Rudy: un servidor web sencillo*

Federico C. Repond
frepond@unq.edu.ar

Esteban Dimitroff Hódi
esteban.dimitroff@unq.edu.ar

15 de marzo de 2016

Introducción

La tarea es implementar un pequeño servidor web en Erlang. El objetivo de este ejercicio es ser capaz de:

- Describir los procedimientos para utilizar la API de socket.
- Describir la estructura de un proceso de servidor.
- Describir el protocolo HTTP.

Como objetivo adicional, aprenderemos un poco de programación Erlang.

1. Un parser de HTTP

Comencemos por un parser HTTP. No construiremos un parser completo ni implementaremos un servidor que pueda responder queries (por ahora) pero haremos lo suficiente para entender cómo funciona Erlang y cómo se define HTTP.

1.1. Un request HTTP

Abramos un archivo `http.erl` y declaremos un módulo `http` en la primera línea. Además exportaremos las funciones que usaremos desde afuera del módulo. Al finalizar, no exportaremos todo, pero mientras estamos probando queremos testear las funciones a medida que las implementamos.

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

*Adaptado al español del material original de Johan Montelius (`https://people.kth.se/~johanmon/dse.html`)

Solo implementaremos el parseo de una request GET de HTTP, evitando algunos detalles para hacernos la vida más fácil. Descarguemos el RFC 2616 de www.ietf.org y sigamos las descripciones para un request. Del RFC tenemos:

```
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
```

Es decir, un request consiste en: una línea de request, una secuencia opcional de encabezados (headers), una marca de fin de línea (CRLF: carriage return line feed) y un cuerpo (body) opcional. Notemos que cada encabezado también está terminado por un CRLF.

Empecemos; implementemos cada función de parsing para que parsee su elemento y retorne una tupla con el resultado parseado y el resto del string.

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

1.2. La línea de request

Ahora miremos la definición de la línea de request en el RFC.

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

La línea de request consiste en: un método, una URI del request y una versión de HTTP. Todas separadas por espacios y terminada en un CRLF.

El método es uno de: `OPTIONS`, `GET`, `HEAD`, etc. Como solo estamos interesados en requests `GET`, es más sencillo.

Para entender como implementar el parser, tenemos que saber como se representan los strings en Erlang. Los strings son listas de enteros y una manera de escribir un entero es `$G`, es decir, el valor ASCII del carácter G. Entonces el string "GET"puede ser escrito `[$G,$E,$T]`, o (si estamos al día con el ASCII) como `[71,69,84]`. Hagamos algo de parseo.

```
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}.
```

Matcheamos el string con una lista que comienza con los enteros de G, E y T, seguido por 32 (que es el valor ASCII para el espacio). Después de haber matcheado el string de entrada con "GET " continuamos con el resto del string `R0`. Encontramos la URI, la versión y finalmente el CRLF que marca el final de la línea de request.

Luego retornamos la tupla `{{get, URI, Ver}, R3}`, cuyo primer elemento es la representación parseada de la línea de request y `R3` es el resto del string.

Nota: Si resulta difícil seguir la explicación y aún no se comenzó a leer la sección "Getting Started" del sitio oficial de Erlang, es ahora el momento de hacerlo.

1.3. La URI

Ahora implementaremos el parseo de la URI. Esto requiere definiciones recursivas. Quizás resulte complicado para aquellos que no tengan experiencia en programación funcional. Y aquellos que si tengan algo de idea de programación funcional pueden decir "esto no es *tail recursive*, se puede hacer mejor", esta bien, la idea es hacer las cosas sencillas y no introducir conceptos que no necesitemos.

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

La URI es retornada como un string. Por supuesto hay un mundo de estructura en ese string. Determina el recurso que estamos buscando, y posiblemente con información de query, etc. Por ahora dejémoslo como un string, pero se puede parsear más adelante.

1.4. Versión

Parsear la versión es simple, o bien es versión "1.0" o "1.1". Representaremos eso con los atoms `v11` y `v10`. Es decir que más adelante podremos cambiar el atom en lugar de parsear el string de nuevo. Por supuesto también significa que nuestro programa dejará de funcionar al recibir solicitudes de otra versión, pero podremos vivir con esa limitación.

```
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

Los headers (o encabezados en determinadas traducciones) tienen una estructura interna, pero solo nos interesa dividirlos en strings individuales y lo

que es más importante, encontrar el final de la sección de headers. Implementaremos esto con dos funciones recursivas; una que consume una secuencia de headers, y otra que consume headers individuales.

