

Universidad Nacional de Quilmes, Departamento de Ciencia y Tecnología
Licenciatura en Desarrollo de Software

Sistemas Distribuidos

Erlang

Federico C. Repond
frepond@unq.edu.ar

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

March 17, 2016



- ▶ Desarrollado a principios de los 80 por Joe Armstrong, Robert Virding y Mike Williams, supervisados por Bjarne Däcker, en el Computer Science Laboratory de Ericsson.
- ▶ Su nombre se debe a Agner Krarup Erlang (1878–1929).
- ▶ Concebido inicialmente como software para productos de telecomunicaciones.
- ▶ Influido por Prolog, LISP, Ada, Miranda, ML, Haskell, Modula y Chill, entre otros.
- ▶ Inicialmente, usó una VM en Prolog. en 1991 se reemplaza por otra escrita en C (con inspiración de la WAM).
- ▶ En 1998 Ericsson decide liberarlo, con una licencia similar a la Mozilla Public License (EPL). Desde la R18 tiene Apache License 2.0.



Erlang es...

- ▶ Funcional (no tipado).
- ▶ Concurrencia con pasaje de mensajes.
- ▶ Pattern-matching en la recepción de mensajes.
- ▶ Opera con módulos y behaviors.
- ▶ Tiene listas por comprensión.



Su implementación... ¹

- ▶ Ejecuta en una máquina virtual (BEAM).
- ▶ Procesos muy livianos (327 words).
- ▶ Creación rápida de un proceso.
- ▶ Permite hot-swap.
- ▶ Debía funcionar el 99.9999999 por ciento del tiempo (puede parar 30 ms al año).

¹<https://pragprog.com/articles/erlang>



Citado junto con Haskell en The Economist.

Meanwhile, a group of obscure programming languages used in academia seems to be making slow but steady progress, crunching large amounts of data in industrial applications and behind the scenes at large websites. Two examples are Erlang and Haskell, both of which are “functional programming” languages.



- El shell es nuestra mejor herramienta como programadores Erlang!

```
[frepond@localhost erlang]$ erl
Erlang/OTP 18 [erts-7.2.1] [source] [64-bit] [smp:8:8]
[async-threads:10] [hipe] [kernel-poll:false] [dtrace]
```

```
Eshell V7.2.1 (abort with ^G)
1> io:format("Hello, world!~n").
Hello, world!
ok
2> q().
ok
[frepond@localhost erlang]$
```



```
[frepond@localhost erlang]$ cat hello.erl
-module(hola).
-export([hola/0]).
hello() -> "Hello, world!".

[frepond@localhost erlang]$ erl
Erlang/OTP 18 [erts-7.2.1] [source] [64-bit] [smp:8:8]
[async-threads:10] [hipec] [kernel-poll:false] [dtrace]
[smp:4:4] [rq:4] [async-threads:0]
[kernel-poll:false]

Eshell V5.7.5  (abort with ^G)
1> c(hello).
{ok,hello}
2> hello:hello().
"Hello, world!"
3> q().
```



- ▶ Se pueden pasar opciones al compilador. Por ejemplo,
`1> c(hola, [native]).`
- ▶ También existe un compilador standalone `erlc`.



- ▶ Erlang tiene un tipo “exótico”, `atom`, que es muy usado.
- ▶ Es similar a un `enum` de C, aunque no se define. También a un `quote` de LISP, o constante de Prolog.
- ▶ Sus nombres deben estar en minúsculas, o entre apóstrofes.
- ▶ Ejemplos: `ok`, `fatal_error`, `'EXIT'`, `'\ $q'`.



- El factorial de 200:

```
2> fact:fact(200).
```

78865786736479050355236321393218506229513597768
7173263294742533 244359449963403342920304284011
98462390417721213891963883025764279024263710506
19266249528299311134628572707633172373969889439
22445621451664240254033291864131227428294853277
52424240757390324032125740557956866022603190417
0324062351700858796178922227896237038973747200
00



- ▶ Caracteres. `$a`, `$\n`.
- ▶ Strings. `"Un string"`. Son, en realidad, listas de enteros.

```
2> "gato\7".
```

```
[103,97,116,111,7]
```



- ▶ Números de punto flotante.
- ▶ Usa la norma IEEE 754 de 64-bits (rango: $\pm 10^{308}$.)
- ▶ El módulo `math` tiene las funciones usuales:
`pi/0`, `sin/1`, `exp/1`, etc.



- ▶ Operaciones, por precedencia decreciente:
 - ▶ `:` `#`
 - ▶ `+` `-` (unarios)
 - ▶ `*` `/` `div` `rem` `bnot` `band`
 - ▶ `+` `-` `bor` `bxor` `bsl` `bsr` `andalso` `orelse`
- ▶ Las precedencias se cambian con paréntesis.



