Report 1: Rudy Seminar Report

Alberto Lorente Leal

September 8, 2011

1 Introduction

In this seminar, we cover how to handle socket connections using TCP. This is a basic and important tool for Distributed Systems becauses this kind of Systems need to communicate with each node or machine remotely in order to receive and send information. Therefore it is important to know and handle this tool as it will surely be used a lot during the development of distributed systems.

The following topics covered with this seminar are as follows:

- We see how a TCP connections are handled in Erlang and also we see the procedures required for a connection (Host, ports involved in the communication) and also the steps needed to handle a TCP connection.

-We get to know how HTTP requests are used and how they are implemented from the RFC of HTTP. It gives us an idea of how we transform the theory, in this case from the RFC into a functional tool.

-The structure of a server, a program/module that primarily is some kind of loop where it constantly listens for incoming transmissions from the port it is assigned to. If an incoming transmission comes he starts to handle it and depending on the request, it responds with a certain reply depending on the request from the client.

2 Main problems and solutions

2.1 Problems during the development of the solution:

There where not many mayor problems during the development of the solution but i would remark the following issues:

-At first getting used to Erlang was one main issue. This is due for being my first contact with functional programming with this course and coming the world of Object Oriented Programming makes it difficult to get used to Erlang. In the end, with the help of the different guides found in the

web for example *www.learnyousomeerlang.com* and the book *Programming Erlang* from the pragmatic bookshelf, I managed to familiarize with Erlang.

2.2 Adding missing pieces of the server:

In the *init(port)* function, its job was to pass the running socket to the handler so we added in the following:

```
{ok, Listen} ->
handler(Listen),
gen_tcp:close(Listen),
ok;
```

In the *handler(Listen)* function it should listen the incoming connection and pass the request, also we had to modify the server so it wouldn't stop after handling the first request so we added a call of the Handler function so it recursively call it self like a loop.

```
{ok, Client} ->
request(Client),
handler(Listen);
```

In the *request(Client)* function as described in the assignment should get the request and parse it. It then sends the reply.

```
{ok, Str} ->
Request = http:parse_request(Str),
Response = reply(Request),
gen_tcp:send(Client, Response);
```

The reply should send a correct HTTP reply so we do the following in the *reply(Request)* function:

```
reply({{get, URI, _}, _, _}) ->
timer:sleep(40),
http:ok(URI).
```

3 Evaluation

Using the benchmark test module provided I did some tests on my laptop. We should have in consideration the specifications of the machine running the test as it is an 11 inch laptop. The CPU is an intel SU7300 @1,3Ghz which is OC @1,7Ghz and it has 4 gigabytes of DDR3 ram at 800mhz.

With this in mind I ran the default test which generates 100 requests to the server connecting to the "localhost" address using the port number 8080. The following results where print in the Erlang Shell:

```
(dde66_alberto_c97122_erlide@alberto-M11X)5> server:start(8080).
true
(dde66_alberto_c97122_erlide@alberto-M11X)6> test:bench("localhost",8080).
40794000
```

This number is the time taken to completely handle the 100 requests generated so if we do the following calculation we should be able to know how many request per second the machine can handle:

$$\frac{TotalRequests}{TotalTime} = \frac{100}{4680} = 0.021 requests/s \tag{1}$$

In the case of using a delay of 400ms.

$$\frac{TotalRequests}{TotalTime} = \frac{100}{40794} = 2.45 * 10^{-3} requests/s \tag{2}$$

I did some experiments modifying the number of requests and see the time it took to handle all the requests. Over 1000 requests, the server start to reject and abort connections probably because it couldn't handle that amount of requests.

Number of Requests	Time in seconds
50	2355
100	4680
200	9516
400	18736
800	37519
1000	46925
1300	error connection aborted

Table 1: Time used to handle requests (40ms delay)

I also checked if there where any difference in performance if we took away the artificial delay in the server with the default number of request we obtained a time of 172 seconds. We can see that there is a great difference without the artificial delay that we inserted.

I also tried to simulate an access from different machines to the same server by opening several Erlang Shells and trying to simultaneously launch the test benchmark. In this case i tried to access through 2 Erlang Shells. It took more or less 8720 seconds. From this results we can see that handling 200 requests from only one source takes slightly more time than sending 100 requests from 2 different sources.

4 Conclusions

Thanks to this problem, i've learnt how a TCP connection is handled in the case of a functional language but it is also quite useful. Having the main idea of TCP works and also how you need to structure the problem to handle the requests gives you some basic knowledge that can be later on used , not only in Erlang but in other programming languages.

On the other hand, we see how to structure the HTTP requests and how we translate the RFC so it can be implemented in a program module.

Finally we get to practise some Erlang and get to see how several properties this language offers may be very useful when treating communications between several nodes.

5 Going Further

Due to my absence during my re-examinations in my home university i tried my best to give some thought on some of the questions that appeared in this section.

5.1 Increasing Throughput

Its possible to probably a design pattern that has a structure to create a thread pool in which each time a new request appears, a new thread is created to handle that request. Possibly in Erlang there is a similar structure or pattern that might be useful. I think that due to the native concurrency support in Erlang it might be easier to implement than in object oriented programming languages.

5.2 HTTP parsing

We can know the length of the body in this case when receiving a HTTP request. In the process of parsing the Header sections received, we can check if one of those headers corresponds to *content-length : value*. If we get the value from that header line, we can really know the length of the body in bytes.