

Erlang

I

Me

- Víctor Martínez
- Vorago 2006 - 2008 (Ruby on Rails)
- Mobile Interactive Group 2008 - 2012 (Ruby on Rails & Erlang)
- Homestays.com 2012 (Ruby on Rails)
- <http://twitter.com/vicmargar>
- vicmargar@gmail.com

Why Erlang?

3

Saturday, 16 June 12

I started working at Mobile Interactive Group as a Rails developers, most of their apps were related with SMS processing. So a typical app would consist of a Rails front-end plus a series of Ruby processes accessing a queue implemented in MySQL.

This has several problems: DB contention, process management, memory consumption, etc...

We decided to look into Erlang.

Erlang

Basic Aspects

4

Saturday, 16 June 12

Erlang was developed by Ericsson in the 80s. Its main feature nowadays is how it handles concurrency on multi-core but this has only been added in the last few years.

In this first part I'll talk about its basic characteristics and how it's similar or different to other languages.

What is 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.

Functional Language

- Not OO
- Functions are grouped in modules
- Immutable state
- No side-effects (almost)

Invariable Variables

```
Eshell V5.9.1 (abort with ^G)
1> A = 1.
1
2> A = 2.
** exception error: no match of right hand side value 2
```

Pattern Matching (I)

```
Eshell V5.9.1 (abort with ^G)
1> [ Head | Tail ] = ["one", "two", "three"].
["one", "two", "three"]
2> Head.
"one"
3> Tail.
["two", "three"]
```

Pattern Matching (II)

```
Eshell V5.9.1 (abort with ^G)
1> {_, S, [H | T]} = {hello, "abc", [1,2,3,4]}.
{hello, "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").
```

IO

Pattern Matching (y IV)

```
Eshell V5.9.1 (abort with ^G)
```

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

```
Guau guau!
```

```
ok
```

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

```
Miauuuuuuu!
```

```
ok
```

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

```
Grrrrrrrrrr!
```

```
ok
```

Recursion

```
-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 V5.9.1 (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]
```

List Comprehensions

```
Eshell V5.9.1 (abort with ^G)
1> [ 2 * X || X <- [1,2,3,4,5] ].  
[2,4,6,8,10]
```

Bit Syntax (I)

```
Eshell V5.9.1 (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
```

Bit Syntax (II)

TCP Header																																																	
Bit offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31																	
0	Source port																Destination port																																
32	Sequence number																																																
64	Acknowledgment number (if ACK set)																																																
96	Data offset	Reserved	N S R	C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I N	Window Size																																					
128	Checksum																Urgent pointer (if URG set)																																
160	Options (if Data Offset > 5) ...																															padding																	
...																																																	

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

Hot Code Loading

I7

Saturday, 16 June 12

Erlang

Concurrency

Concurrency Vs Parallelism

19

Saturday, 16 June 12

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).

Escalability

20

Saturday, 16 June 12

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.

Fault Tolerance (Let it Crash)

21

Saturday, 16 June 12

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.

