Isabel Ghourchian CINTE14

Homework 1 Rudy - A small web server

1 Introduction

Our task in this seminar was to implement a small web server which consisted of a HTTP parser, a program that acts like a web server waiting for incoming requests and delivers a reply. And a small benchmark program which acts as a client and generates requests. This is achieved by the usage of a socket API and the functions defined in the gen_tcp library which allows the server to open a listening socket and listen to incoming requests from a client.

A web server has a client-server architecture where the client interacts with the server, it uses the TCP transport layer protocol. This client-server architecture is distributed system architecture, distributed services are called on by clients.

2 Main problems and solutions

The purpose of the HTTP parser is to parse a HTTP GET request. A request consist of: a request line, headers, carriage return line feed and an optional body. Each header is terminated by a carriage return line feed which in other word means a new line. The parsing of the different elements in the request was done by different function which recursively traversed the characters in the request and returned the parsed result.

A HTTP web server was then implemented with the help of the gen_tcp library in combination with the skeleton code provided in the assignment:

• init(Port): the procedure that will initialize the server, takes a port number (for example 8080), opens a listening socket and passes the socket to handler/1. Once the request has been handled the socket will be closed.
• handler(Listen): will listen to the socket for an incoming connection. Once a client has connect it will pass the connection to request/1. When the request is handled the connection is closed.
• request(Client): will read the request from the client connection and parse it. It will then parse the request using your http parser and pass the request to reply/1. The reply is then sent back to the client.
• reply(Request): this is where we decide what to reply, how to turn the reply into a well formed HTTP reply.

One problem with the initial server was that it terminated right after handling one request. In order for it to able to handle to new requests was to add a recursive call in handler/1. This made the server listen to new requests after it had handled one.

One optional task was to increase throughput by allowing several processes to listen to a socket. This was done by adding the following code in the server program:

```
start_proc(0, _) -> ok;
start_proc(N, Listen) ->
  spawn(rudy, handler, [Listen]),
  start_proc(N-1, Listen).
```

The function start_proc/2 spawns N number of threads in the handler/1 function which makes it possible for the server to listen several incoming requests.

3 Evaluation

To evaluate the performance of the web server a small benchmark program was implemented to serve as a client and generating 100 requests to the server. It measured the time it took to receive a reply from the server program. A small delay was put right before sending the reply from the server to simulate file handling.

The performance was measured in form of number of requests per seconds. It took 4155549 ms for 100 requests which corresponds to 4 seconds for 100 requests and that is 25 requests per seconds. When running the benchmark program from a different machine and localhost the server received a total of 200 requests and took twice the time of 100 requests, approximately 8 seconds. This makes sense since the number of requests sent by client has doubled has doubled. When removing the delay, the time took to receive a reply decreased to 58688 ms which is a significant difference.

When using the parallel solution and running the program from different machines the total time was 4131130 for 200 requests. This corresponds to approximately 4 seconds which means that the server can handle about 50 requests per seconds which is an improvement from the sequential program.

4 Conclusions

In this seminar I have learned the basics of HTTP and how the client-server architecture can be implemented in Erlang. Also how multithreading in Erlang is implemented and how communication from different machines work. The assignment also gave a brief introduction to sockets when working with the Erlang gen_tcp library.