

# Erlang

---

## Aspectos Básicos

I

# ¿Qué es Erlang/OTP?

Erlang is a programming language used to build **massively scalable soft real-time systems** with requirements on **high availability**. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for **concurrency, distribution and fault tolerance**. Originally developed at **Ericsson**, it was released as **open source** in 1998.

OTP is a set of Erlang **libraries and design principles** providing middle-ware to develop these systems.

**soft real-time systems:** Sistemas en tiempo real en los que no se pueden hacer garantías sobre el tiempo exacto que va a llevar una acción. Estas garantías solo las pueden hacer dispositivos hardware muy específicos, eso sería **hard real-time**.

conurrencia, distribución y tolerancia a fallos van unidas, si quieres un sistema que realice un gran número de tareas y que además sea capaz de gestionar errores tanto de software como de hardware (que antes o después se producirán), necesitas que ese sistema esté distribuido en varias máquinas.

# Lenguaje Funcional

- No OO
- Funciones agrupadas en módulos
- Estado inmutable
- Funciones sin side-effects (casi)

# Variables Invariables

```
Eshell V5.7.5 (abort with ^G)
1> A = 1.
1
2> A = 2.
** exception error: no match of right hand side value 2
```

# Pattern Matching (I)

```
Eshell V5.7.5 (abort with ^G)
1> [ H | T ] = ["uno", "dos", "tres"].
["uno", "dos", "tres"]
2> H.
"uno"
3> T.
["dos", "tres"]
```

# Pattern Matching (II)

```
Eshell V5.7.5 (abort with ^G)
1> {_, S, [H | T]} = {hola, "abc", [1,2,3,4]}.
{hola, "abc", [1,2,3,4]}
2> S.
"abc"
3> H.
1
4> T.
[2,3,4]
```

# Pattern Matching (III)

```
-module(animales).  
  
-export([sonido/1]).  
  
sonido(perro) ->  
    io:format("Guau guau!~n");  
  
sonido(gato) ->  
    io:format("Miauuuuuuu!~n");  
  
sonido(_Otro) ->  
    io:format("Grrrrrrrrrr!~n").
```

# Pattern Matching (y IV)

```
Eshell V5.7.5 (abort with ^G)
```

```
1> animales:sonido(perro).
```

```
Guau guau!
```

```
ok
```

```
2> animales:sonido(gato).
```

```
Miauuuuuuu!
```

```
ok
```

```
3> animales:sonido(zorro).
```

```
Grrrrrrrrrr!
```

```
ok
```

# Recursividad

```
-module(recursion).
-export([map/2]).

map(F, List) ->
    map(F, [], List).

map(_F, Result, []) ->
    Result;
map(F, Result, [Head | Tail]) ->
    map(F, Result ++ [F(Head)], Tail).
```

```
Eshell V5.7.5 (abort with ^G)
1> recursion:map(fun(E) -> 2*E end, [1,2,3,4,5]).
```

[2,4,6,8,10]

# Higher Order Functions

```
2> PorDos = fun(E) -> 2 * E end.  
#Fun<erl_eval.6.13229925>  
3> recursion:map(PorDos, [1,2,3,4,5]).  
[2,4,6,8,10]
```

IO

# List Comprehensions

```
Eshell V5.7.5 (abort with ^G)
1> [ 2 * X || X <- [1,2,3,4,5] ].  
[2,4,6,8,10]
```

# Bit Syntax (I)

```
Eshell V5.7.5 (abort with ^G)
1> Color = 16#F09A29.
15768105
2> <<R:8, G:8, B:8>> = <<Color:24>>.
<<240,154,41>>
3> R.
240
4> G.
154
5> B.
41
```



# Otras Características

- Hot Code Loading
- Operaciones IO asíncronas

# Erlang

---

Concurrencia

15

# Concurrencia VS Paralelismo

16

Monday, 18 July 2011

16

Concurrencia: Se produce cuando varias tareas se inician ejecutan y finalizan superpuestas en el mismo periodo de tiempo. No significa necesariamente que en un instante determinado se estén ejecutando a la vez.