Actor Model

- Lightweight Processes
- Independent Memory
- Asynchronous Messages
- Async IO

22

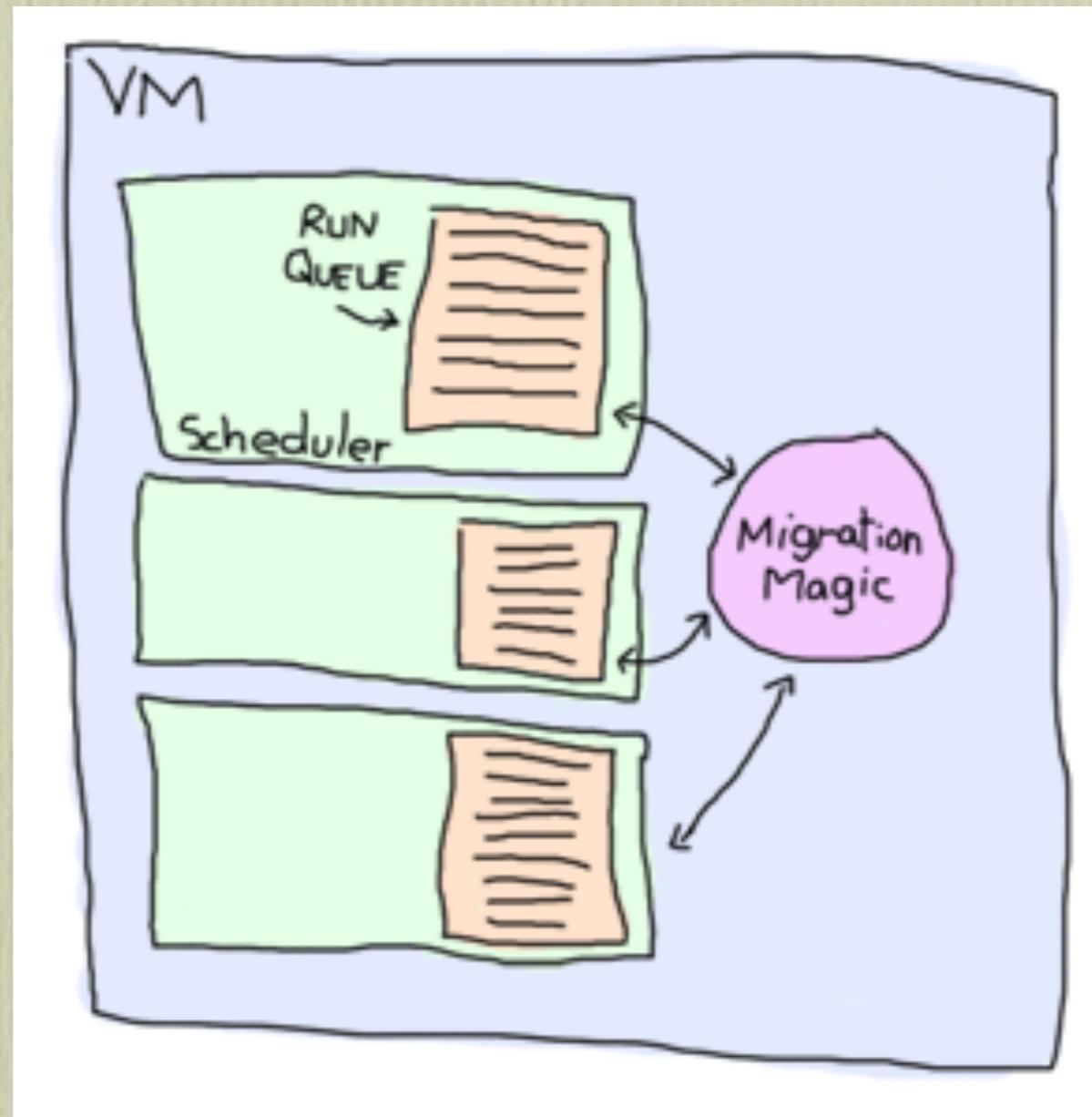
Saturday, 16 June 12

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.

Process Execution



23

Saturday, 16 June 12

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.

