Erlang: Calentando Motores*

Federico C. Repond
frepond@unq.edu.ar

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

17 de marzo de 2016

1. Introducción

Este es una introducción rápida a Erlang dado que hay muchos tutoriales disponibles el la web. Sin embargo vamos a describir las herramientas que se necesitan para tener un entorno de desarrollo levantado y corriendo. Vamos a asumir que tiene conocimiento de algunos lenguajes de programación, han oído de programación funcional y que recursión no es un misterio.

2. Empezamos

Inicialmente, si usan sus propias computadoras van a necesitar instalar Erlang. En las computadoras de la UNQ posiblemente con algo de suerte ya lo tengamos. Lo primero que necesitamos es instalar el SDK de Erlang. La mejor fuente para encontrar los paquetes binarios para las distintas plataformas es Erlang Solutions. Encontraremos los binarios para Windows, Linux y OS X. Vamos a usar la versión R18 o superior. Si se encuentran utilizando Linux generalmente se encuentra en los repositorios de la distribución, aunque muchas veces no encontraremos la última versión. El SDK incluye el compilador, el shell, todas las librerías y la máquina virtual. No incluye el entorno de desarrollo.

2.1. Erlang

Una vez instalado Erlang deberíamos poder arrancar el shell de Erlang. Para esto lo hacemos buscando la entrada del programa en el menú de Windows o a mano desde el shell del sistema. Dado que más adelante vamos a necesitar establecer parámetros de runtime es conveniente aprender a arrancar Erlang a mano. En el shell del sistema hacemos:

```
$ erl
```

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

Esto debería iniciar el shell de Erlang y mostrarnos algo del estilo:

```
Erlang (BEAM) emulator version 5.6.1 [source] [smp:4]
[async-threads:0] [hipe] [kernel-poll:false]
Eshell V5.6.1  (abort with ^G)
1>
```

Ingresar `help().` (notar el punto al final) seguido de un return para ver todos los comandos del shell y `halt().` para salir del shell.

3. Hello World

Abrir un archivo `hello.erl` y escribir lo siguiente:

```
-module(hello).

-export([world/0]).

world()->
  "Hello world!".
```

Ahora abrir un shell de Erlang, compilar y cargar el archivo con el comando `c(hello)`, y llamar la función `hello:world()`. Recordemos que tenemos que terminar los comando del shell con un punto. Si todo salió bien, hemos escrito, compilado y ejecutado nuestro primer programa Erlang.

4. Programación Concurrente

Erlang fue diseñado desde su concepción para la programación concurrente. Vamos a aprender rápidamente como dividir un programa en procesos que se comunican y de esta forma obtener una mejor estructura. Probar lo siguiente:

```
-module(wait).

-export([hello/0]).

hello() ->
    receive
      X -> io:format("aaa! surprise, a message: ~s~n", [X])
    end.
```

La función `io:format` va a imprimir un string a stdout reemplazando los caracteres de control (los caracteres precedidos por por una tilde) con los elementos de la lista. La `~s` significa que el primer elemento debería ser un string, `~n` simplemente imprime una nueva línea. Cargar el módulo anterior y ejecutar el comando:

```
P = spawn(wait, hello, []).
```

La variable `P` está ligada al identificador del proceso creado. El proceso fue creado y llamado con la función `hello/0` (esta es la forma en que nombramos una función con 0 argumentos). Ahora el proceso está esperando por nuevos mensajes. En el mismo shell de Erlang ejecutar el comando:

```
P ! "hello".
```

Esto va a enviar un mensaje, en este caso el string `"hello"`, al proceso que va a despertar y continuar la ejecución.

Para hacer la vida más simple generalmente se registran los identificadores de procesos con un nombre que puede ser accedido por todos los procesos. Si 2 procesos se quieren comunicar, deben conoces el identificador del otro proceso. O bien un proceso envía el identificador al otro proceso cuando es creado en un mensaje, o a través de un nombre de un proceso registrado. Probar los siguiente en el shell (primero ingresar `f()`) para hacer que el shell olvide el ligado anterior de `P`):

```
P = spawn(wait, hello, []).
```

Ahora registrar el identificador del proceso con el nombre `"foo"`.

```
register(foo, P).
```

Finalmente enviar un mensaje al proceso.

```
foo ! "hello".
```

En este ejemplo lo único que enviamos fue un string pero podríamos enviar cualquier estructura de datos compleja. La sentencia `receive` puede tener varias cláusulas que intentan capturar los mensajes entrantes. Una cláusula solo va a ser usada si se encuentra una que concuerde. Probar:

```
-module(tic).

-export([first/0]).

first() ->
    receive
      {tic, X} ->
          io:format("tic: ~w~n", [X]),
          second()
end.

second() ->
    receive
      {tac, X} ->
```

```
            io:format("tac: ~w~n", [X]),
            last();
        {toe, X} ->
            io:format("toe: ~w~n", [X]),
            last()
    end.

last() ->
    receive
      X ->
            io:format("end: ~w~n", [X])
end.
```

Luego en un shell ejecutar los siguientes comandos:

```
P = spawn(tic, first, []).

P ! {toe, bar}.

P ! {tac, gurka}.

P ! {tic, foo}.
```

¿En que orden fueron recibidos los mensajes? Notar como los mensajes son encolados y como el proceso selecciona en que orden los procesa.

5. Programación Distribuida

La programación distribuida es relativamente fácil en Erlang, el único problema que vamos a tener es encontrar el nombre de nuestro nodo. Los sistemas distribuidos en Erlang trabajan normalmente con nombres de dominio en lugar de direcciones de IP explícitas. Esto puede ser un problema dado que estamos trabajando con laptops o PCs que generalmente no tienen asignados nombres en el DNS. Pero hay una solución.

