Rudy: Implementation of a small web server

Tobias Johansson

September 17, 2015

1 Introduction

A small rudimentary web server called Rudy have been implemented in Erlang. To manage this firstly Erlang programming had to be learned. The code for parsing the HTTP request was given and the basic structure of a web server too.

The World Wide Web is a distributed system that contains very many web servers. To be able to handle multiple clients for a service simultaneously and increasing throughput, concurrent execution is used. Therefore an implementation is done and presented in the report there each request is handled concurrently.

Finally an evaluation of the implemented web server is done with benchmark tests.

2 Main problems and solutions

To complete the web server the given code for the structure of a web server had to be understood, the HTTP protocol to be reviewed and Erlang's API for TCP/IP socket learned. Before this the Erlang language also had to be learned. For learning Erlang the book *Programming Erlang: Software for a Concurrent World* by Joe Armstrong was studied.

2.1 HTTP

HTTP is a request-reply protocol implemented over TCP/IP. It is based on the client-server system architecture where web browsers and other clients act as clients and fetch recourses from web servers which act as servers.

A HTTP message is sent in plain text, ASCII coded, the message have the following parts in mentioned order:

- A request line if it is a request message, a status line if it is a response message.
- Header fields.

- An empty line.
- An optional message body.

The request and response line have three fields. The request line consist of a method that indicates request type, a request URI and HTTP version and the response line consist of HTTP version, a status code and status phrase.

A simple HTTP Erlang module was given that could parse a request to method, header and body. The parse function was only implemented to handle requests with the `GET` method. The module also had functions for generate messages for a HTTP response and a HTTP request.

2.2 TCP/IP sockets

The structure of a web server was given in code, but to be able to complete the rudimentary server the Erlang built-in `gen_tcp` module was studied. The goal with this study was to understand how to initiate a listening on a port, handle a request and reply it.

The following information about the `gen_tcp` was retrieved. To initiate a listening port `listen/2` is called. To wait for an incoming connection `accept/1` is called that will wait and return first when a incomming connection is gotten, an incoming request. Once the connection to the client is set the input can be read with `recv/2`. To then response to the request the module's function `send/2` is used.

2.3 Implementation

With the mentioned information in previous subsections the completion and implementation of the rudimentary web server Rudy was succeeded.

From the initiation function that set up the port a handler loop is called that will suspend and wait for a incoming requests, when a request is gotten handle it and then recursively call itself. The HTTP request is extracted with `parse_request/1` from the given HTTP module. From the request a response body, HTML code, is generated from URI in the request as below. The body is inserted in a response message and sent to the client.

```
retrieve_html(URI) ->
    "<!DOCTYPE html><html><head><title>404</title></head><body><h1>404</h1>
    <p>The requested URL " ++ URI ++ " could not be retrieved.</p><hr>
    <p><i>Rudy</i></p></body></html>".
```

2.4 Concurrent

To implement concurrent execution of incoming requests the used solution was to spawn a new process to handle every incoming request. To be able

to easier evaluate the performance in comparison to a sequential solution, there all incoming request are handled one at a time, a macro was written. The macro given below is making it possible to choose "Sequential mode" or "Parallel mode" during compilation.

```
-ifdef(seq).
-define(MODE, "Sequential").
-define(REQUEST(X), request(X)).
-else.
-define(MODE, "Parallel").
-define(REQUEST(X), spawn_link(fun() -> request(X) end)).
-endif.
```

3 Evaluation

To evaluate the implemented web server measurement was done of how many request per second that can be served and the response time. For this evaluation a test module was given that simulates 100 sequential requests and returns the time it took. The software Apache JMeter was also used to benchmark the server.

To simulate file handling, server side scripting etcetera a small delay on 40 ms was inserted before the server response sending. To test the artificial delay, the given test module was used and ran at the same machine as the server, it gave the result of 22.0 request/s with the delay and 1678.3 request/s without.

The first benchmark test was made with the given test module. For simulating the load from multiple users the module was ran on multiple OS processes in parallel (on the same machine as the server). To achieve that the processes started approximately the same time a bash script was written to start them in the terminal background. The web server was tested both in "Sequential mode" and "Parallel mode", with 1 up to 5 artificial users. The result from the test is presented in Table 1 where the benchmark for the individual users have been summed.

	1 user	2 users	3 users	4 users	5 users	
Sequential	22.2	22.5	22.5	22.5	22.5	request/s
Parallel	22.2	43.8	65.9	86.8	109.4	request/s

Table 1: Test module benchmark

The second test was with Apache JMeter running on another machine connected to the web server over internet. JMeter simulated 100 users HTTP requests from Rudy over 1 second measuring their response time. The result is given below in Table 2.

	Average	Max	
Sequential	4808	21001	ms
Parallel	92	1172	ms

Table 2: JMeter benchmark

4 Conclusions

To implement the rudimentary web server in Erlang the author of this report learned a lot about Erlang programming and concurrent programming. It also gave a useful review of the TCP/IP and HTTP protocols and a good insight over the structure of server process.

And the result of the benchmarks really shows the benefits of concurrent programming for increasing throughput by handling multiple clients for a service simultaneously. The parallel server approach does not have any benefits while only serving one user thread, but in the test module benchmark, see Table 1, the sum of request/s almost increases linear with almost 2 times higher result with 2 users and 5 times higher result with 5 users. If looking further to next benchmark, see Table 2, the parallel approach with 100 simulated users gives a *50 times* lower response time!