Paralelismo: Las tareas se ejecutan exactamente al mismo tiempo, por ejemplo en un procesador multicore.

Erlang tiene concurrencia desde sus inicios en los años 80, el paralelismo ha sido añadido recientemente (SMP).

# Escalabilidad

I7

Monday, 18 July 2011

17

Erlang está diseñado desde el principio para soportar un gran número de procesos ligeros. El objetivo de la escalabilidad es superar las limitaciones del hardware. Hay dos formas de hacer esto, añadiendo mejor hardware (más RAM, discos más rápidos, etc...) o añadiendo más máquinas. La primera opción es útil sólo hasta cierto punto.

# Tolerancia a fallos (Let it Crash)

18

Monday, 18 July 2011

18

Puedes intentar evitar que el software tenga bugs, pero es muy difícil evitar todos los posibles errores. Aún así no hay forma de evitar que se produzcan fallos hardware.

Los errores ocurren, Erlang proporciona formas de gestionar los errores cuando ocurren en vez de evitar que se produzcan. El hecho de que los procesos en Erlang tengan su memoria independiente y de que no haya necesidad de locks para acceder a zonas de memoria compartida evita que un proceso pueda dejar a otro en un estado inconsistente en caso de ‘crash’.

La idea de Erlang es que si un proceso es que si un proceso está en un estado inconsistente lo mejor es dejarlo morir lo antes posible sin afectar a otros procesos.

La única forma de evitar que un fallo de hardware afecte a un programa es haciendo que el programa se ejecute en más de una máquina al mismo tiempo.

# Modelo de Actores

- Procesos ligeros
- Memoria independiente
- Envío de mensajes asíncronos
- Operaciones IO asíncronas

