Erlang

“Software for a concurrent world”

CWI Amsterdam

Arjan Scherpenisse
14 december 2012

arjan@miraclethings.nl

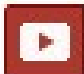
Agenda

- Intro
- Language essentials
- Concurrency / failure
- OTP
- VM / GC / optimisations / pitfalls

0:54 / 5:50

The Ghetto

gar1t · 11 videos



Subscribe 752

3,554

61 0

What is Erlang?

+

Open
Source

=

History

- Created in ’86
- Open-sourced in ’98
- “Programming Erlang” published in ’07
- Taking off in popularity (3 more books on the way)

github
SOCIAL CODE HOSTING

icrosoft

YAHOO!

RabbitMQ
en Source Enterprise Messaging

Erlang Philosophy

- The world is concurrent
- Things in the world dont share data
- Things communicate with messages
- Things fail

- Joe Armstrong

Language essentials

She's got the look

```
-module(math_o_matics).
-export([square/1]).

square(X) ->
    X * X.
```

Data types

- Integer / float
- Atoms
- Tuples
- Lists
- Binaries

… so where are strings?

Datatypes (cont)

- Functions
- Pids
- Ports

The "fun" in functional

- Anonymous functions
- Used for higher-order programming

```
Square = fun(X) -> X * X end.

Cube = fun(X) -> Square(X) * X end.
```

List Comprehensions

- Takes an expression and a set of qualifiers and returns another list (like Python's)
- Looks like: `[X || Q1, Q2, ... Qn ]`

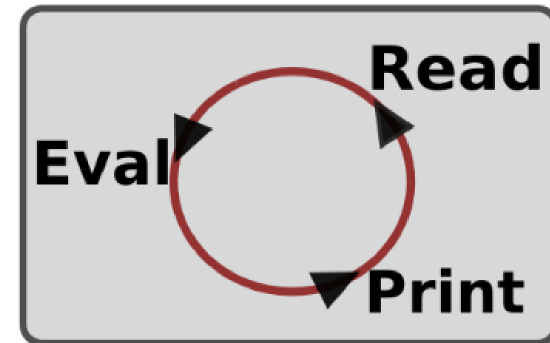
```
qsort([]) -> [];
qsort([Pivot|T]) ->
  qsort([X || X <- T, X < Pivot])
  ++ [Pivot] ++
  qsort([X || X <- T, X >= Pivot]).
```

Recursion (important)

- Function directly or indirectly calling itself
- It's how you iterate!
- Tail optimization enables long running loops (eliminates unbounded stack growth)

The Shell (super important)

- Single most important tool for Erlang developers!
- Use to experiment, learn, prove/disprove
- Important to know:
  - `erl` starts the shell!
  - `ctrl-c, ctrl-c` stops the shell abruptly!
  - `q().` shuts down cleanly
  - Expressions always end in a period!



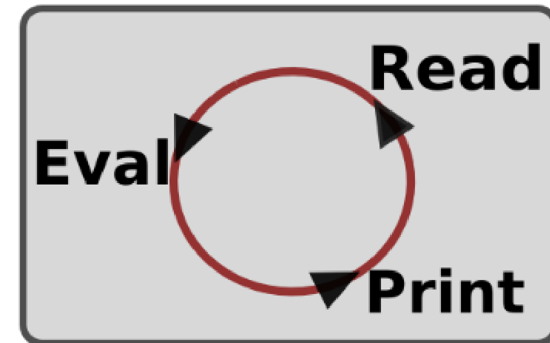
Guards

- Simple tests against a pattern matching
- Can make code more concise

```
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.
```

The Shell (super important)

- Single most important tool for Erlang developers!
- Use to experiment, learn, prove/disprove
- Important to know:
  - `erl` starts the shell!
  - `ctrl-c, ctrl-c` stops the shell abruptly!
  - `q().` shuts down cleanly
  - Expressions always end in a period!



