Report 2: Rudy homework report

Douglas Hammarstam

September 9, 2022

1 Introduction

In this project, a simple web server has been implemented.

In this seminar, the main topic covered was the HTTP-protocol and how to parse it using Erlang. Furthermore, the seminar covered how to use the gen_tcp package in order to receive network HTTP-requests and handle the requests by using above mentioned parsing of the HTTP request.

This is important in order to get an understanding of how to use Erlang in order to efficiently parse HTTP-requests (strings) using recursion.

It is also important in order to understand basic Erlang features like recursion, pattern-matching aswell as spawning processes in order to distribute an application on multiple threads.

2 Main problems and solutions

In order to build a small web server i Erlang we use the gen_tcp package which has an API for listening to incoming HTTP-requests on a given port.

```
  gen-tcp:listen(Port, Opt)
```

The package also provides the possibility to accept incoming TCPconnections.

```
  gen_tcp:accept(Listen)
```

Where "Listen" is port to listen on.

After that, we can handle each request by spawning a new process for each request. This makes the program more responsive to many requests at the same time, since we are able to handle requests concurrently.

```
spawn(fun() -> request(Client) end),
```

We also try limiting the amount of processes being created in order to create a upper bound on the amount of threads being created by the computer

By creating a worker pool of processes that keep hold of their own index, in order to distribute the workload across them.

In order to handle these requests. We need to do some parsing of the http-request.

The first problem in the assignment was to parse the incoming HTTP-request. This was done by dividing the problem into different sections; Firstly, parsing the method used in the request (GET). Then the URL is received by parsing each character after the space (32) character after the GET string until we reach another space character (32). Secondly, we get the HTTP-version (v10 or v11) by simply pattern matching the string to either of those values. Thirdly, we parse a line-break character followed by the headers, which we also recursively parse. Fourthly we parse the body where we assume that the rest of the request is made up of the body, which is not always the case in a real world scenario.

3 Evaluation

In order to evaluate the program, I used the test.erl and the test2.erl files.

The test.erl file does not create multiple nodes (clients). The test2.erl file does create multiple nodes (clients).

4 Conclusions

Using the test.erl with multiple threads does not improve performance since there are not multiple clients

The test2.erl files uses multiple nodes and therefore using more threads improves the performance.

I used 5 nodes and 100 requests per node.

Threads	Response time (ms)
1	40531
2	20259
4	10133
8	8279

The program ran the slowest with one thread, and the fastest with 8 or more threads. This is probably because my computer has only 8 threads.

Creating a lot of threads did not impact performance since using unlimited threads did not have any impact on performance, it performs as good as 8 threads. Going from one to two threads produces almost twice the performance. Going from two to four threads produces also almost twice the performance. Going from four to eight threads produces 15 percent more performance. Using more than eight threads does not impact performance.

I have learn't how to use the gen_tcp package to handle HTTP-requests. I have also learn't how to parse a request using recursion. I have also learn't that using more processes for handling multiple client connecting makes a web-server more scalable (handle more requests).