19

Monday, 18 July 2011

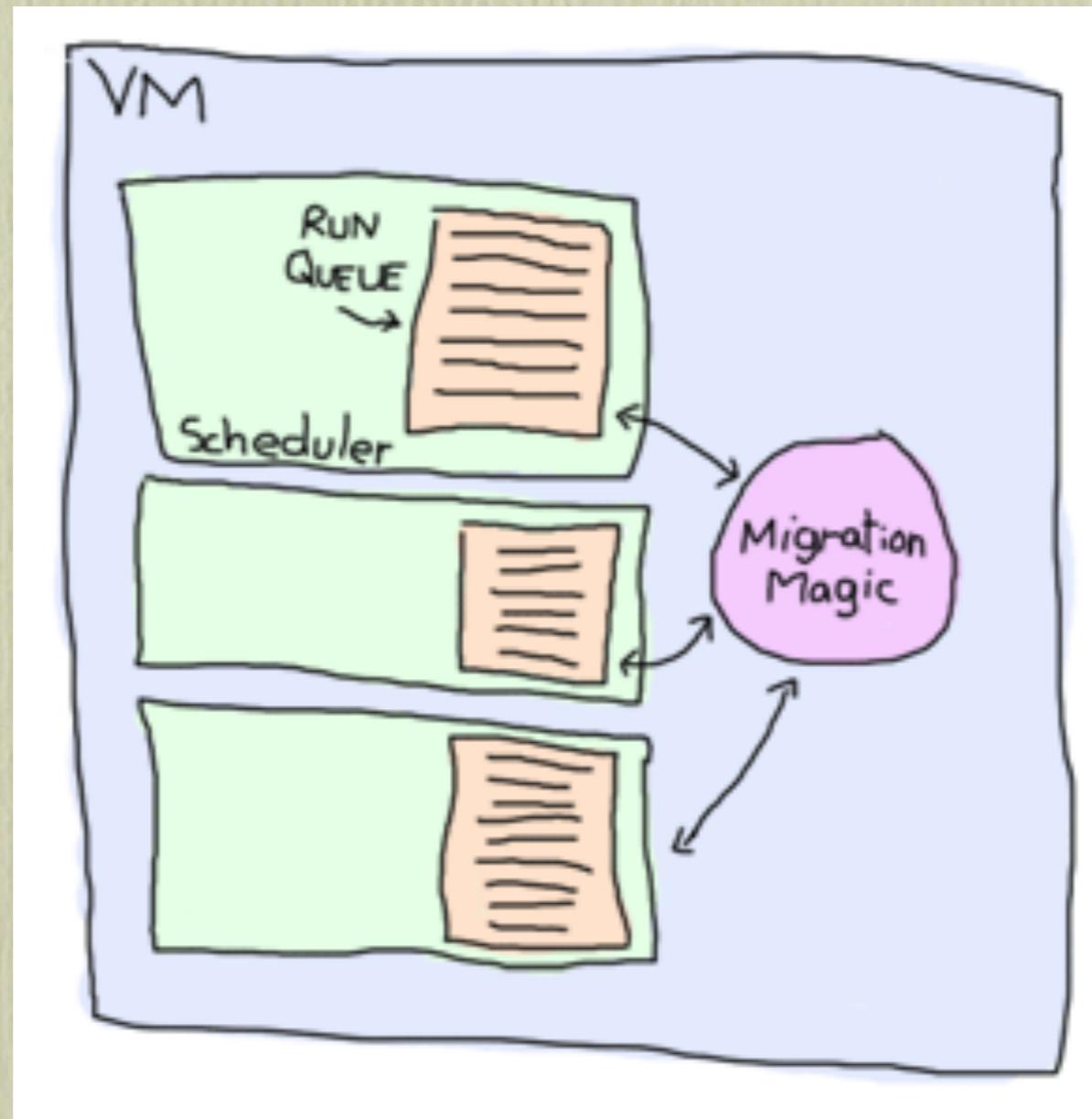
19

Procesos ligeros: Los procesos se pueden crear y destruir muy rápidamente y la máquina virtual puede cambiar entre un proceso y otro muy rápidamente también.

Memoria independiente: Los procesos no comparten memoria, de manera que se evita la necesidad de locks y que un proceso en un estado inconsistente pueda afectar a otro. También facilita la recolección de basura una vez un proceso ha terminado. Esto ayuda en la implementación de soft real-time systems ya que las pausas de GC son muy pequeñas.

Envío de mensajes: La única forma de pasar datos de un proceso a otro es enviando un mensaje, lo cual implica copiar datos de una zona de memoria a otra. Es más lento pero más seguro.

# Ejecución de procesos



20

Monday, 18 July 2011

20

Por defecto hay un scheduler por core.

Cada scheduler tiene una cola de ejecución que se ejecuta secuencialmente.

En caso de que una cola de ejecución sea mucho más grande que otra existe una lógica para mover tareas de un scheduler a otro.

