Report 1: Rudy Server Assignment

Addi Djikic – addi@kth.se

September 13, 2017

1 Introduction

For this assignment we built a webserver and a HTTP parser. First the parser was built and then the Rudy server with a socket-API. The server opens a port of our choice and waits for the request, after received and parsed the it outputs a reply in a web-browser. The main topic for this assignment was to cover a simple HTTP-protocol and how it interacts with a simple server. This topic is important in the aspects of distributed system due to servers are used all over the world for every day use to handle and deliver contents of different kind. Servers are nodes that all are connected to the world wide web and handles systems distributed over the world.

2 Main problems and solutions

The first task was to only use one server and one client, where the server waits on a different machine for the request, replies, and later terminates. This was achieved by using socket API and using the `gen_tcp` library in Erlang, and we got a reply at the page `http://localhost:8080/foo` successfully by using seperate terminals to simulate different machines. However, a server should be able to wait for several request and not terminate after one. This was easily fixed by modifying the handler and make it call itself recursively again. Also a start and stop function was added so that one could easily inject when the Rudy-server should terminate and stop waiting for requests.

A problem with only using one-server-one-client is the lack of request handling in real life, when we want several clients or servers. The time-complexity would probably grow linearly as *O(n)* if the response time is constant and if several clients wants to send multiple request. We will investigate this in the evaluation. But a solution for this is "multi-threading" the problem, where the solutions is provided from the course and we can use this as:

```
start(Port, N) ->
    register(rudy4, spawn(fun() -> init(Port, N) end)).
```

From the Rudy4 program where N is number of concurrent listeners we want parallel bind to the same port, and from the test we use the bench marking test as:

```
bench(Host, Port, C, N) ->
```

Where we set how much clients there are and how many requests each they want to provide to Rudy4.

3 Evaluation

To evaluate the problem we will investigate the response time by looking at the ratio between the number of requests on different amount of listeners and clients. The bench test was changed so that we can choose how many requests we want to test as:

```
bench(Host, Port, Requests) ->
```

As we can see in Fig. 1, the time complexity, as predicted, grows linearly as we add more request from one client to one server but with a 40 ms artificial delay to simulate file handling that we added.

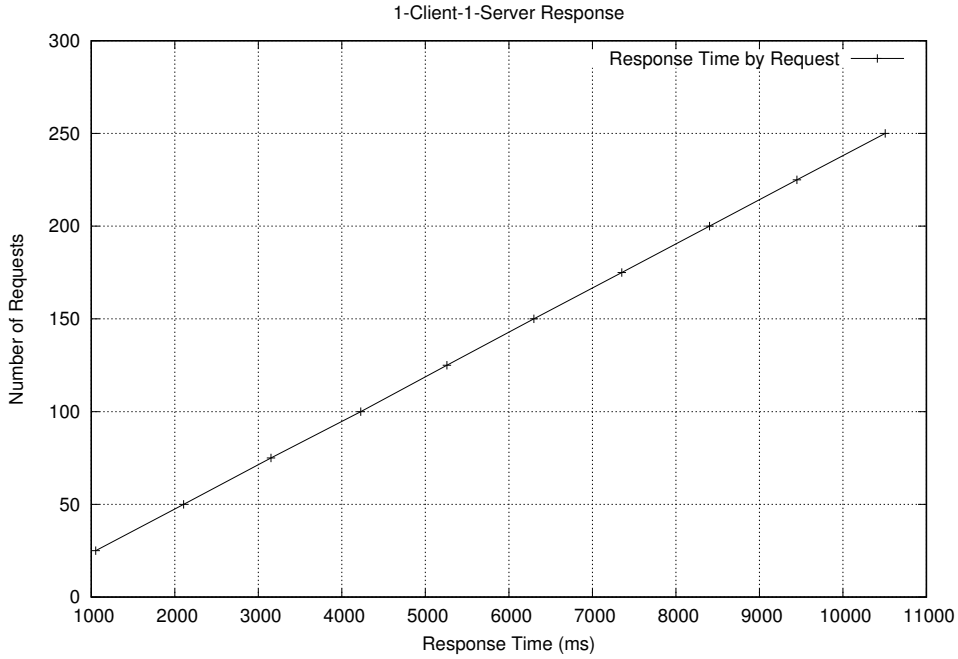

Figure 1: Visualization of the response time compared number of request when using one client and one server with 40ms artificial delay

