Report 1: Rudy

Abyel Tesfay

2021-09-15

1 Introduction

The following report intends to describe the students implementation and benchmark of the *Rudy* web server in Erlang. The purpose with this homework is to let the student familiarize with the Socket API, the structure of a server process and the basics of the HTTP protocol. The results of the benchmark are presented in a later section. The report also covers some possible improvements for the *rudy* server and how they can be made.

2 Rudy

The Rudy server makes use of the *gen_tcp* library, which provides an interface for Erlang processes to communicate with TCP/IP sockets. The following files were used:

- *http.erl*: A HTTP parser that parses incoming requests. The parser returns a tuple containing the following the request line, a list of headers and the body (if there is any).
- *rudy.erl*: A server that opens a listening socket, waits for incoming connections, reads and parses the request, then reply with an 200 OK response. For this the program is structured into several functions, each taking care of a specific part of the whole process. The reader is referred to *rudy.erl* for implementation details.

3 Benchmarks

For the benchmarks, the given benchmark files *test.erl* and *test2.erl* were used. which are available in the assignment page. The purpose with the benchmark is to measure how long it takes to receive a set of requests, and the amount requests the server can handle per second. Note that all benchmarks were conducted in the same machine. The benchmarks consists of the following tests:

- One client making several requests to a server.
- Multiple clients making requests to a single-threaded server

4 Evaluation

In the benchmark, the client thread generates multiple HTTP requests to the *rudy* server while measuring the time it takes to receive all responses. The amount requests N made in this test is 100. Since the assignment introduces an artificial delay (40 ms) for the server, benchmarks were made with both the delay and without. The results are presented in Figure 1.

Client	**Delay**	**Time (s)**	**Request handle rate (N/s)**
One client	No	0,0987	1013,038
One client	Yes	5,5307	18,081
Multiple clients	No	0,0440	2271,076
Multiple clients	Yes	4,6885	21,329

Table 1: The type of client and server, as well as average time and request handle rate

Based on the figure, one can notice the significance of the artificial delay. Without the delay the client is able to send at a rate of 1013 requests/second, compared to only 18 requests/second with delay. Handling a few requests might make the difference small, but as the server has to handle more requests more time is spent on the artificial delay. When running multiple clients (each client process sends 25 requests each), the amount time becomes halved. With the artificial delay however the same problem is encountered, more time is spent on the delay rather than serving requests.

A possible improvement is to make the server multi-threaded. This improvement makes use of the server multiple cores, which may be idle. This is done by letting the server spawn a number of threads (or processes in Erlang) in which each thread listens on the same socket. Each thread accepts any incoming requests, services them and returns to listening on the socket. Through this the server increases its throughput. As for scalability, you could have a controlling process which spawn or kills threads depending on the flow of requests, a higher flow means the controller will spawn more threads to help the existing ones.

5 Optional tasks

As an optional task, the task 4.1 was implemented by using the *rudy4.erl* file. The benchmarks here also measure the time to receive 100 responses,

and the *request handle* rate (s). The table 2 was generated to visualise throughput.

Clients x Requests	**Time (s)**	**Request handle rate (N/s)**
1 x 100	5,002	19,992
2 x 50	2,509	39,857
4 x 25	1,341	74,571
5 x 20	1,031	96,993
10 x 10	0,538	185,874
20 x 5	0,278	359,712
25 x 4	0,232	431,034
50 x 2	0,125	800
100 x 1	0,076	1315,789

Table 2: The results of client X requests sent (per client), average time and request handle rate (s)

Given the table, one can see that the throughput increases as more client processes send requests to the server in parallel. In reality, thousands of unique clients will request service from the web server simultaneously. In order to cope with this the server can spawn a pool of handlers in which they all sit and listen on the same socket, until they can serve a client. The Erlang language is well designed for massive concurrency, this lets the *rudy* server spawn up to 10000 processes (though the amount of cores is still limited).