Side-Effects

- Erlang is not pure-functional language
- Side effects:
  - *Messages* - Pid ! Msg – send message to process Pid
  - *Signals* - i.e. your linked process is dead
  - *Exceptions*
  - *I/O*
  - *Process Dictionary* - similar to TLS (thread local storage)
  - *ETS / DETS* - fast in-memory / persistent lookup tables
  - *Mnesia* - distributed realtime database (STM)

Concurrency / failure

Concurrency

- **spawn, spawn_link**
- **!, receive**

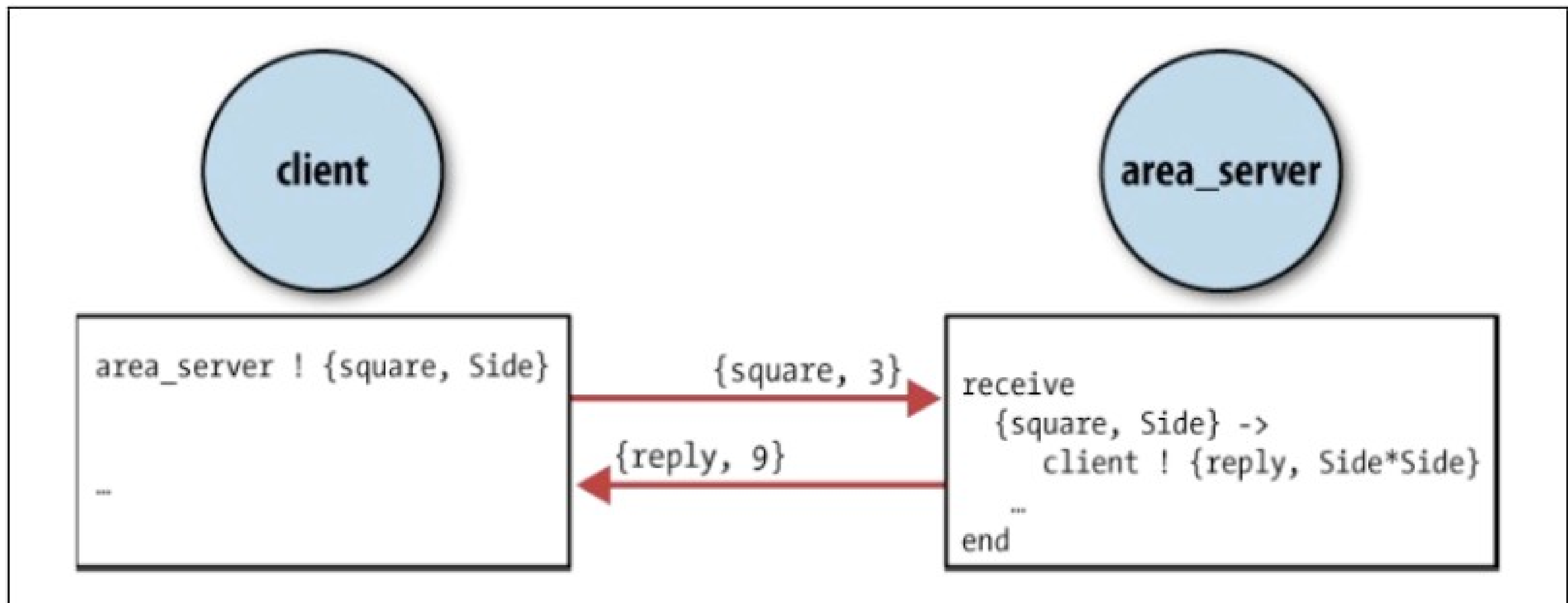
- Concurrency properties
  - Lightweight process model in Erlang VM
    - (not OS processes or threads)
  - No data sharing (inherited from functional nature)
  - Built in distributed system support
    - Safe and conditional messaging
    - Process monitoring and failure notification

Simple Process Communication

```
1> F = fun() -> receive {From, Msg} ->
    From ! {got, Msg, thanks} end end.
#Fun<erl_eval.20.67289768>
2> Pid = spawn(F).
<0.36.0>
3> flush().
ok
4> Pid ! {self(), "hey"}.
{<0.33.0>,"hey"}
5> flush().
Shell got {got, "hey", thanks}
ok
```

Concurrency and Messaging

- Model: create a process for every independent activity
  - Processes are lightweight
- Communicate between processes using messages
- (Crazy) Example to get in the Erlang mindset



