Rudy - a small web server

Mattias Cederlund, mcede@kth.se

September 16, 2014

1 Introduction

The task was to implement a small web server in Erlang, which consisted of a HTTP parser implementation and usage of a socket API.

The objective of the assignment was to gain knowledge in creating server processes, using sockets and the HTTP protocol as they are recurring technical concepts used in distributed systems. Also the assignment was useful for learning some basic Erlang.

2 HTTP Parser

The first part of the assignment was to implement a simple HTTP parser. The main task was to split the request into its components. Using the RFC 2616 format of a HTTP request the parsing was split in three parts, request-line, headers and body.

The HTTP parser code was provided and the main idea when parsing each component was to recursively traverse the string while looking for a special combination of characters that marked the end of each component. The parser would then consume each character from the beginning until the combination was found.

3 Rudy, server and sockets

Rudy, the http server, was implemented using the provided skeleton code consisting of four procedures: init, handler, request and reply.

init(Port) should initialize a listening socket and pass it to handler/1. This was done by calling handler(Listen) where marked.

handler(Listen) should listen to the socket for an incoming request. When a connection is incoming it should handle the request before terminating. This was done by calling request(Client) where marked.

request(Client) should parse the request using the HTTP parser and pass it to reply/1. This was done by adding Request = http:parse_request(Str) where marked.

reply(Request) should generate a reply to the provided request. The reply was generated using http:ok, and I chose to pass URI as the argument. The response would then simply consist of the URI. A request for "GET /foo" would generate the answer "/foo".

One problem with the server implementation was the fact that it terminated after each request was served. To solve this a new call to handler(Listen) was made after the request was handled using request(Client). The server would then continue listening for new requests after it handled the previous.

4 Evaluation

To test performance of the server the provided test program with the output function disabled was used. Also a delay of 40 ms was added to the reply function to simulate a heavy server operation.

The test program used 4680 ms to execute. This corresponded to 46,8 ms/request (or 21 requests/s) since the test program was set up to make 100 requests. These numbers suggests that the delay actually is significant.

Further, some other delays were tested and results are presented in Table 1. The results show a linear pattern and using linear fit gives the following formula:

```
Execution time (ms) = 103x + 780
```

where x is the artificial delay in ms. The constant value is the parsing overhead.

Sleep time (ms)	Benchmark time (ms)
10	1560
20	3120
40	4680
80	9360
160	17160

Table 1: Results when running benchmark test with different delays

Testing the server with multiple benchmark tests running at the same time and 40 ms delay had the following results (Table 2):

Number of tests running	Benchmark time (ms)
1	4680
2	9096
3	13453

Table 2: Results when running benchmark test with different delays

The results show a linear pattern with the total number of requests, as expected using a single-threaded program.

Testing a naive multi-threaded implementation where a new process is spawned to handle each request gave the same execution time as when running only one benchmark concurrently. The test was executed with 3 concurrent benchmark tests running. Having a quad-core processor this result is not so unexpected as all cores may be used, but I expected a larger overhead for creating processes.

5 Conclusions

I wouldn't say this assignment introduced any new concepts for me, but it was useful for refreshing my knowledge in Erlang programming.