Report 1: Rudy

Fabrizio Demaria

October 1, 2014

1 Introduction

The main task for this laboratory was to build a small web server in Erlang. This requires to adopt procedures for using a socket API, define the structure of a server process and use the HTTP protocol.

2 Main problems and solutions

The first step consisted in developing a Erlang module capable of parsing a basic HTTP request. In particular we implemented the parsing of a HTTP *get* request, consisting of a request line, optionals headers and body. Afterwards we implemented a very simple reply function (*200 OK* status code) and the four procedures needed by our Rudy server.

The first version of this implementation was able to handle a single request before terminating. We modified the *handler* procedure to recursively listen to new connections after serving a request and we created two functions *start(Port)* and *stop()* to start and stop the server when needed.

```
handler(Listen) ->
   case gen_tcp:accept(Listen) of
      {ok, Client} ->
         request(Client),
         handler(Listen);
      {error, Error} ->
         error
   end.
```

With all the code ready and running we performed some benchmarks, described in the next section.

3 Evaluation

We run benchmarks to evaluate the response time of our server, introducing different delays (calling a sleep function in the *reply* procedure) in order to emulate some computational activities at server's side. Moreover we could set how many requests to include in a single run from client to server. Up to this point our server doesn't implement multithreading, thus all the requests are processed sequentially. All the different configurations of introduced delay and number of requestes per run have been tested on two different machines for server and client, and the values have been calculated as an average of ten measurements. It is important to underline that results strongly depend on the network status and machines' configuration.

	0 ms	20 ms	40 ms	80 ms	160 ms	300 ms
1 Request	0.0155	0.0375	0.0595	0.105	0.1813	0.3249
10 Requestes	0.1061	0.365	0.6425	0.9905	1.8405	3.1805
100 Requestes	1.6469	4.2813	6.2343	10.5937	19.1499	34.0749

Table 1: Mean response time in seconds (average out of ten measurements); each column corresponds to a differend value of introduced delay.

From Table 1 and Figure 1 it is possible to state that the response time increases almost linearly with the introduced delay for all of the three configurations: 1, 10 and 100 requests per run. The introduced delay affects every single request in the running processes and for this reason the response time increases linearly with the number of requests per run.

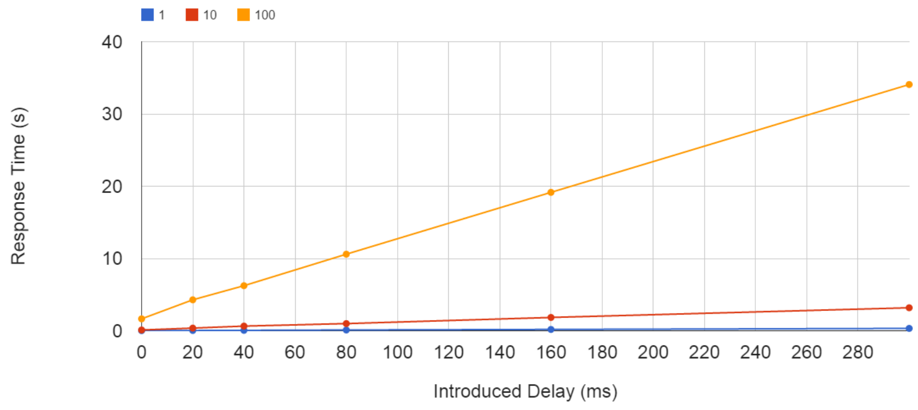

Figure 1: Comparison chart representing results for runs with 1, 10, 100 requests.

The requests per seconds (Table 2) have been calculated using the values of Table 1 and the visual result is shown in Figure 2.

	0 ms	20 ms	40 ms	80 ms	160 ms	300 ms
1 Request	64.1222	26.5991	16.7799	9.4344	5.5128	3.0769
10 Requestes	94.1662	27.3528	15.5619	10.0949	5.4330	3.1440
100 Requestes	60.7166	23.3568	16.0400	9.4394	5.2219	2.9347

Table 2: Mean number of requests served per second (average out of ten measurements).

The trends of the three lines in Figure 2 are almost identical because our server elaborates requests sequentially: for this reason there is no difference in sending multiple requests per run or sending a series of single request runs.

Moreover, we can state that the artificial delay in the order of tens of milliseconds is significant in the overall response time.