The Actor Model

- Actors communicate through messages
- Actors make local decisions as to handle messages
  - Send more messages
  - Create more actors
  - Decide how to respond to future messages (change local state)
- Actor model prohibits shared state and is a nice formalism for parallel computing

The Erlang Process Model

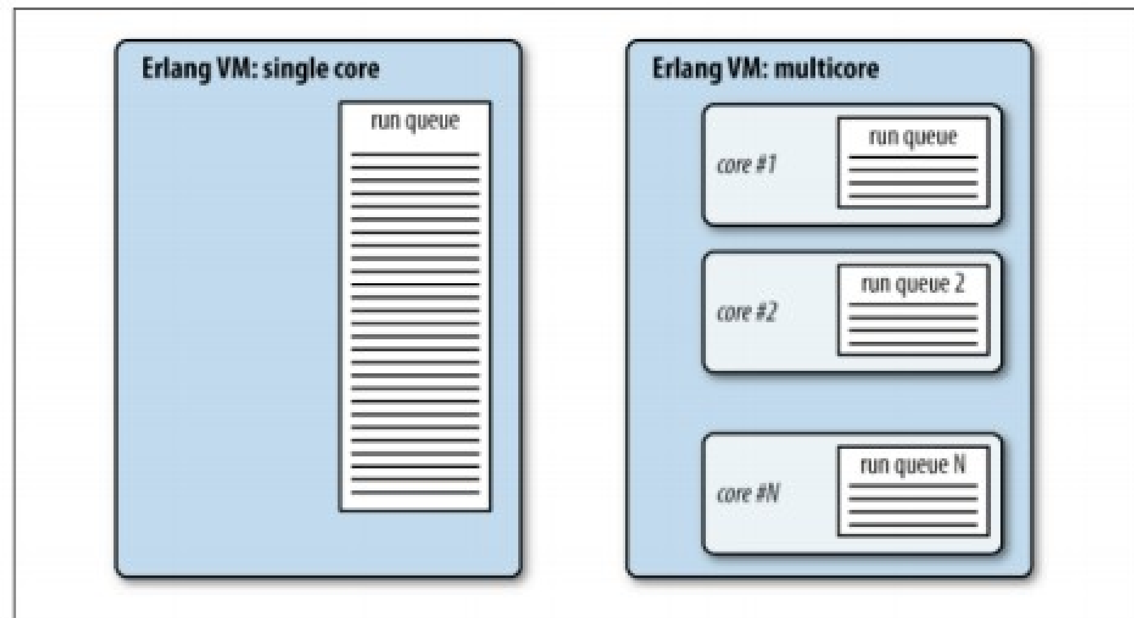
- A concurrent execution context in the Erlang VM
  - Stack, yes in so much as Erlang uses a stack.
  - No heap. No memory allocation at all in fact.
- Processes do not map to OS processes or threads
  - Consume no resources in the underlying OS
- Can create thousands or millions of processes on a single Erlang instance
  - Initially processes have small memory footprints
  - Idle processes await receive use no resources (until a message is delivered to them)

Process Scheduling

- Each process get a number of *reductions*
  - Reductions are processing credits
  - Each operation decreases the reductions
- Processes are preempted when
  - They perform a receive (await a message)
  - Reductions go to zero
  - Blocking/slow calls (e.g. I/O, system calls) are implemented via send/receive so they get handled automatically
- Realizing soft-real time
  - Small reductions (about 2000) per process
  - Lightweight process switching
  - No memory protection issues

Multi-Core Support

- Treat a multicore machine as a distributed system
  - Separate run queues, independent scheduling across cores
- No changes to actor model
  - Processes communicate through messages
- VM can leverage shared memory
  - For code, literals, process information, messaging
- Other systems things
  - Process migration
  - Load balancing



LET IT CRASH

BY

Process Error Handling

- Features underlying built-in fault tolerance
- Ethos
  - “Let it crash and let someone else deal with it”
  - “Crash early” (Crash often?)
- Process linking and monitoring
  - Notification if processes fail or are unavailable/disconnected
  - Symmetric (linked) and assymetric (monitor)
  - Exit signals (notification messages when processes terminate)

link/1 and spawnlink/3

- link/1 connects to an existing process
- spawnlink/3 spawns a process linked to current proc.
  - This is atomic and almost always preferable to spawn then link