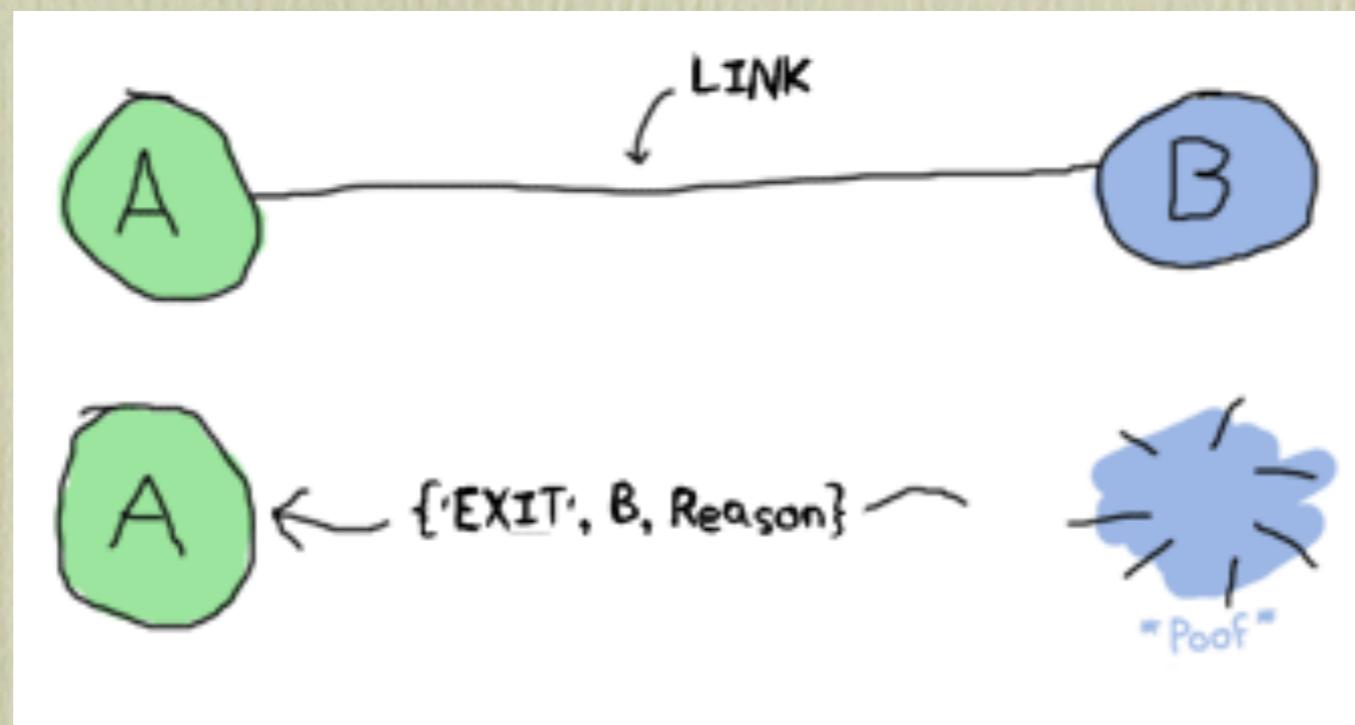
# Ejemplo de Proceso (I)

```
-module(contador).
-export([contar/0]).  
  
contar() ->  
    contar(0).  
  
contar(Valor) ->  
    receive  
        cuenta ->  
            contar(Valor + 1);  
        valor ->  
            io:format("El valor es: ~p~n", [Valor]),  
            contar(Valor);  
        Msg ->  
            io:format("Mensaje desconocido: ~p~n", [Msg]),  
            contar(Valor)  
    end.
```

# Ejemplo de Proceso (y II)

```
Eshell V5.7.5 (abort with ^G)
1> Alberto = spawn(contador, contar, []).
<0.34.0>
2> Alberto ! cuenta.
cuenta
3> Alberto ! cuenta.
cuenta
4> Alberto ! valor.
El valor es: 2
valor
5> Alberto ! cuenta.
cuenta
6> Alberto ! valor.
El valor es: 3
valor
```

# Links (I)



# Links (II)

```
padre() ->
    process_flag(trap_exit, true),
    PidHijo = spawn_link(links, hijo, []),
    io:format("Padre: ~p, Hijo: ~p~n", [self(), PidHijo]),
    recibir_mensajes().  
  
recibir_mensajes() ->
    receive
        Msg ->
            io:format("Mensaje Padre: ~p~n", [Msg]),
            recibir_mensajes()
    end.  
hijo() ->
    receive
        Msg ->
            io:format("Mensaje Hijo: ~p~n", [Msg]),
            hijo()
    end.
```