- ▶ Puntos a tener en cuenta sobre los operadores de comparación.
 - ▶ Menor o igual se denota como en Prolog: `=<`.
 - ▶ Tiene dos operadores de igualdad: `:=` y `==`. También dos de diferencia: `=/=` y `/=`. (`1 := 1.0 (false)`, `1 == 1.0 (true)`)
 - ▶ Las primeras versiones se usan para comparar dos términos cualesquiera. Las segundas, si se usa punto flotante.
- ▶ En Erlang todos los términos son comparables. El criterio es:
`atom < number < list < tuple`.
- ▶ Los enteros y flotantes se comparan de la manera usual.
- ▶ Los demás se comparan según lo anterior, o en orden lexicográfico.



- ▶ Tuplas. {ok, 1}, {'EXIT', {bad_argument}}.
- ▶ Listas. [1, 2, 3], [ok, 10].
- ▶ Cuidado con las listas impropias:

```
[1| [2]] -> [1, 2],  
[1| 2]   -> [1| 2].
```

- ▶ Podrían usarse para un generador de enteros infinito²:

```
-module(generator).  
-export([ints_from/1]).  
  
-spec ints_from(integer()) -> fun().  
ints_from(N) ->  
    fun() ->  
        [N | ints_from(N + 1)]  
  
end.
```

²<https://github.com/rpt/estreams>



- ▶ Variables. Sus nombres deben empezar con mayúsculas. Son, en realidad, bindings de pattern–matching, y se asignan sólo una vez.
- ▶ ¿Qué pattern–matchings tiene Erlang?



- ▶ Las constantes en los mismos ctors Deben coincidir.
- ▶ Las variables asignadas deben tener el mismo valor que las constantes respectivas.
- ▶ Las variables no asignadas se ligan a los valores correspondientes.
- ▶ Ejemplos.
 $\{N, N\} = \{3, 3\}$. $N = 3$. $\{N, N\} = \{3, 4\}$. *% error*



- ▶ Las tuplas no se prestan fácilmente a cambios en los programas. Los records tratan de remediar esto.

- ▶ Se declaran así (ej.)

```
-record(usuario, {nombre = "Juan Perez", dir, tel}).
```

```
rd(person, {nombre = "Juan Perez", dir, tel}). % shell
```

- ▶ Se crean así.

```
R=#usuario{}
```

```
R=#usuario{dir = "Paso 1342"}.
```

```
R=#usuario{nombre = "José García",  
            dir = "Paso 1341", tel = 1626}.
```



- ▶ Los campos se pueden acceder:

```
R#usuario.nombre.
```

- ▶ También con pattern matching,

```
#usuario{nombre = N, dir = D, tel = T} = R
```

- ▶ También se pueden modificar campos (en realidad, se hace una copia alterando los campos indicados).

```
R1=R#usuario{dir="lejos"}
```

- ▶ Podemos testear por tipo.

```
foo(P) when is_record(U, usuario) -> un_usuario;
```



- Funciones. Ejemplo:

```
cosa(0) -> io:format("cero!~n"), ok;  
cosa(N) when N>0 ->  
    io:format("~p~n", [N]), cosa(N-1).
```

- Las definiciones terminan con punto.
- Las cláusulas terminan con punto y coma. Pueden tener guardas.
- La secuencia se indica con coma.



- ▶ Permutaciones de una lista.

```
perms([]) -> [[]];
```

```
perms(L) -> [[X | Y] || X <- L, Y <- perms(L -- [X])].
```

- ▶ La recursión de cola se transforma en iteración.
- ▶ Pueden coexistir, en el mismo módulo, funciones con igual nombre y distinta aridad. Ejemplos:

```
io:format/1, io:format/2
```



- Funciones anónimas (clausuras) se definen así (ej.):

```
F = fun(X, Y) -> X + Y end, F(3, 4).
```

- Admiten más de una cláusula. Ejemplo:

```
F = fun(0) -> 10; (N) when N > 0 -> 0 end.
```

- Se usan también para tomar funciones de otros módulos. Ej.

```
F = fun io:format/1. % un argumento.
```



Guardas.

- ▶ Se forman con secuencias de expresiones separadas por “;”.
- ▶ Si una secuencia es verdadera, la guarda es verdadera. Las expresiones se separan con “,”.
- ▶ Si una expresión es falsa, la secuencia es falsa.



Las expresiones pueden ser:

- ▶ `true` (otras constantes evalúan a `false`).
- ▶ BIFs.
- ▶ Comparaciones.
- ▶ Expresiones aritméticas.
- ▶ Expresiones booleanas.
- ▶ Cortocircuitos `andalso` y `orelse`.



