ID2201 Distributed Systems, basic course

Rudy: A Small Web Server

Johan Montelius, Vladimir Vlassov and Klas Segeljakt
Email: {johanmon,vladv,klasseg}@kth.se
July 7, 2023

Introduction

Your task is to implement a small web server in Erlang. The aim of this exercise is that you should be able to:

- describe the procedures for using a socket API;
- describe the structure of a server process;
- describe the HTTP protocol.

As a side effect, you will also learn how to do some Erlang programming.

1 An HTTP parser

Let's start by implementing an HTTP parser. We will not build a complete parser, nor will we implement a server that can reply to queries (yet), but we will do enough to understand how Erlang works and how HTTP is defined.

1.1 An HTTP request

Open a file `http.erl` and declare a module `http` on the first line. Also, export the functions that we will use from the outside of the module. In the end, we will not export everything, but while you're debugging, you want to test the functions as we implement them.

```
-module(http).
-export([parse_request/1]).
```

We're only going to implement the parsing of an HTTP `GET` request and avoid some details to make life easier. Download the RFC 2616 from www.ietf.org and follow the descriptions of a request. From the RFC, we have the following:

```
  Request         = Request-Line               ; Section 5.1
                    *(( general-header         ; Section 4.5
                      | request-header         ; Section 5.3
                      | entity-header ) CRLF)  ; Section 7.1
                    CRLF
                    [ message-body ]           ; Section 4.3
```

So a request consists of a request line, an optional sequence of headers, a carriage return line feed (CRLF), and an optional body. We also know that each header is terminated by a carriage return line feed. OK, let's go; we implement each parsing function so that it will parse its element and return a tuple consisting of the parsed result and the rest of the string.

```
parse_request(R0) ->
    {Request, R1} = request_line(R0),
    {Headers, R2} = headers(R1),
    {Body, _} = message_body(R2),
    {Request, Headers, Body}.
```

1.2 The request line

Now take a look at the RFC and the definition of the request line.

```
      Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

The request line consists of a method, a request URI, and an HTTP version. All are separated by space characters and terminated by a carriage return and line feed. The method is one of `OPTIONS`, `GET`, `HEAD` etc. Since we are only interested in `GET` requests, life becomes easy.

To understand how we implement the parser, you must know how strings are represented in Erlang. Strings are lists of integers, and one way of writing an integer is `$G`, that is the ASCII value of the character G. So the string "GET" could be written `[$G,$E,$T]`, or (if you know your ASCII) as `[71,69,84]`. Now let's do some string parsing.

```
request_line([$G, $E, $T, 32 |R0]) ->
    {URI, R1} = request_uri(R0),
    {Ver, R2} = http_version(R1),
    [13,10|R3] = R2,
    {{get, URI, Ver}, R3}.
```

We match the input string with a list starting with the integers of G, E, and T, followed by 32 (which is the ASCII value for space). After having matched the input string with "GET " we continue with the rest of the string `R0`. We find the URI, the version, and finally, the carriage return and line feed that is the end of the request line.

We then return the tuple {{`get, URI, Ver`}, `R3`}, the first element is the parsed representation of the request line, and `R3` is the rest of the string.

If you have a problem following the description and have not yet started to read the Erlang "Getting Started" section, it is now high time to do so.

1.3 The URI

Next, we implement the parsing of the URI. This requires recursive definitions. This might twist your head if you have not been exposed to functional programming. If you have done some functional programming, you might say "this is not tail recursive; I can do it better", that is fine; I only wanted to make things easy and not introduce things we don't need.

```
request_uri([32|R0])->
    {[], R0};
request_uri([C|R0]) ->
    {Rest, R1} = request_uri(R0),
    {[C|Rest], R1}.
```

The URI is returned as a string. There is, of course, a whole lot o structure to that string. We will find the resource we are looking for, possibly with query information, etc. We leave it as a string for now but feel free to parse it later.

1.4 Version information

Parsing the version is simple: either version "1.0" or "1.1". We represent this by the atoms `v11` and `v10`. This means we can later switch on the atom rather than again parsing a string. It does mean that our program will stop working when 1.2 is released, but that will probably not be this week.

```
http_version([$H, $T, $T, $P, $/, $1, $., $1 | R0]) ->
    {v11, R0};
http_version([$H, $T, $T, $P, $/, $1, $., $0 | R0]) ->
    {v10, R0}.
```

1.5 Headers

Headers also have internal structure, but we are only interested in dividing them into individual strings and, most important, finding the end of the

header section. We implement this as two recursive functions; one that consumes a sequence of headers and one that consumes individual headers.