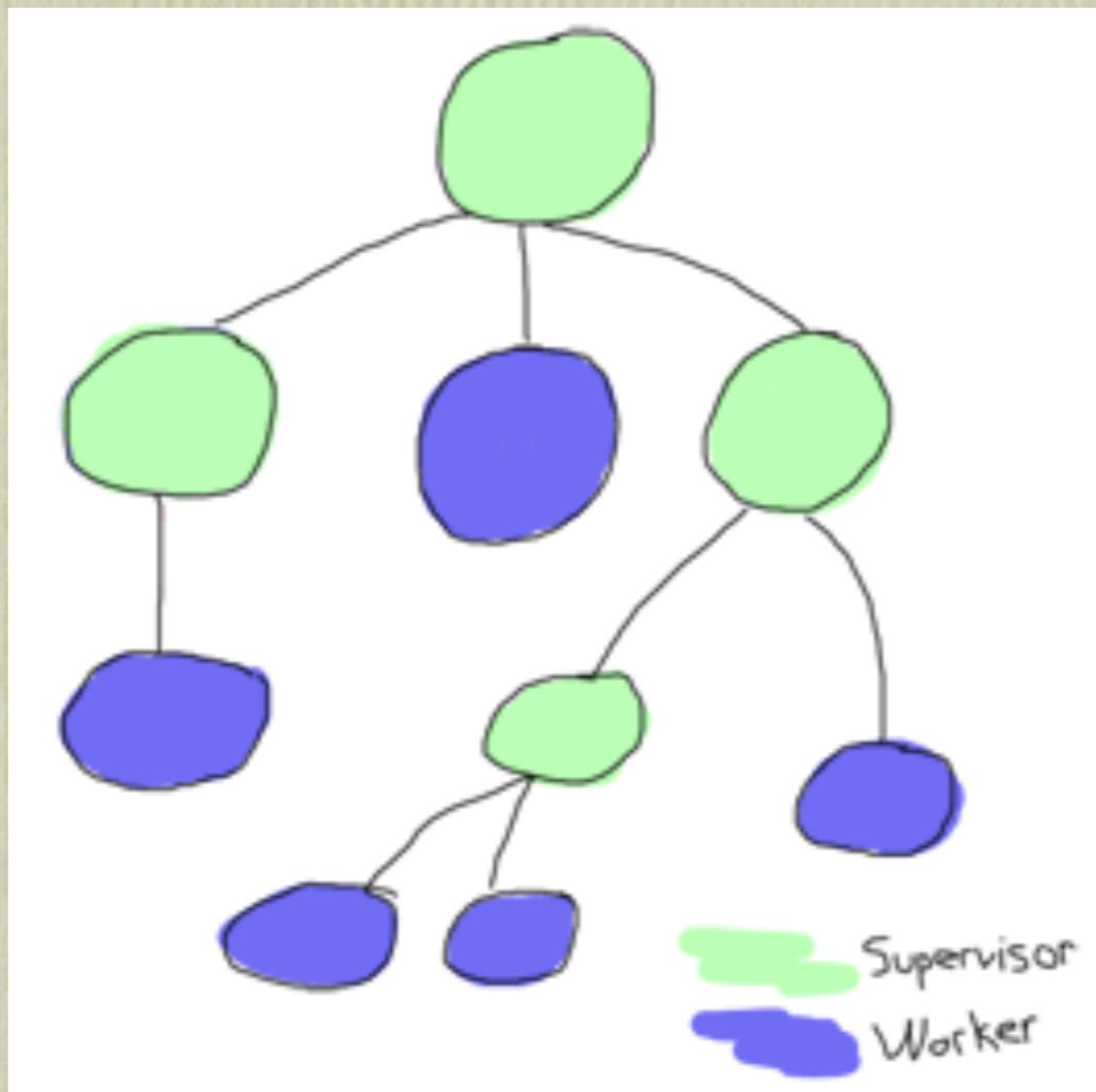
# Links (y III)

```
Eshell V5.7.5 (abort with ^G)
1> Padre = spawn(links, padre, []).
Padre: <0.34.0>, Hijo: <0.35.0>
<0.34.0>
2> Padre ! hola.
Mensaje Padre: hola
hola
3> Hijo = list_to_pid("<0.35.0>").
<0.35.0>
4> Hijo ! hola.
hola
Mensaje Hijo: hola
5> exit(Hijo, kill).
Mensaje Padre: {'EXIT',<0.35.0>,killed}
true
```