```
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. El cuerpo

Finalmente necesitamos parsear el cuerpo del request y para eso haremos las cosas muy fáciles (incluso haciendo trampa). Vamos a asumir que el cuerpo es todo lo que queda del string, pero la verdad no es tan simple. Si llamamos a nuestra función con un string como argumento de entrada no hay mucha discusión de cuán largo es el cuerpo, pero no es tan sencillo si queremos parsear un stream de bytes de entrada. ¿Cuándo llegamos al final? ¿Cuándo debemos dejar de esperar más? El largo del cuerpo esta encodeado en los headers del request. O, mejor dicho, en la especificación de HTTP 1.0 y 1.1 hay varios métodos alternativos de determinar el tamaño del cuerpo. Si investigamos mejor en las especificaciones vamos a encontrar que es bastante engorroso. Por nuestra parte, en nuestro pequeño mundo trataremos al cuerpo como el resto del string.

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

1.7. Un pequeño test

Tenemos ahora todas las piezas y si compilamos y cargamos el módulo en una consola Erlang ya podemos parsear un request. Llamemos a la funcion con el prefijo del módulo `http` y demosle un string para parsear.

```
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. Las respuestas

No vamos a devolver respuestas muy interesante desde nuestro server, pero hagamos una función que devuelva una respuesta HTTP con un código de estado 200 (200 es el código para decir que todo esta bien). Otra función que puede ser conveniente tener es una que genere un request. Además, exportemos estas funciones en el modulo `http`, vamos a usarlas más adelante.

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

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

Notemos el doble `\r\n`; uno al final de la linea de status, y otro al final de la sección de headers. Una respuesta apropiada debería contener headers que describan el contenido y el tamaño del cuerpo de la respuesta, pero un browser standard va a entender lo que le enviemos.

Si hasta ahora se hace complicado, recordemos estudiar la sección "Getting Started" e intentar ir a través de los tutorials de la web, antes de continuar. Si no hay problemas, entonces adelante.

2. La primera respuesta

La tarea ahora es arrancar un programa que espere requests de entrada, devuelva una respuesta y termina. No es un gran web server, pero va a mostrarnos cómo trabajar con sockets. La lección importante es que un socket que un servidor escucha no es lo mismo que el socket usado más adelante para comunicación.

Llamemos el primer intento `rudy`, abramos un archivo nuevo y agreguemos una declaración de módulo. Deberíamos definir cuatro procedimientos:

- `init(Port)`: el procedimiento que inicializará el servidor, toma un número de puerto (por ejemplo 8080), abre un socket en modo escucha y pasa el socket a `handler/1`. Una vez que el request fue procesado el socket se cerrará.
- `handler(Listen)`: escuchará el socket esperando una conexión de entrada. Una vez que un cliente se conecta pasara la conexión a `request/1`. Cuando el request haya sido procesado se cerrará la conexión.
- `request(Client)`: leerá el request desde la conexión del cliente y la parseará. Usara nuestro http parser y pasará la request a `reply/1`. La respuesta luego es enviada al cliente.
- `reply(Request)`: es donde decidimos qué responder, como convertir la respuesta en una respuesta HTTP bien formada.

El programa puede tener la siguiente estructura: (los ":" están abiertos para completar):

```
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. La API de socket

Para implementar los procedimientos de arriba, vamos a necesitar las funciones definidas en la librería `gen_tcp`. Busquemos la librería en el Kernel Reference Manual en sección "Application/kernel" de la documentación. Lo siguiente debería ser útil:

- `gen_tcp:listen(Port, Option)`: así es como el server abre un socket en modo escucha. Pasamos el numero de puerto como argumento y usamos la siguiente lista de opciones: `[{active, false}, {reuseaddr, true}]`. Al usar estas opciones veremos los bytes como una lista de enteros en lugar de una estructura binaria. Tendremos que leer la entrada usando `recv/2`

en lugar de enviárnosla como mensajes. La dirección del puerto deberá ser usada una y otra vez.

- `gen_tcp:accept(Listen)`: Así es como aceptamos un request de entrada. Si es exitoso, tendremos un canal de comunicación abierto con el cliente.
- `gen_tcp:recv(Client, 0)`: Una vez que tengamos conexión con el cliente leeremos la entrada y la retornaremos como un string. El argumento 0, dice al sistema que lea todo lo posible.
- `gen_tcp:send(Client, Reply)`: Así es como devolvemos la respuesta, en forma de string, al cliente.
- `gen_tcp:close(Socket)`: Una vez que terminamos, necesitamos cerrar la conexión. Notar también que necesitamos cerrar el socket de escucha que abrimos al principio.

