Rudy: a small web server

Antonios Kouzoupis <antkou@kth.se>

September 17, 2014

1 Introduction

In this seminar our task is to write a very simple web server in Erlang. It will be able to receive only *GET* requests from a client and respond with a message. I have implemented a file delivery functionality so it is closer to a real web server. Some metrics will also be provided to evaluate the overall performance of our small server.

After we finish this seminar we will be able to:

- understand and use the socket API of Erlang
- understand the structure of a server process
- understand and describe the main aspects of HTTP protocol
- do some programming in Erlang

A web server, as a client–server architecture, is a distributed system in terms that there are resources several users can share, resources that we probably do not know where they reside physically. The communication between the client and the server is message oriented using *HTTP* as an application layer protocol and *TCP* as a transport layer protocol.

2 Main problems and solutions

The first task was to complete the TCP structure of the server. The source code at Figure 1 illustrates the sequence of a connection. In line 3 we set up a socket to listen to port *Port* with the options specified in line 2. Then in case of an incoming connection, we accept the connection (lines 12 – 17) and the *request/1* function receives the request. In line 23 we parse the client's request, we prepare a response by calling the *reply/1* function and we send the reply in line 25. Finally we close the connection to the client (line 29).

The initial requirement was to build a web server which responds with a **200 OK** status code to a client's request. I have extended that requirement so that it can respond with a file requested by the client.

The first issue was to extract the file path from the *URI* which was trivial. Using pattern match I extracted the path and name of file without the initial slash, code 47 in ASCII table and encapsulate it in the tuple that the http parser returns.

```
init(Port) ->
    Opt = [list, {active, false}, {reuseaddr, true}],
    case gen_tcp:listen(Port, Opt) of
        {ok, Listen} ->
            handler(Listen),
            gen_tcp:close(Listen);
        {error, Reason} ->
            io:format("rudy: Listen Error ~w\n", [Reason])
    end.

handler(Listen) ->
    case gen_tcp:accept(Listen) of
        {ok, Client} -> request(Client),
            handler(Listen);
        {error, Reason} ->
            io:format("rudy: Accept Error ~w\n", [Reason])
    end.

request(Client) ->
    Recv = gen_tcp:recv(Client, 0),
    case Recv of
        {ok, Str} ->
            Req = http:parse_request(Str),
            Response = reply(Req),
            gen_tcp:send(Client, Response);
        {error, Reason} ->
            io:format("rudy: Receive Error ~w\n", [Reason])
    end,
    gen_tcp:close(Client).
```

Figure 1: TCP Server

```
reply({{get, _, _}, File, _, _}) ->
            case file:read_file(File) of
                {ok, ContentBin} ->
                    Content =
                        unicode:characters_to_list(ContentBin,
                            unicode),
                    http:ok(Content);
                {error, Reason} ->
                    io:format("Rudy: Read File ~w\n", [Reason])
            end.
```

Figure 2: reply/1 function

Scenario	Time (ms)	Requests/sec
localhost/File exists	43.54	2296
localhost/File does not exist	60.55	1651
localhost/One request	0.749	
Remote Host (Greece)	17002	6
Remote Host (wlan)	609	164
2 Remote Hosts (wlan)	1293	77

Table 1: Metrics

Figure 2 illustrates the *reply/1* function which reads a file and sends its content to the client. In line 1 we pattern patch to get file name and in line 2 we use the built-in function of Erlang to read the contents of file. *read_file/1* function returns a binary data object of the contents so in line 5 we convert it to a list of characters and finally in line 6 we call the *ok/1* function which assemble the response.

The problem with delivering files was that most of the browsers after making the initial request of the web page, they automatically made a second request to get the *favicon.ico*. Using Wireshark to intercept the packets, it was clear that after each request for a web page, a second one was made for the favicon and as a consequence Rudy did not crash but it produced a lot of error messages. On the contrary, if we use a command line interface browser like Elinks, there is no such problem.

3 Evaluation

In the following section I will provide some metrics taken regarding the average response time of our web server for different scenarios. The metrics presented in Table 1 were taken by the benchmark provided and making 100 request each time.

As we expected, the response time when running the benchmark from localhost is much lower than running the benchmark from another machine in the same network. Very interesting is the fact that the response time measured in case of not finding the requested file is greater than the case when the file is found. If Rudy does not find a file it prints out an error message in terminal, so probably all these I/O interrupts produce some delay. Also, I run the Rudy server in a machine in Greece, introducing a significant amount of latency which leads us to high response time. Finally, two of my colleagues run the benchmark for 100 requests simultaneously which obviously resulted in a delayed response because of the single threaded architecture of our web server.

4 Conclusions

To conclude, in this seminar we have learnt the basics of HTTP protocol, how the request and reply is constructed and how to write a simple TCP server. We also had the opportunity to dive into Erlang and functional programming aspects.