When the 40 ms delay is removed we see a change in the response time, which can be visualized in Fig. 2. We observe that delay had an impact and that the response time is significantly faster when the delay is removed. We could serve around 1500 requests per second, but could vary without delay (but depends somewhat how powerful computer processors you have) and around 25 requests per second with delay. But a more detailed table of some requests per second without delay can be seen in the Table 1

Now, the `rudy4` and `test2` programs were provided from the course ID2201 to handle the problem with multi-threading server and when we use several clients. This will handle the problem when we for example have clients that want to send several requests each, this will cause a problem and stack the response time if we lack listeners available. As it can be seen in the simple Table 2, using for example one client and one listener yields the same response time as with four of each. But using four clients on only one listeners gives a response time that is four times as large, this is due to that every client wants to send 100 requests each, and therefore they are stacked.

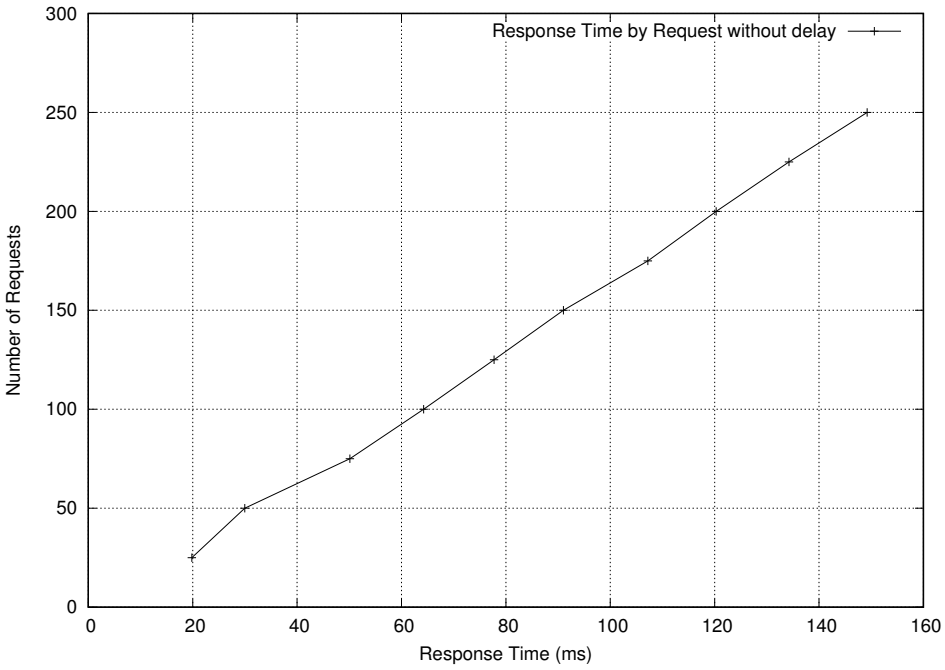

Figure 2: Visualization of the response time compared number of request when using one client and one server with no added artificial delay

Requests	Response Time (ms)	Request/Second
25	19.832	1260,58
75	50.101	1496,97
125	77.745	1607,82
175	107.201	1632,44
225	134.237	1676,13

Table 1: Showing how many requests per second we can serve without any artificial delay with one client and one server

Requests	Clients	Listeners	Time (ms)
100	1	1	4203
100	1	4	4200
100	4	1	16468
100	4	4	4236

Table 2: Comparing the response time with `rudy4` and `test2` by changing the number of clients and servers with the same amount of requests

4 Conclusions

We have studied how a simple HTTP-protocol works along with a simple socket-server and how important it is to use concurrency in the aspects of distributed systems and the world wide web. We saw how clients and server delays can cause a great impact with a

thousand request difference. For example on how the time stacks if we lack listeners for several clients that want to send multiple requests each. This is very useful to know due to we have to take into consideration how computing power and number of concurrent listeners can change. If a listener is not ready for a client, we saw in the evaluation how greatly the response times can increase. This is not optimal in real life, and we can see the absolute importance of concurrency. Also we saw the value and importance of functional programming in such a language as Erlang.