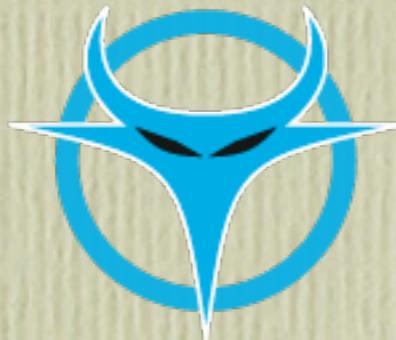
# OTP

- gen\_server, gen\_fsm, gen\_event
- applications
- supervisors

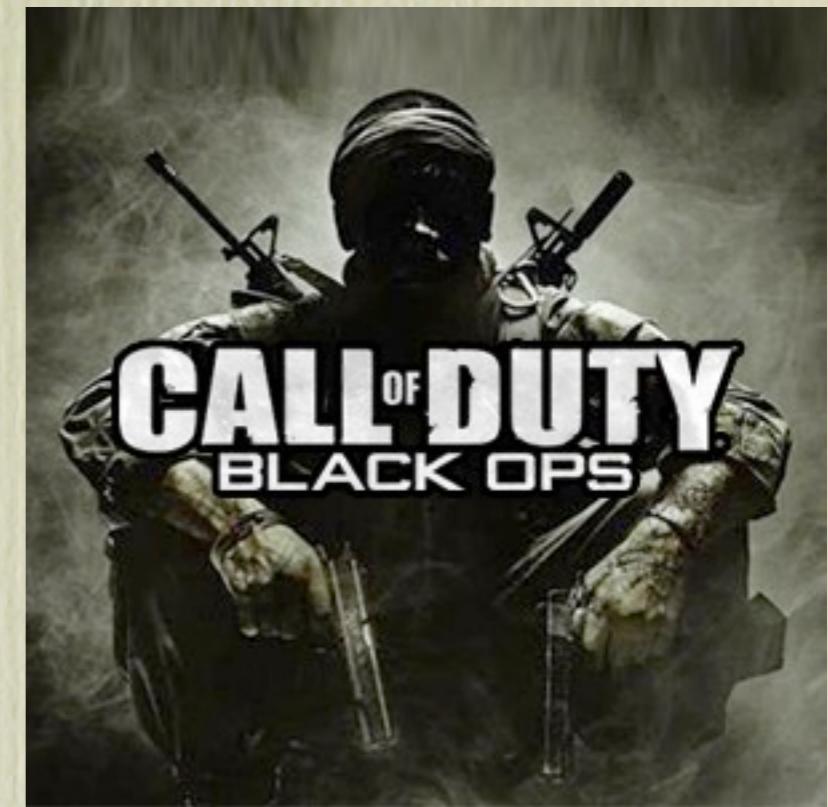
# Supervisors



# ¿Quién usa Erlang? (I)



DEMONWARE



28

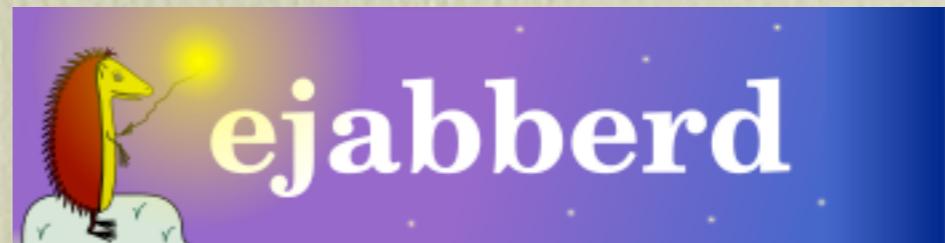
mobile  
interactive  
group

Monday, 18 July 2011

Amazon – SimpleDB utilizada para sus servicios en la nube  
Yahoo – Harvester sistema para recoger datos de múltiples fuentes