- ▶ Un bitstring es una secuencia de bits. Ej.: `<<10, 11, 12>>`.
- ▶ Se puede especificar cuántos bits ocupará cada valor. Ej. `<<23:4, 1:1, 3:2>>`.
- ▶ No es necesario que estén alineados en 8 bytes.
- ▶ Soportan pattern-matching. Ej. `<<C:4, D:2>> = <<1998:6>>`.

- ▶ Además de bitstring, los campos de un binario aceptan otros modificadores, como `unit`, `float`, `integer`, `binary`, `bits`, `signed`, `unsigned`, `utf16`, etc.
- ▶ Erlang es, internamente, big-endian.

```
A = <<1245:16>>,
<<A1:8, A2:8>> = A,
io:format("~p ~p~n", [A1, A2]),
B = <<1245:16/little>>,
<<B1:8, B2:8>> = B,
io:format("~p ~p~n", [B1, B2]).
```

- ▶ Imprime:

```
4 221
221 4
```



- Guardemos y extraigamos un float.

```
10> A = <<3.14/float>>.  
<<64,9,30,184,81,235,133,31>>  
11> <<F/float>> = A.  
<<64,9,30,184,81,235,133,31>>  
12> F.  
3.14
```

- Guardemos -1 (signed) y extraigámoslo.

```
31> A= <<-1:32>>.
```

```
<<"ÿÿÿÿ">>
```

```
32> <<B:32>> = A.
```

```
<<"ÿÿÿÿ">>
```

```
33> B.
```

```
4294967295
```

```
34> <<C:32/signed>> = A.
```

```
<<"ÿÿÿÿ">>
```

```
35> C.
```

```
-1
```



- Otro ejemplo. Saquemos el segundo byte.

```
48> A = <<16#12345678:32>>.
```

```
<<18,52,86,120>>
```

```
49> <<_:8, B:8, _/bits>> = A.
```

```
<<18,52,86,120>>
```

```
50> B.
```

```
52
```



- Un parser para segmentos TCP.

```
<<SourcePort:16, DestinationPort:16, AckNumber:32,  
DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,  
Checksum:16, UrgentPointer:16,  
Payload/binary>> = SomeBinary.
```



La parte impura de Erlang: cada proceso tiene un diccionario mutable y privado. (entre otras!)

- ▶ Se insertan pares clave–valor con `put(C, V)`.
- ▶ Se rescatan con `get(C)` o `get()`.
- ▶ Se borran con `erase(C)` o `erase()`

Concurrencia.

- ▶ Cada proceso tiene un pid, disponible con `self()`.
- ▶ Se crean con `spawn/1` y `spawn/3`. Ejemplos.

```
spawn(io, format, ["Hola ~p~n", ["mundo!"]]).
```

```
-module(sp3).  
-export([init/1, proc/1]).  
proc(P) -> io:format("Hola, ~p~n", [P]).  
init(0) -> ok;  
init(N) when N > 0 ->  
    spawn(sp3, proc, [N]), init(N - 1).
```




- ▶ Un proceso puede conseguir un identificador único entre nodos conectados, llamado referencia. Se obtiene con `make_ref`.
- ▶ Cada proceso tiene un inbox, donde puede recibir mensajes arbitrarios.



- Dos procesos se pueden comunicar con ! y receive con pattern matching y guardas. Ejemplo.

```
-module(first_proc).  
-export([init/1]).
```

```
p() -> receive {pid, Pid} ->  
    io:format("soy ~p~n", [Pid]), p1(Pid) end.  
p1(P) -> receive {a, A} ->  
    io:format("~p: llega ~p~n", [P, A]), p1(P);  
    fin -> io:format("~p: muero~n", [P]) end.  
init(0) -> ok;  
init(N) when N > 0 -> Pid = spawn(fun p/0),  
    Pid ! {pid, Pid}, Pid ! {a, N},  
    Pid ! fin,  
    init(N - 1).
```



- Un mensaje puede hasta ser código. Un ejemplo de hot-swap.

```
-module(hot_swap).  
-export([init/0, loop/1]).
```

```
loop(F) ->  
    receive  
        {swap, FF} -> loop(FF);  
        {arg, {Pid, X}} -> Pid ! F(X), loop(F)  
    end.
```

```
init() -> spawn(q, loop, [fun(X) -> 2 * X end]).
```



► Un ejemplo de uso.

```
1> P = hot_swap:init().
<0.34.0>
2> P!{arg, {self(), 15}}. receive X -> X end.
{arg,{<0.32.0>,15}}
3> receive X -> X end.
30
4> P!{swap, fun(X) -> X * X end}.
{swap,#Fun<erl_eval.6.80247286>}
5> P!{arg, {self(), 15}}. receive XX -> XX end.
{arg,{<0.32.0>,15}}
6> receive XX -> XX end.
225
```



- Un aspecto delicado: `receive` tiene semántica de asignación. Un ejemplo.

```
-module(recv).  
-export([init/0]).
```

```
proc() ->  
    receive {ok, X} ->  
        io:format("~p~n", [X])  
    end,  
    io:format("~p~n", [X]).
```

```
init() ->  
    Pid = spawn(fun proc/0),  
    Pid ! {ok, 34}.
```



► Produce.