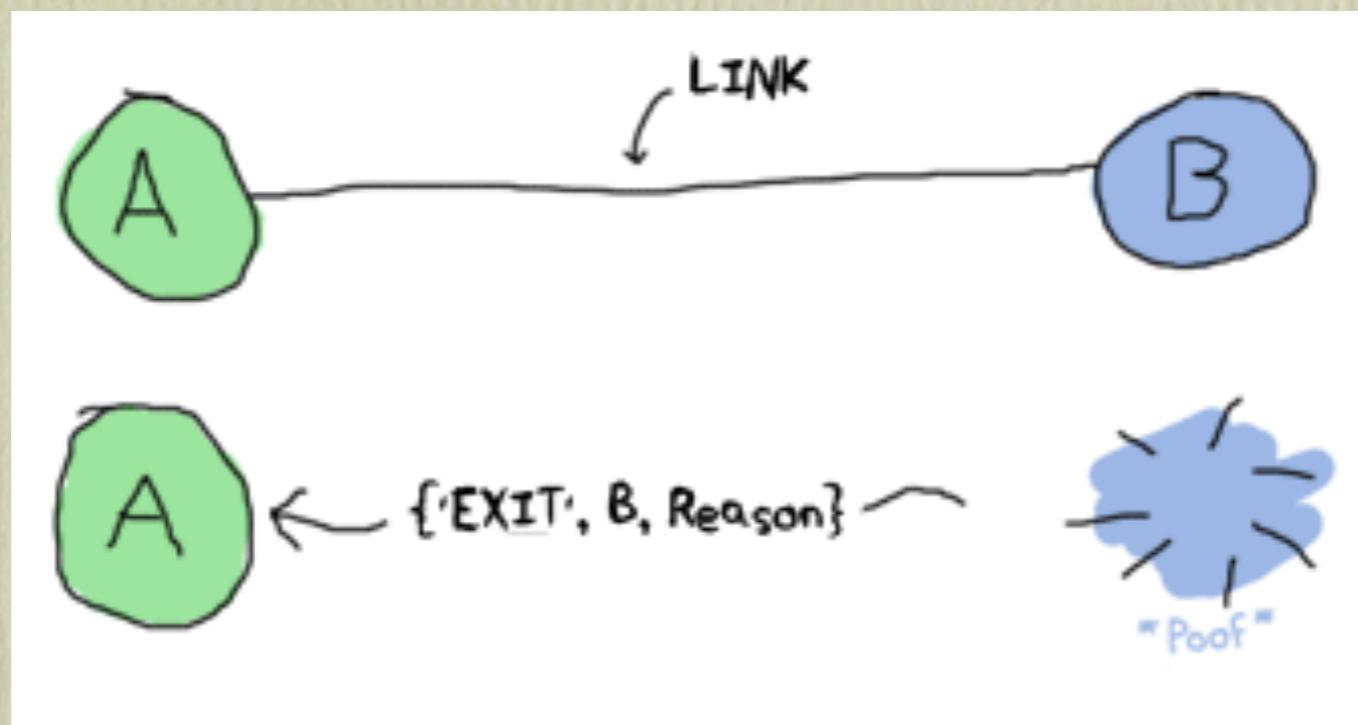
Process Example (I)

```
-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.
```

Process Example (y II)

```
Eshell V5.9.1 (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.
```