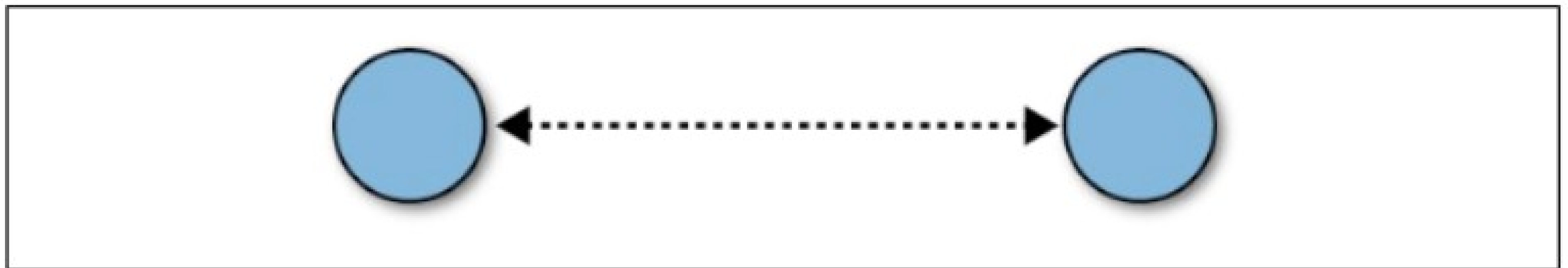


Figure 6-1. Linked processes

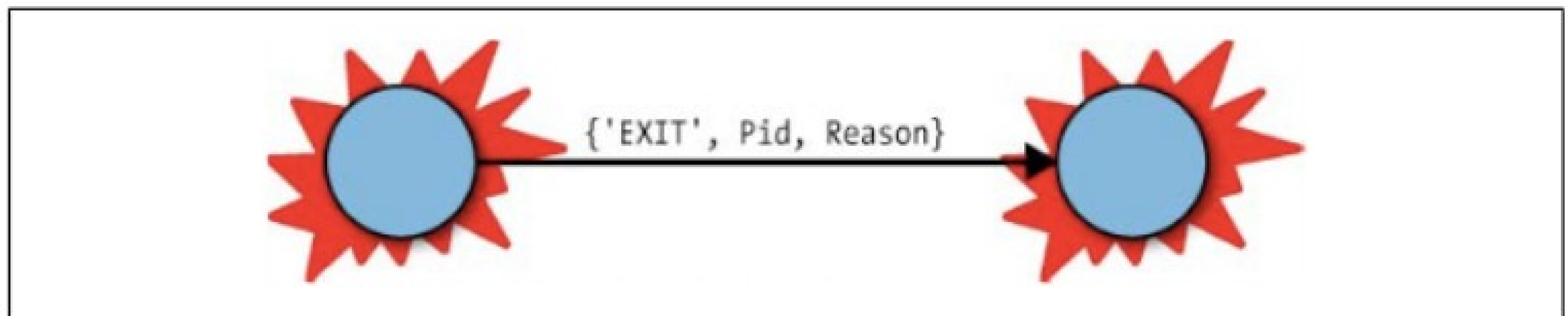


Figure 6-2. Exit signals

Error Propagation

- link/1 is a good technique to propagate errors
  - To fail all related process (the Erlang way) and reveal errors
  - To have a monitor restart