```
headers([13,10|R0]) ->
    {[],R0};
headers(R0) ->
    {Header, R1} = header(R0),
    {Rest, R2} = headers(R1),
    {[Header|Rest], R2}.

header([13,10|R0]) ->
    {[], R0};
header([C|R0]) ->
    {Rest, R1} = header(R0),
    {[C|Rest], R1}.
```

1.6 The body

The last thing we need is parsing of the body, and we will make things very easy (even cheating). We assume that the body is everything that is left, but the truth is more complex. If we call our function with a string as an input argument, there is little discussion of how large the body is, but this is not easy if we want to parse an incoming stream of bytes. When do we reach the end? When should we stop waiting for more? The length of the body is therefore encoded in the request's headers. Or rather, in the specification of HTTP 1.0 and 1.1, there are several alternative ways of determining the length of the body. If you dig deeper into the specs, you will find it messy. In our little world, we will, however, treat the rest of the string as the body.

```
message_body(R) ->
    {R, []}.
```

1.7 A small test

You now have all the pieces, and if you compile and load the module in an Erlang shell, you can parse a request. Call the function with the http module prefix and give it a string to parse.

```
7>c(http).
{ok,http}
8>http:parse_request("GET /index.html HTTP/1.1\r\nfoo 34\r\n\r\nHello").
{{get,"/index.html",v11},["foo 34"],"Hello"}
9>
```

1.8 What about replies

We will not deliver very interesting replies from our server, but here is a function that generates an HTTP reply with a status code 200 (200 is all OK). Another function can be convenient when we want to generate a request. Also, export these from the http module; we will use them later.

```
ok(Body) ->
    "HTTP/1.1 200 OK\r\n" ++ "\r\n" ++ Body.

get(URI) ->
    "GET " ++ URI ++ " HTTP/1.1\r\n" ++ "\r\n".
```

Note the double $\backslash r \backslash n$, one to end the status line and one to end the header section. A proper reply should contain headers that describe the content and the size of the body, but a normal browser will understand what we mean.

If you have had problems so far, we suggest that you study the "Getting Started" section and go through the tutorials found on the web before carrying on. If you have not had any problems so far, please continue.

2 The first reply

Your task is to start a program that waits for an incoming request, delivers a reply, and then terminates. This is not much of a web server, but it will show you how to work with sockets. The important lesson here is that a socket that a server listens to differs from the socket later used for communication.

Call the first test rudy, open a new file, and add a module declaration. You should define four procedures:

- `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 connected, it will pass the connection to `request/1`. When the request is handled, the connection is closed.
- `request(Client)`: It will read the request from the client connection and parse it. It will then parse the request using your http parser and pass it to `reply/1`. The reply is then sent back to the client.
- `reply(Request)`: this is where we decide what to reply and how to turn the reply into a well-formed HTTP reply.

The program could have the following structure (the ":" and "..." are open for you to fill in):

```
init(Port) ->
    Opt = [list, {active, false}, {reuseaddr, true}],
    case gen_tcp:listen(Port, Opt) of
        {ok, Listen} ->
            :
            gen_tcp:close(Listen),
            ok;
        {error, Error} ->
            error
    end.

handler(Listen) ->
    case gen_tcp:accept(Listen) of
        {ok, Client} ->
            :
        {error, Error} ->
            error
     end.

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

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

2.1 Socket API

You will need the functions defined in the `gen_tcp` library to implement the above procedures. Look up the library in the Kernel Reference Manual under "Application/kernel" in the documentation. The following will get you started:

- `gen_tcp:listen(Port, Option)`: this is how the server opens a listening socket. We will pass the port number as an argument and use the following option: `list`, {`active, false`}, {`reuseaddr, true`}. Using these options, we will see the bytes as a list of integers instead of a binary structure. We will need to read the input using `recv/2` rather than having it sent to us as messages. The port address should be used again and again.
- `gen_tcp:accept(Listen)`: this is how we accept an incoming request. If it succeeds, we will have a communication channel open to the client.
- `gen_tcp:recv(Client, 0)`: once we have the connection to the Client, we will read the input and return it as a string. The `0` augment 0, tells the system to read as much as possible.
- `gen_tcp:send(Client, Reply)`: this is how we send back a reply, in the form of a string, to the client.
- `gen_tcp:close(Socket)`: once we are done, we need to close the connection. Note that we also have to close the listening socket that we opened in the beginning.

Fill in the missing pieces, compile, and start the program. Use your browser to retrieve the "page" by accessing "http://localhost:8080/foo". Any luck?