28

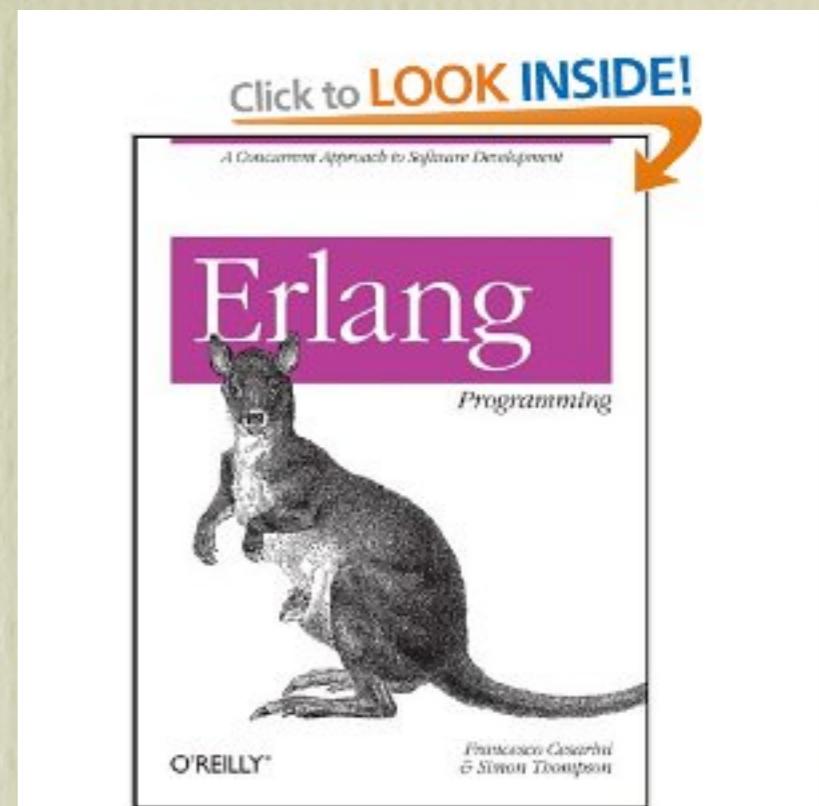
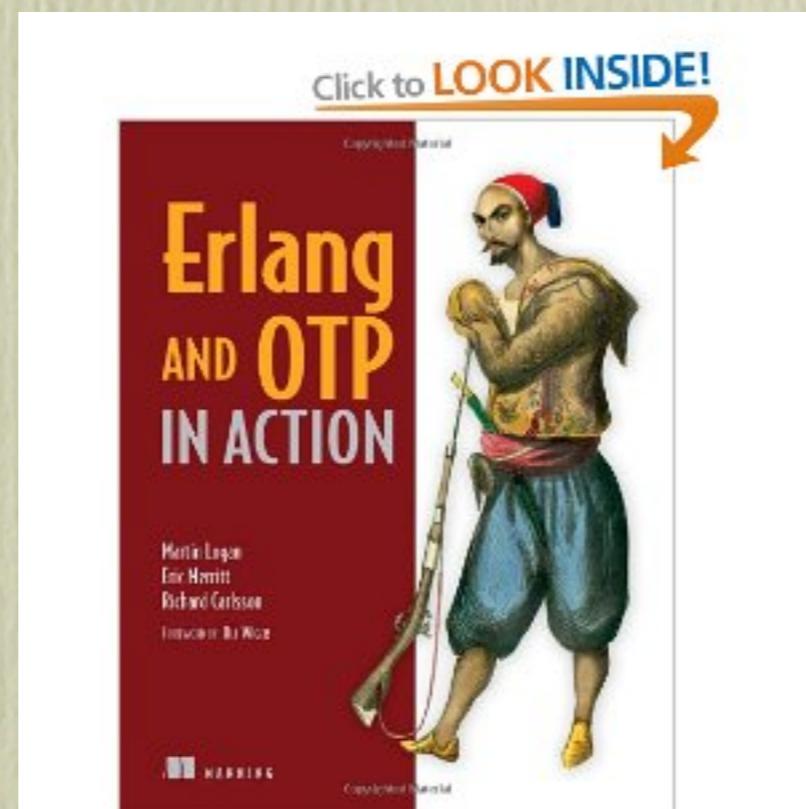
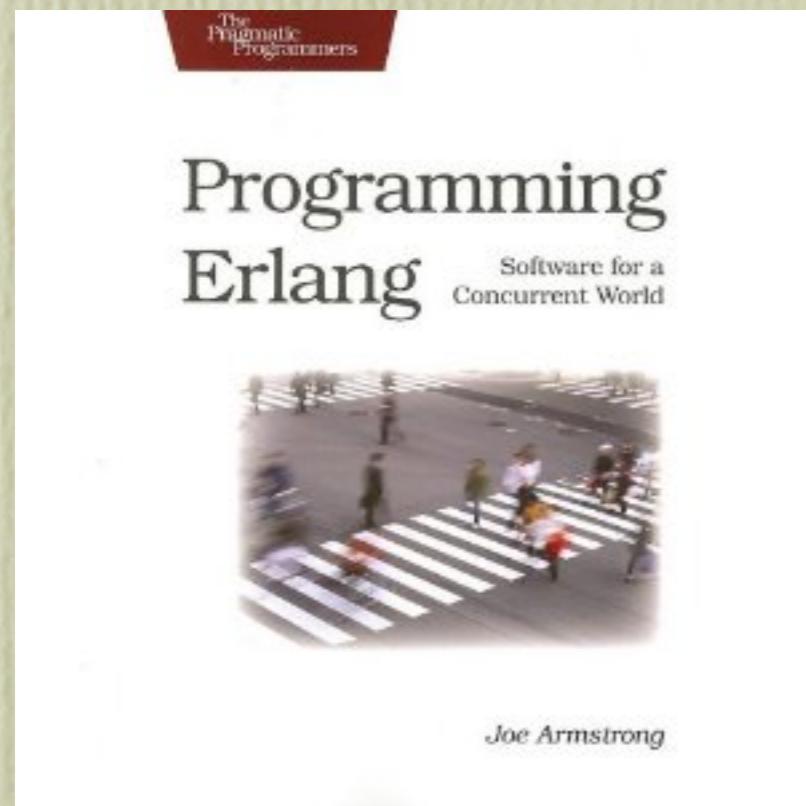
# ¿Quién usa Erlang? (y II)