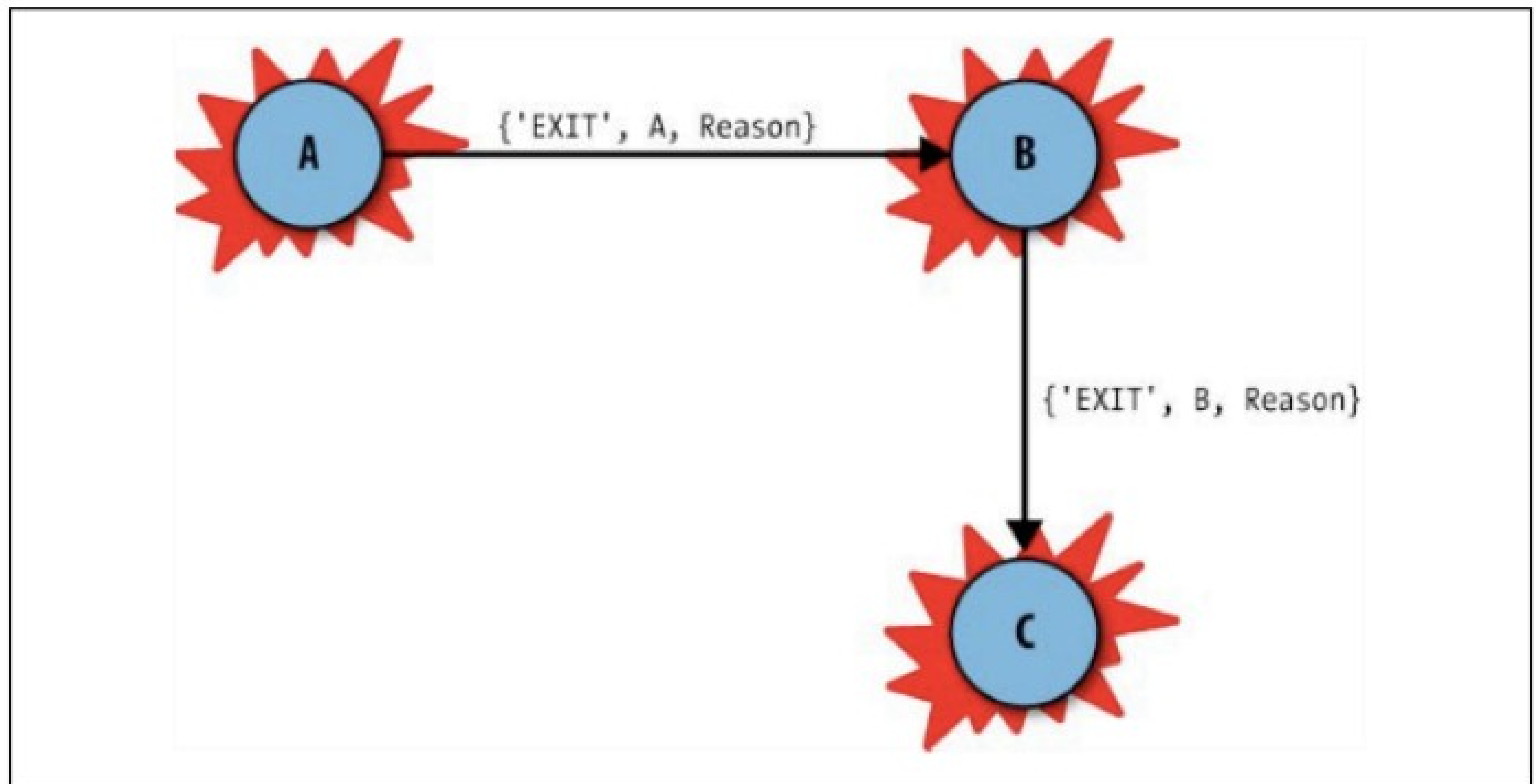
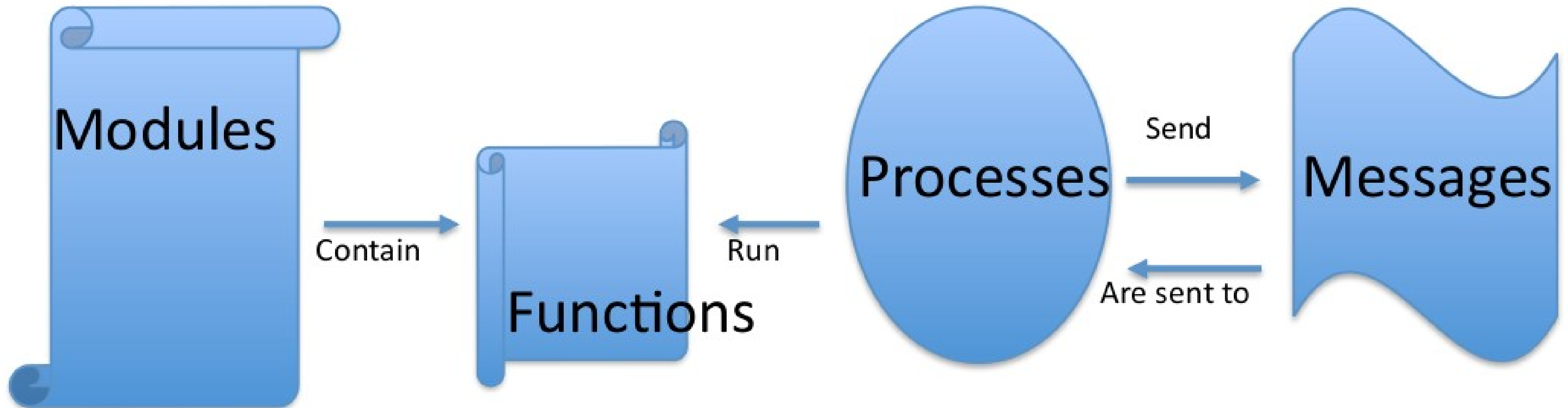