34

{ok, 34}

34



- `receive` puede tener un timeout en milisegundos.

```
receive X -> f(X)
after 300 -> io:format("timeout!~n")
end
```

- Una función para vaciar el inbox.

```
vaciar() ->
    receive _ -> vaciar();
    after 0 -> ok
end.
```



- Ejemplo: un mutex implementado con send/receive.

```
-module(mutex).  
-export([create/0, lock/1, unlock/1]).
```

```
create() ->  
    spawn(fun() -> unlocked() end).
```

```
lock(Mutex) ->  
    Mutex ! {lock, self()},  
    receive  
        ok -> ok  
    end.
```

```
unlock(Lock) ->  
    Lock ! {unlock, self()}.
```




```
unlocked() ->
    receive {lock, LockedBy} ->
        LockedBy ! ok,
        locked(LockedBy, 1)
    end.
```

```
locked(LockedBy, Count) ->
    receive
        {unlock, LockedBy} ->
            if Count == 1 -> unlocked();
            true -> locked(LockedBy, Count - 1)
        end;
        {lock, LockedBy} ->
            LockedBy ! ok,
            locked(LockedBy, Count + 1)
    end.
```



```
1> M = mutex:create().
<0.72.0>
> mutex:lock(M).
ok
2> spawn(fun() -> mutex:lock(M),
          io:format("unlocked!~n", []),
          mutex:unlock(M) end).
<0.86.0>
3> mutex:unlock(M).
unlocked!
{unlock,<0.69.0>}
```

- Erlang ofrece case. Ejemplo.

```
case cmp(A, B) of
  greater ->
    f(A - B),
    ok;
  equals when A == 0 ->
    zero;
  _ ->
    no_ok
end
```

- If es un caso particular de case, no muy usado.

```
if
  A > B -> io:format("mayor~n");
  A < B -> io:format("menor~n");
  true -> io:format("igual~n")
end
```

Las excepciones se clasifican en:

- ▶ `error`, causadas por división por cero, etc. Se levantan con `error`.
- ▶ `exit`, donde el proceso puede informar por qué termina. Se levantan con `exit`.
- ▶ `throw`, para excepciones definidas por el usuario. Se levantan con `throw`.

- ▶ Se capturan:

```
try 1/0
catch
  _:_ -> io:format("auch!~n")
end.
auch!
ok
```

- ▶ Para expresiones que se procesan con case

```
try 1/0 of
  1 -> io:format("uno!~n");
  N -> io:format("otra cosa!~n")
catch
  _:_ -> io:format("auch!~n")
after
  io:format("chau!~n")
end.
```

- Un ejemplo más:

```
f() -> throw(cosa).  
g() -> exit(cosa).  
h() -> error(cosa).  
init() ->  
    lists:foreach(  
        fun(F) ->  
            try F()  
            catch A:B ->  
                io:format("~p:~p~n", [A, B])  
            end  
        end, [fun f/0, fun g/0, fun h/0]).
```

- La salida es:

```
throw:cosa  
exit:cosa  
error:cosa
```



- ▶ Un behaviour podría verse como una interface de Java.
- ▶ Obliga a que el módulo que la implementa deba exportar determinadas funciones.
- ▶ Ejemplo.

```
-module(behav0).  
-export([behaviour_info/1]).
```

```
behaviour_info(callbacks) ->  
    [{zstart, 1}, {ztop, 0}];  
behaviour_info(_) -> undefined.
```

- Se puede usar así:

```
-module(usebeha0).  
-export([zstart/1, ztop/0]).  
-behaviour(beha0).
```

```
zstart(X) -> X.
```

```
ztop() -> ok.
```




- Tracing es un aliado fundamental a la hora de entender que está sucediendo.

```
dbg:tracer().  
dbg:p(all,c).  
dbg:tpl(except, init, []). % función específica  
dbg:tpl(except, '_', []). % todo el módulo
```



Agradecimientos

Agradecemos a *Guido Macchi* y *Diego Llarrull* de la Universidad Nacional de Rosario, por la guía y material para el armado de la intro a Erlang.