2.2 A server

Now a server should, of course, not terminate after one request. The server should run and provide a service until it is manually terminated. To achieve this, we need to listen to a new connection once the first has been handled. This is easily achieved by modifying the `handler/1` procedure so that it calls itself recursively once the first request has been handled.

The problem is, of course, how to terminate the server. If it is suspended while waiting for an incoming connection, the only way to terminate it is to kill the process. You don't want to kill the Erlang shell, so one solution is to let the server run as a separate Erlang process and then register this process under a name in order to kill it.

```
-export([start/1, stop/0]).

start(Port) ->
    register(rudy, spawn(fun() -> init(Port) end)).

stop() ->
    exit(whereis(rudy), "time to die").
```

This is quite brutal, and one should, of course, do things in a more controlled manner, but it works for now.

3 The assignment

You should complete the rudimentary server described above and do some experiments. Set up the server on one machine and access it from another machine. A small benchmark program can generate requests and measure the time it takes to receive the answers.

```
-module(test).
-export([bench/2]).


bench(Host, Port) ->
    Start = erlang:system_time(micro_seconds),
    run(100, Host, Port),
    Finish = erlang:system_time(micro_seconds),
    Finish - Start.


run(N, Host, Port) ->
    if
        N == 0 ->
            ok;
        true ->
            request(Host, Port),
            run(N-1, Host, Port)
    end.


request(Host, Port) ->
    Opt = [list, {active, false}, {reuseaddr, true}],
    {ok, Server} = gen_tcp:connect(Host, Port, Opt),
    gen_tcp:send(Server, http:get("foo")),
    Recv = gen_tcp:recv(Server, 0),
    case Recv of
        {ok, _} ->
            ok;
        {error, Error} ->
            io:format("test: error: ~w~n", [Error])
    end,
    gen_tcp:close(Server).
```

Insert a small delay (40ms) in the handling of the request to simulate file handling, server-side scripting, etc.

```
reply({{get, URI, _}, _, _}) ->
```

```
timer:sleep(40),
http:ok(...).
```

How many requests per second can we serve? Is our artificial delay significant, or does it disappear in the parsing overhead? What happens if you run the benchmarks on several machines simultaneously? Run some tests and report your findings.

4 Going further

Consider and describe in your report possible improvements to the server. The most important is a multi-threaded extension.

Optional task for extra bonus: implement at least one of the improvements.

4.1 Increasing throughput

As it looks for now, our web server will wait for a request, serve it, and wait for the next. If the serving of a request depends on other processes, such as the file system or some database, we will be idle waiting while a new request might already be available for parsing. We want to increase the throughput of our server by letting each request be handled concurrently.

Should we create a new process for each incoming reply? Does it take time to create a process? What will happen if we have thousands of requests a minute? A better approach might be to create a pool of handlers and assign requests to them; if there are available handlers, the request will be put on hold or ignored.

Erlang even allows several processes to listen to a socket. One could create a fixed number of sequential servers, all listening to the same socket.

The Erlang system has support for multiprocessor architectures. If you have dual-core handy, you can do some performance testing. Read more on how to start Erlang with multiprocessor support. Things could improve a single-core processor since the system will be better at hiding latencies.

Run tests to see how much you can increase Rudy's request-serving throughput compared to the single-threaded implementation.

4.2 HTTP parsing

If you have done things the easy way, you might have taken for granted that the entire HTTP request is complete in the first string you read from the socket. This might not be the case, the request might be divided into several blocks, and you will have to concatenate these before you can read a complete request.

One simple solution is to do a first scan of the string and look for a double CR, LF. This will be the end of the header section; if you don't find it, you will have to wait for more. How do you know how long the body is?

Extend Rudy's HTTP parsing logic to handle requests divided into multiple blocks and demonstrate that your solution works.

4.3 Deliver files

It's not much of a web server if it can only deliver canned answers. Let's extend the server so that it can deliver files. To do this, we need to parse the URI request and separate the path and file from a possible query or index. Once we have the file, we need to make up a proper reply header containing the proper size, type, and coding descriptors. When your implementation works, demonstrate that you can deliver files from Rudy to your browser.

4.4 Robustness

The way the server is terminated could be more elegant. One could do a much better job by having the sockets active and delivering the connections as messages. The server could then be free to receive either control messages from a controlling process or a message from the socket.

Another problem with the current implementation is that it will leave sockets open if things go wrong. A more robust way is to trap exceptions, close sockets, open files, etc., before terminating. Ensure the server is not shut down abruptly before all current requests are handled.

Extend Rudy so that it can be shut down gracefully by sending it a `kill` message; demonstrate that it works as expected.