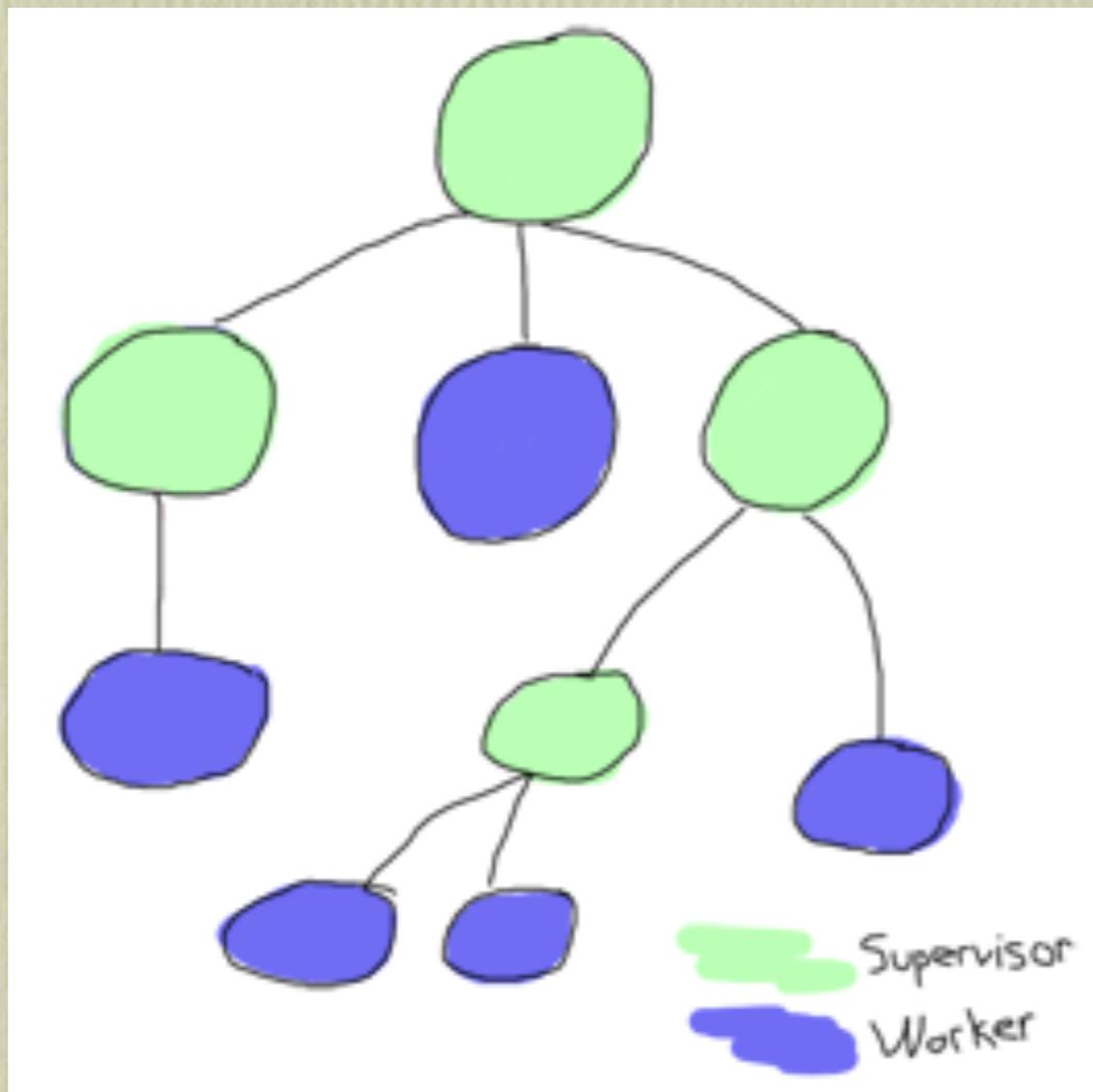
Links (y III)

```
Eshell V5.9.1 (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

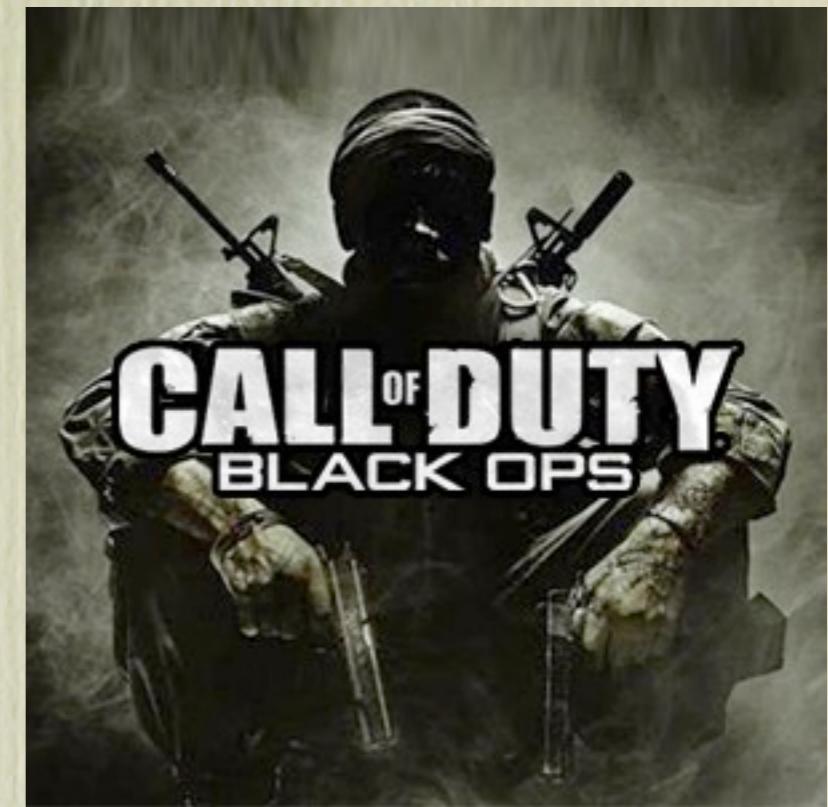
Supervisors



Who uses Erlang? (I)