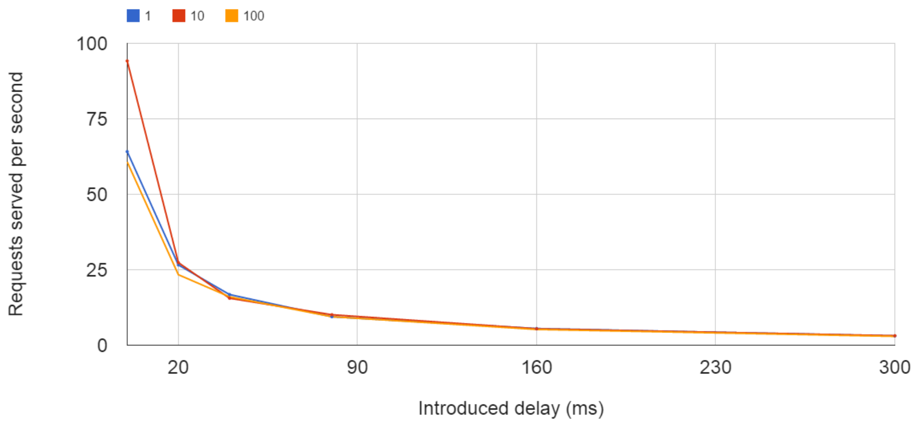

Figure 2: Comparison chart about the requests served per second with 1, 10, 100 requests per run.

Figure 3 shows performance comparison when executing tests on the same machine (localhost) and on different machines through Wi-Fi Internet connection. It is noticeable that the trend is the same but the response time is 10 to 20 % faster when server and client are on the same machine (testing with 100 requests per run and 40 ms delay).

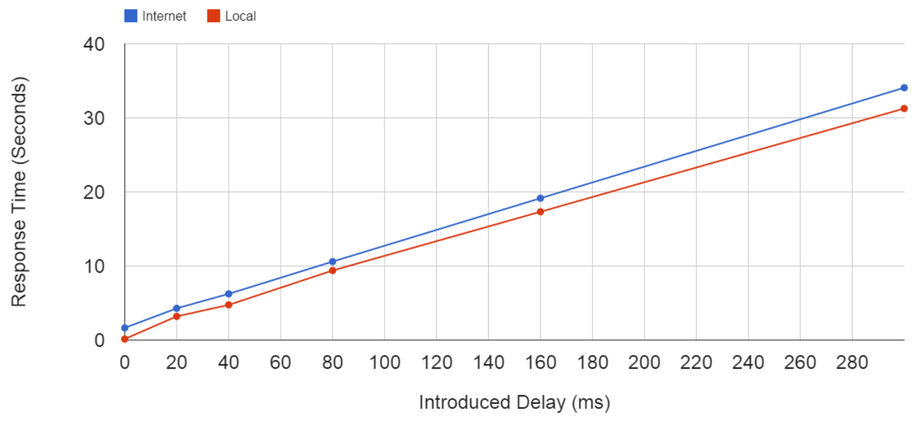

Figure 3: Comparison chart between local and network requests (100 requests per run with 40 ms delay).

As a last test we tried to execute runs from two different client simultaneously on the same server. This test was performed on three different machines connected through Wi-Fi Internet connection. In Figure 4 data have been evaluated executing ten runs with 100 requests per run and 40 ms of introduced delay. The server executes the simultaneous incoming requests independently from the client they belong to and for this reason the two clients conclude their operations almost within the same amount of seconds, but it requires more time

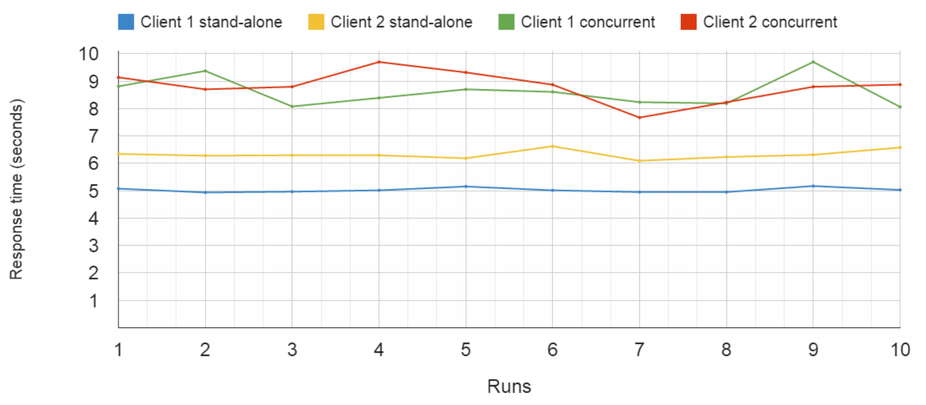

Figure 4: Testing multiple clients issuing requests simultaneously (green and red lines) and independently (blue and yellow line).

than the case of a stand-alone run on the server. The time does not double when two clients are sending requests, but the increase of time is in the range of 60-70%. The server is not multithreaded and the processing of the requests happens in sequential fashion, but we can speculate that the queuing system is able to feed the server more rapidly when multiple clients are sending requests at the same rate. It is possible to notice that Client 1 is faster than Client 2 in the stand-alone runs: our results strongly depend on the configuration of the machines and the network status.

4 Conclusions

The report was useful to improve Erlang programming skills and understand basic sequential client-server structures. This report is an important starting point upon which we will develop improvements like: multithreading, file transfer and robustness.