Rudy, a small web server

Anestos Nikolaos Ektoras

September 16, 2014

1 Introduction

In this exercise, we implement a small web server in Erlang.

The goal is to understand the HTTP protocol, how a simple web server handles requests and how Erlang communicates with sockets (in this case we use gen_tcp module which provides functions for communicating with sockets using TCP/IP protocol).

In the seminar we will discuss the following topics:

- How a server should behave in a more real-world environment, and ways to improve the throughput of the server handling requests concurrently.
- How can we make sure that the HTTP request is complete.
- How to deliver files.
- Better ways to terminate the server.

2 Main problems and solutions

After the initial setup of the HTTP request and response (http module) we create the server which will listen for an incoming connection in a given port, does the HTTP request and delivers the reply. This implementation will stop listening to the socket after the connection is handled. An easy way to make the server run more than one times is to add a recursive call to handler/1 function.

We did several tests with and without artificial delay, either in the same computer or in WLAN with my classmates, which are reported in the next section.

3 File serving

I was able to implement a simple file serving by changing the reply/1 function.

```
reply({{get, URI, _}, _, _}) ->
        Path = string:substr(URI, 2),
        {_, File} = file:read_file(Path),
        http:ok(File).
```

Getting the URI, we remove the first slash and get the path of the file we need to serve. We get the contents of the file with file:read_file/1 and reply to the client using http:ok/1 and the contents of the File as the body of the response.

4 Evaluation

Running the benchmark program with different scenarios gives us the following results.

Network	Average Response time (μs)	# of requests/s
Localhost	845	1183
WLAN	7.600	131

Table 1: benchmark doing 1 request per run without delay

Network	Average Response time (μs)	# of requests/s
Localhost	43.496	2299
WLAN	1.283.778	79
2 computers running the benchmark simultaneously	1.148.091	87

Table 2: benchmark doing 100 requests per run without delay

Network	Average Response time (μs)	# of requests/s
Localhost	4.2143.954	23
WLAN	4.969.644	20

Table 3: benchmark doing 100 requests per run with 40ms delay

5 Conclusions

It is clear that introducing an artificial delay to simulate file serving or some other server function, limits the number of request it can handle dramatically. Thus another server architecture is needed in order to simulate real life scenarios.