DEMONWARE



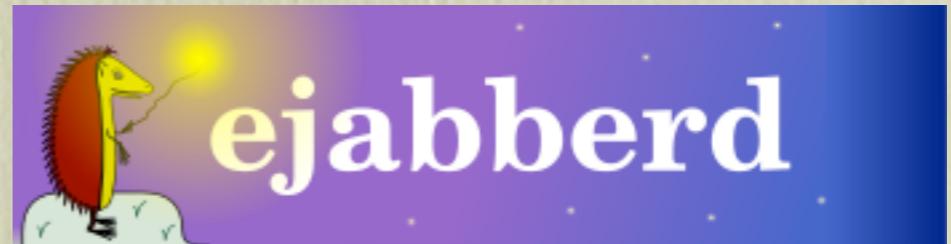
31

mobile
interactive
group

Saturday, 16 June 12

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

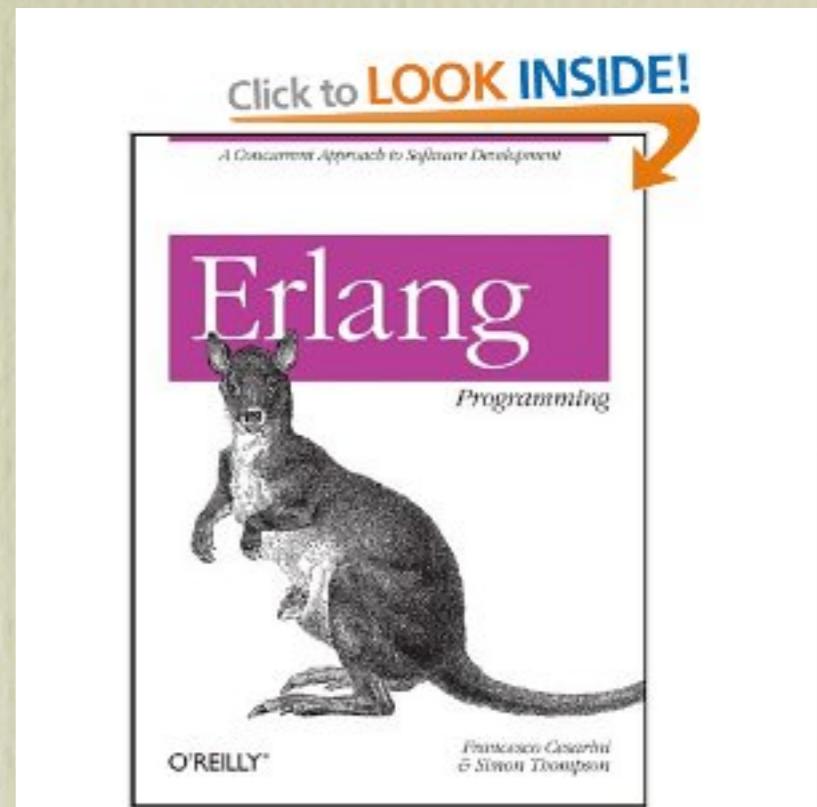
