41 003	41 919
1000	42 069	41 811	41 919	42 242	82 647	82 006

When running a parallell server with 2 processes the number of requests

handeled per second depends on how many of the processes that are running. Each of the processes can handle ablout 25 requests per second. When only one client does the requests only one process is started. When two clients makes requests the server start two processes, increasing the servers requests per second to about 50.

3.3 Comparing results

In this table the times from each result is added and divided by (the number of tests multiplied by the number of shells), to get an average time for each test. The times are in milli seconds rounded without decimals.

$$average\ time = \frac{all\ times\ from\ test\ added}{\#tests * \#shells}$$

Comparing results					
	Time (ms)				
	Sequential		**Parallell - 2 running processes**		
Req	**One shell**	**Two shells**	**One client**	**Two clients**	**Four clients**
100	4 274	7 935	4 193	4 229	8 203
500	21 183	41 071	20 953	21 144	41 461
1000	42 446	82 551	41 940	42 081	82 327

In the table we can see that a sequential server has about the same amount of requests per second(25) independent of how many clients that does the requests, but the time per client gets multiplied by the total number of clients. The parallell server can serv about 25 requests per second and running process, making it possible to serv up to about 50 requests per second when 2 processes is running.

4 Conclusions

This laboration helped me to gain knowledge to understand listening sockets and give me hands on experience to use the socket API gen_tcp. Last year I did a course in elixir, a functional language that originates from Erlang, the knowledge I gained in elixir helped me a lot in this laboration. The results shows that a parallell server implementation with 2 running processes is not faster than a sequential when only one client/ shell is used. However when 2 parallell processes are running on the server twice the amount of clients can be handeled in the same time as the sequential server can handle.