Conectarse a la red y ejecutar ipconfig o ifconfig para encontrar nuestra dirección de IP. Vamos a usar esa dirección en forma explícita cuando iniciamos un nodo Erlang.

5.1. Nombre de un Nodo

Cuando se inicia Erlang se puede hacer que sea accesible en la red dándole un nombre. También vamos a querer asignarle una *cookie* secreta. Cualquier nodo que conozca la cookie va tener acceso para poder hacer cualquier cosa en dicho nodo. Esto es claramente bastante peligroso, es muy fácil para un nodo maligno bajar toda la red.

Iniciar un nuevo shell Erlang con el siguiente comando, reemplazando el IP por el que tenga cada uno.

```
$ erl -name foo@130.237.250.69 -setcookie secret
```

En el shell de Erlang se puede obtener el nombre del nodo con la *BIF* `node()`. Debería retornar algo como `'foo@130.237.250.69'`

Esto sin embargo nos va a impedir ejecutar múltiples shells de Erlang en un mismo nodo. Además hay que cambiar esto cada vez que nuestro IP cambia.

Iniciar un segundo shell Erlang usando el nombre `bar` en la misma máquina u otra. Cargar e iniciar el módulo `wait` definido anteriormente en el nodo foo.

Ahora en el nodo bar probar lo siguiente:

```
{wait, 'foo@130.237.250.69'} ! "a message from bar".
```

Es interesante ver como la comunicación entre procesos en Erlang es manejada de la misma forma independientemente de si el proceso ejecuta en el mismo shell, otro shell o en un shell en otro host. Lo único que cambia es como el shell Erlang inicia y como se accede a procesos registrados externamente.

Si se ejecuta en una computadora donde no se pueden abrir ports para comunicarse se necesita correr Erlang en el modo de distribución local. Se debe iniciar Erlang con un nombre corto de la siguiente forma:

```
erl -sname foo -setcookie secret
```

```
$ erl -sname foo -setcookie secret
```

El resto es igual pero se usa el nombre `foo` sin la dirección de IP.

6. Ejercicio: Ping-Pong

Si solo podemos enviar mensajes a procesos registrados no sería un sistema transparente. Podemos enviar mensajes a cualquier proceso pero el problema es obtener el identificador del proceso. No hay forma de escribir el valor del identificador del proceso `<0.70.0>` que el shell imprime.

El único proceso que conoce el identificador de proceso de un proceso es el que lo crea y el propio proceso. El propio proceso puede obtener el identificador llamando la BIF `self()`. Por supuesto podemos pasar identificadores de procesos, ya sea como parámetros o enviarlos en un mensaje a otro proceso conocido. Una vez que un proceso lo conoce puede usar el identificador sin importar si es un proceso local o remoto.

Probar definir en una máquina un proceso ping, que envía un mensaje a un proceso registrado en otra máquina con su propio identificador de proceso. El proceso debe esperar por una respuesta. El receptor en la otra máquina debe observar el mensaje y usar el identificador del proceso para enviar una respuesta.

Una aclaración, la primitiva send `!` acepta tanto nombres registrados, nombres remotos como identificadores de procesos. Esto puede causar problemas algunas veces. Si implementamos un sistema concurrente que no es distribuido y tenemos un proceso registrado con el nombre `foo`, podemos pasar el atom `foo`

y otros procesos tratarlo como un identificador de proceso. Ahora si hacemos la aplicación distribuida un proceso remoto no será capaz de utilizarlo como un identificador de proceso dado que el proceso es local al nodo que lo registra. Por eso debemos estar atentos de que lo que enviamos, es un identificador de proceso (que puede ser usado en forma remota) o es un nombre bajo el cual el proceso puede estar registrado.

7. Rebar3

Es recomendable el uso de *rebar3* para los proyectos que iremos trabajando durante el curso. Si bien no vamos a usar *OTP* (Open Telecom Platform) en principio, el uso de rebar3 nos dará una estructura adecuada al proyecto según los estándares de Erlang, con ventajas adicionales como compilación incremental y hotswap de código entre otras.

7.1. Nuevo Proyecto

Para crear un nuevo proyecto usamos:

```
$ rebar3 new app <erl_app_name>
```

Esto va a crear un proyecto con la siguiente estructura, donde en `src` podremos agregar nuestros fuentes. Por el momento podemos ignorar los archivos `erl_app_app.erl, erl_app_sup.erl, erl_app.app.src` que como mencionamos son parte de la infraestructura de OTP.

```
===> Writing erl_app/src/erl_app_app.erl
===> Writing erl_app/src/erl_app_sup.erl
===> Writing erl_app/src/erl_app.app.src
===> Writing erl_app/rebar.config
===> Writing erl_app/.gitignore
===> Writing erl_app/LICENSE
===> Writing erl_app/README.md
```

Para compilar el proyecto hacemos:

```
$ cd erl_app
$ rebar3 compile
===> Verifying dependencies...
===> Compiling erl_app
```

Esto va a generar un folder `_build` con nuestro archivos .beam.
Si queremos abrir un shell con nuestros módulos cargados:

```
$ rebar3 shell
```

Esto compilara nuestros módulo en caso de ser necesario. Luego en el shell podremos recompilar y cargar los módulos que modificamos sin salir de este.

```
1> r3:do(compile).
```

Además rebar3 provee soporte para manejo de dependencias, ejecución de tests, etc. Para mayor información referirse a `https://www.rebar3.org/v3.0/docs`