A Small Web Server
Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Introduction

Your task is to implement a small web server in Elixir. 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

As a side effect you will also learn a bit about the HTTP protocol.

1 A HTTP parser

Let's start by implementing a 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 Elixir can parse a HTTP request.

1.1 A HTTP request

Open a file `http.ex` and declare a module `HTTP` on the first line.

```
defmodule HTTP do

end
```

We're only going to implement the parsing of a HTTP get request; and avoiding some details to make life easier. If you download the RFC 2616 from www.ietf.org you will find this descriptions of a request. From the RFC we have:

```
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 consist of: a request line, an optional sequence of head-
ers (the * means zero or more), a carriage-return line-feed (CRLF) and, an
optional body (optional since it is enclosed in []). We also note that each
header is terminated by a carriage-return line-feed. OK, let's go; we imple-
ment 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.

```elixir
def parse_request(r0) do
  {request, r1} = request_line(r0)
  {headers, r2} = headers(r1)
  {body, _} = message_body(r2)
  {request, headers, body}
end
```

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 consist of: a method, a request URI and, a http version.
All 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 inter-
ested in get requests, life becomes easy.

To understand how we implement the parser you must know how strings
are represented in Elixir. 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.

```elixir
def request_line([?G, ?E, ?T, 32 | r0]) do
  {uri, r1} = request_uri(r0)
  {ver, r2} = http_version(r1)
  [13, 10 | r3] = r2
  {{:get, uri, ver}, r3}
end
```

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.

1.3 The URI

Next we implement the parsing of the URI. This requires recursive definitions. I know that you can write this as a tail recursive function but right now I don't think it is worth the trouble.

```elixir
def request_uri([32 | r0]), do: {[], r0}

def request_uri([c | r0]) do
  {rest, r1} = request_uri(r0)
  {[c | rest], r1}
end
```

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

1.4 Version information

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

```elixir
def http_version([?H, ?T, ?T, ?P, ?/, ?1, ?., ?1 | r0]) do
  {:v11, r0}
end

def http_version([?H, ?T, ?T, ?P, ?/, ?1, ?., ?0 | r0]) do
  {:v10, r0}
end
```

1.5 Headers

Headers also have internal structure but we are only interested in dividing them up into individual strings and most important find 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.

```
def headers([13, 10 | r0]), do: {[], r0}

def headers(r0) do
  {header, r1} = header(r0)
  {rest, r2} = headers(r1)
  {[header | rest], r2}
end

def header([13, 10 | r0]), do: {[], r0}

def header([c | r0]) do
  {rest, r1} = header(r0)
  {[c | rest], r1}
end
```

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 not that simple. If we call our function with a string as 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 headers of the request. 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 that it is quite messy. In our little world we will however treat the rest of the string as the body.

```
def message_body(r), do: {r, []}
```

1.7   a small test

You now have all the pieces to parse a request; try the string below. Note how we encode the special characters in a string, \r is carriage-return and \n is line-feed.

```
'GET /index.html HTTP/1.1\r\nfoo 34\r\n\r\nHello'
```

1.8   What about replies

We will not deliver very interesting replies from our server but here is a function that generates a HTTP reply with a 200 status code (200 is all OK). Another function can be convenient to have when we want to generate

a request. Also export these from the http module, we are going to use them later.

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

```
def ok(body) do
  "HTTP/1.1 200 OK\r\n\r\n #{body}"
end

def get(uri) do
  "GET #{uri} HTTP/1.1\r\n\r\n"
end
```

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

2   The first reply

Your task now 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 listen to is not the same thing as the socket later used for communication.

Call the first test rudy (since it is a rudimentary server), 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 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.

The program could have the following structure:

```
defmodule Rudy do

  def init(port) do
    opt = [:list, active: false, reuseaddr: true]
```

```elixir
    case :gen_tcp.listen(port, opt) do
      {:ok, listen} ->
        ...
        :gen_tcp.close(listen)
        :ok
      {:error, error} ->
        error
    end
  end

  def handler(listen) do
    case :gen_tcp.accept(listen) do
      {:ok, client} ->
        ...
      {:error, error} ->
        error
    end
  end

  def request(client) do
    recv = :gen_tcp.recv(client, 0)
    case recv do
      {:ok, str} ->
        ...
        ...
        :gen_tcp.send(client, response)
      {:error, error} ->
        IO.puts("RUDY ERROR: #{error}")
    end
    :gen_tcp.close(client)
  end

  def reply({{:get, uri, _}, _, _}) do
    HTTP.ok("Hello!")
  end

end
```

2.1 Socket API

In order to implement the above procedures you will need the functions
defined in the Erlang :gen_tcp library. Look up the library in the Kernel
Reference Manual found under "Application/kernel" in the documentation.

The following will get you starting:

- :gen_tcp.listen(port, option): this is how a listening socket is opened by the server. We will pass the port number as an argument and use the following option: :list, active: false, reuseaddr: true. Using these option 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 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 handler/1 procedure so that it calls itself recursively once the first request has been handled.

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

```
def start(port) do
  Process.register(spawn(fn -> init(port) end), :rudy)
end
```

```
def stop() do
  Process.exit(Process.whereis(:rudy), "Time to die!")
end
```

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.

```
defmodule Test do

  @number_requests 100

  def bench(host, port) do
    start = Time.utc_now()
    run(@number_requests, host, port)
    finish = Time.utc_now()
    diff = Time.diff(finish, start, :millisecond)
    IO.puts("Benchmark: #{@number_requests} requests in #{diff} ms")
  end

  defp run(0, _host, _port), do: :ok
  defp run(n, host, port) do
    request(host, port)
    run(n - 1, host, port)
  end

  defp request(host, port) do
    opt = [:list, active: false, reuseaddr: true]
    {:ok, server} = :gen_tcp.connect(host, port, opt)
    :gen_tcp.send(server, HTTP.get("foo"))
    {:ok, _reply} = :gen_tcp.recv(server, 0)
    :gen_tcp.close(server)
  end

end
```

You might want to change the procedure `request/2` during debugging to see if the socket communication actually works.

Remove the print-out calls so that we're measuring the performance of the server and not the performance of the output function. Insert a small delay (10ms) in the handling of the request to simulate file handling, server side scripting etc.

```elixir
def reply({{:get, uri, _}, _, _}) do
  :timer.sleep(10)
  HTTP.ok("Hello!")
end
```

Now, how many request 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 at the same time? Run some tests and report your findings.

4 Increasing throughput

Our web server as it looks right now will wait for a request, server the request 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.

Your task is to increase the throughput of our server by letting each request be handled concurrently. This means that the server should handle each request in its own Elixir process.

Should we create a new process for each incoming reply? Does it not take time to create a process (not really)? What will happen if we have thousand of requests a minute? A better approach might be to create a pool of handlers and assign request to them; if there are now available handlers the request will be put on hold or ignored.

Elixir even allows several processes to listen to a socket. One could create a fixed number of sequential servers that are all listening to the same socket. This is probably the easiest solution but you have to do some reading to understand the TCP module.

The Elixir system has support for multicore architectures. If you have dual-core handy you can do some performance testing. Read more on how to start Elixir with multiprocessor support. Things might even improve a single core processor since the system will be better in hiding latency. Do some performance measurements to see if your concurrent solution outperforms the sequential solution. You will have to change the benchmark program so that it is also concurrent.

5 Optional extensions

If you want to have some more challenges you can try to do the following improvements.

5.1 HTTP parsing

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

One simple solution is as follows: 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?

5.2 Deliver files

It's not much of an web server if it can only deliver canned answers. Let's extend the server so that it can deliver files. In order to do this we need to parse the URI request and separate the path and file form a possible query or index. Once we have the file in our hand we need to make up a proper reply header containing the proper size, type and coding descriptors.

5.3 Robustness

The way the server is terminated is of course not very elegant. One could do a much better job by having the sockets being active and deliver 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 and close sockets, open files etc before terminating.