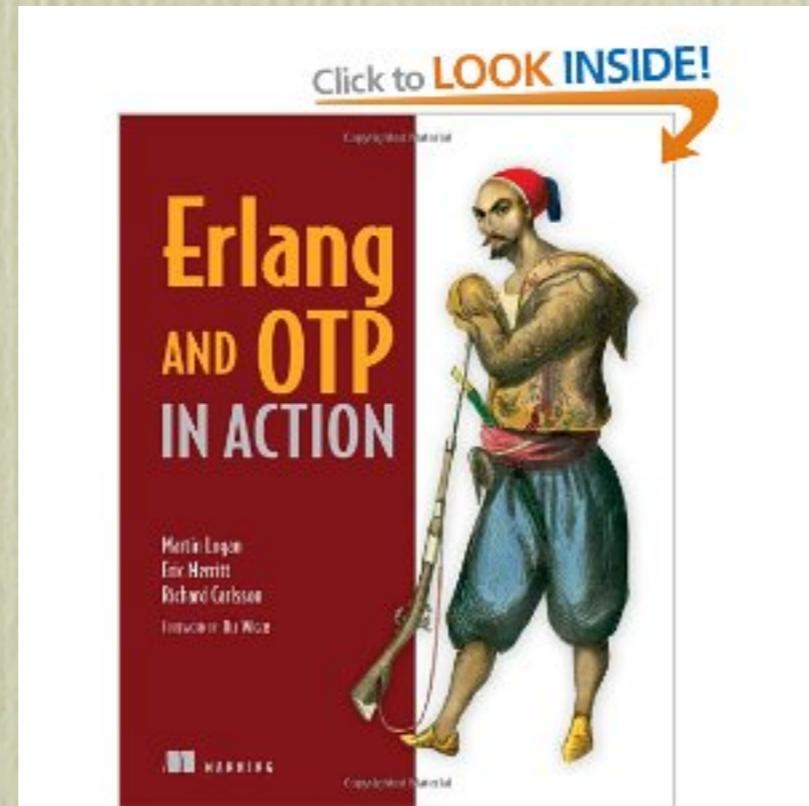
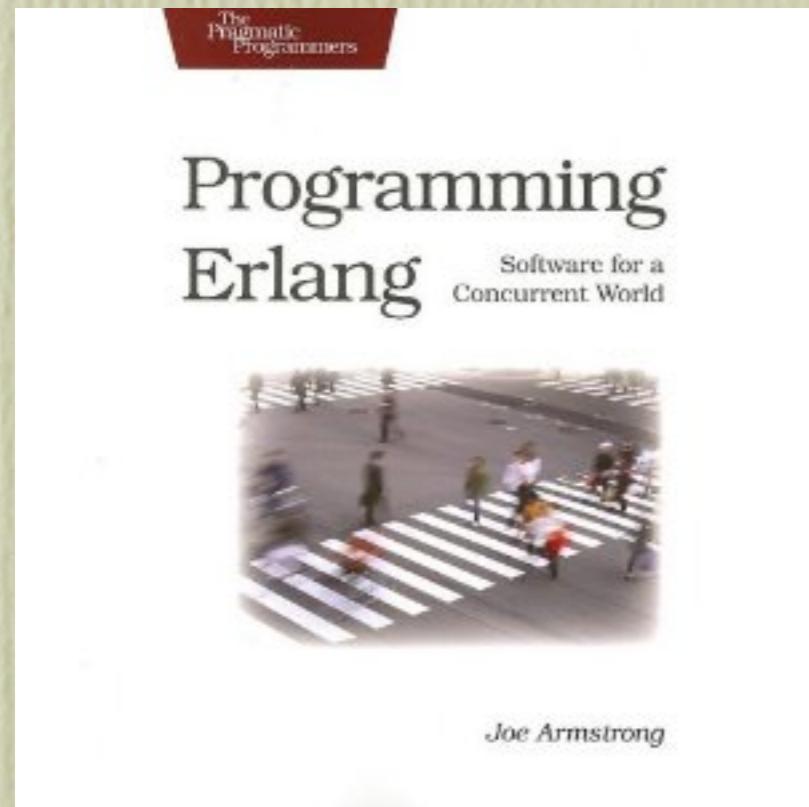
Who uses Erlang? (y II)



Resources

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

Resources (y II)



The End

<https://github.com/vicmargar>