Completemos las partes faltantes, compilemos y arranquemos el programa. Usemos nuestro browser para obtener la "página" accediendo a "http://localhost:8080/foo". ¿Hubo suerte?

2.2. Un server

Ahora, un server por supuesto no debe terminar después de un request. El sever debería correr y proveer servicio hasta que sea cerrado manualmente. Para lograr esto necesitamos escuchar una nueva conexión una vez que la primera ha sido procesada. Esto es facil de lograr, modificando `handler/1` para que se llame a sí mismo recursivamente una vez procesado el primer request.

Un porblema es, claro, cómo cerrar el server. Si esta suspendido esperando conexiones, la única forma de cerrarlo es matando el proceso. Nosotros no queremos matar la consola de Erlang, entonces una solución es correr el server en un proceso separado y registrar este proceso con un nombre para poder cerrarlo.

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

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

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

Esto es bastante brutal, y uno por supuesto debería hacer las cosas de una manera un poco más controlada, pero por ahora alcanza.

3. El ejercicio

Se debe completar el server rudimentario descripto arriba y hacer algunos experimentos. Setear el server en una máquina y accederlo desde otra. Un pe-

queño programa de benchmark puede generar requests y medir el tiempo que lleva recibir las respuestas.

```
-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).
```

Eliminemos cualquier llamada a imprimir en la consola, para medir la performance del server y no de la función de output. Insertemos un pequeño delay (40ms) en el proceso del request para simular cualquier procesamiento de archivos, scripting del lado del servidor, etc.

```
reply({{get, URI, _}, _, _}) ->
    timer:sleep(40),
    http:ok(...).
```

Bien, ¿cuántos requests por segundo podemos servir? ¿Nuestro delay artificial es importante o desaparece dentro del overhead de parsing? ¿Qué ocurre si ejecutamos los benchmarks en varias máquinas al mismo tiempo? Hacer algunos tests y reportemos los resultados.

4. Un poco más allá

Discutamos (y aquel que lo desee puede implementar) algunas mejoras al server. Lo más importante es una extensión para hacerlo multi-threaded.

4.1. Incrementando el rendimiento

Así como está nuestro server, esperará por un request, lo responderá, y esperará al siguiente. Si para servir el request dependemos de otros procesos, como un sistema de archivos o alguna base de datos, el server estará ocioso mientras otro request puede estar listo para ser parseado. Queremos mejorar el rendimiento de nuestro server permitiendo que cada request sea procesado concurrentemente.

¿Deberíamos crear un nuevo proceso por cada request de entrada? ¿Toma tiempo crear un nuevo proceso? ¿Qué ocurriría si tenemos miles de requests por minuto? Un mejor enfoque puede ser crear un pool de handlers y asignar cada request a ellos; si no hay handlers disponibles ponemos el request en espera, o lo ignoramos.

Erlang permite incluso varios procesos escuchando un socket. Uno podría crear un número de servers secuenciales que están todos escuchando al mismo socket.

El sistema de Erlang tiene soporte para arquitecturas multiprocesador. Si se tiene un dual-core, se pueden hacer algunos tests de performance. Leer sobre cómo arrancar Erlang con soprte para multiprocesador. Las cosas deberían mejorar incluso en un procesador de un solo core ya que el sistema mejorará al minimizar las latencias.

4.2. Parseo de HTTP

Si las cosas se hicieron del modo fácil, dimos por descontado que el request HTTP estará completo en el primer string que leamos del socket. No siempre es el caso, un request puede estar dividido en varios bloques y deberíamos concatenarlos antes de leer el request completo.

Una solución simple es hacer una primera recorrida del string buscando un doble CRLF. Ésto será el fin de la sección de headers; si no lo encontramos, entonces deberemos esperar por más. ¿Cómo sabremos el tamaño del cuerpo?

4.3. Devolver archivos

No es un gran web server si solo podemos devolver respuestas fijas. Expandamos el server para que pueda retornar archivos. Para lograr eso necesitaremos parsear la URI y serparar el path y el nombre de archivo de un posible query o índice. Una vez que tengamos el nombre de archivo a mano, necesitamos construir una header de respuesta apropiado conteniendo el tamaño, tipo y condiciones de encoding.

4.4. Robustez

La manera en que el server se apaga es, evidentemente, no muy elegante. Uno podría hacer un trabajo mucho mejor teniendo los sockets activos y enviar las conexiones como mensajes. El server podria entonces estar libre para recibir mensajes de control desde un proceso controlador o mensajes desde el socket.

Otro problema con la implementación actual, es que dejara el socket abierto si las cosas salen mal. Un mecanismo más robusto debe capturar excepciones y cerrar sockets, archivos abiertos, etc. antes de terminar.