# Recursos

- <http://www.erlang.org>
- <http://learnyousomeerlang.com>
- <http://trapexit.org/>
- <http://www.scribd.com/doc/44221/Thinking-in-Erlang>

# Recursos (y II)



# Rubinius

```
require 'actor'

contador = Actor.spawn do
  valor = 0
  loop do
    case Actor.receive
    when :cuenta then valor += 1
    when :valor then puts "El valor es: " + valor.to_s
    else puts "Mensaje desconocido"
    end
  end
end
```

# Rubinius (y II)

```
rbx-head :013 > contador << :cuenta
=> #<Actor:0xfb0 ....>
rbx-head :014 > contador << :cuenta
=> #<Actor:0xfb0 ....>
rbx-head :015 > contador << :valor
=> #<Actor:0xfb0 ....>
rbx-head :016 > El valor es: 2
rbx-head :017 >     contador << :xxx
=> #<Actor:0xfb0 ....>
rbx-head :018 > Mensaje desconocido
```

# Celluloid (I)

```
class Sheen
  include Celluloid::Actor

  def initialize(name)
    @name = name
    @status = :fired
  end

  def win
    @status = :winning
  end

  def current_status
    "#{@name} is #{@status}!"
  end
end
```

# Celluloid (y II)

```
>> charlie = Sheen.spawn "Charlie Sheen"
=> #<Celluloid::Actor(Sheen:0x13c0) @name="Charlie Sheen"
@status=:fired>
>> charlie.current_status
=> "Charlie Sheen is fired!"
>> charlie.win!
=> nil
>> charlie.current_status
=> "Charlie Sheen is winning!"
```

35

# Reia

36

Monday, 18 July 2011

36

Lenguaje inspirado en la sintaxis de Ruby sobre la máquina virtual de Erlang, su autor Tony Arcieri ha parado su desarrollo debido a las similitudes con Elixir que está más avanzado

<http://www.unlimitednovelty.com/2011/06/why-im-stopping-work-on-reia.html>

# Elixir

```
module Math
  def fibonacci(0)
    0
  end

  def fibonacci(1)
    1
  end

  def fibonacci(n)
    fibonacci(n - 1) + fibonacci(n - 2)
  end
end
```