Figure 6-3. Propagation of exit signals

Fundamental pieces



OTP: Open Telecom Platform

OTP=Open Telecom Platform

- Library of implemented application frameworks
  - Users customize behavior by callback functions
- Client/server
  - OTP handles timeouts, safegaurds, message protocol, process registration
- Process monitoring
  - OTP handles fault detection, automatic restart, start and shutdown, managing process state
- Effective code reuse!

OTP

- application
  - Contains independent code, multiple applications per Erlang *node*
- supervisor
  - Supervises worker processes and supervisors
- gen_server
  - Basic work unit

GC / optimisation / pitfalls

Performance

- Processes are cheap
- Data copying is expensive
- Handle much of request in single process
- Cache locally in process dict
- Reduce size of messages
- Bigger binaries (> 64 bytes) are shared and reference counted, make use of them.
- String processing is expensive
- Keep eye on process inboxes

Garbage Collection

- *per process,* no global gc sweep = no locking of vm
- can specify per-process initial heap size
- prevent big GCs
  - keep big data outside process
  - or in process dict
  - large binaries = external, use refcounting

Common Pitfalls

- Message inbox overflowing
- Atom table injection / crash
- VM tweaks needed for
  - nr of open file descriptors
  - nr of processes
  - heap size

Erlang is a good fit for:

- Irregular concurrency:
  - Task-level
  - Fine-grained parallelism
- Network servers
- Distributed systems
- Middleware:
  - Parallel databases
  - Message Queue servers
- Soft Realtime / Embedded applications
- Monitoring, control and testing tools

Not so good fit for:

- Concurrency for synchronized parallel execution
  - Data Parallelism
- Floating-point intensive code (HPC)
- Text Processing / Unicode
  - Unicode support now much better
- Traditional GUI (supports Tk and wxWidgets)
- Hard Realtime applications
- Extensive interop with other languages/VMs
  - improvement here with NIFs and Erjang - Erlang on JVM

Outro

What *I* mainly use it for

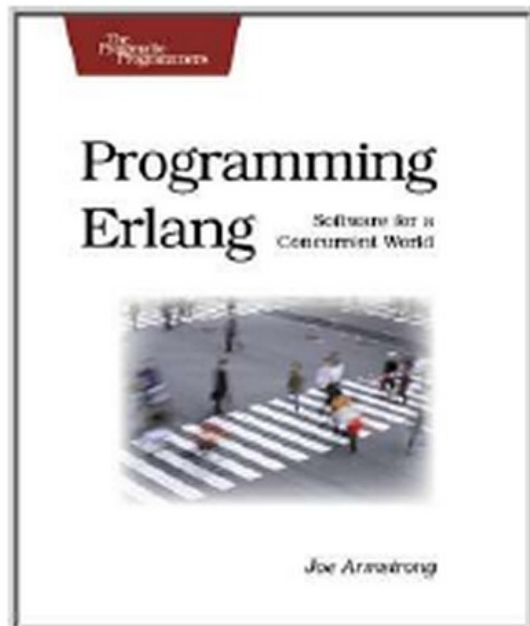
- Full-stack web development
  - http://zotonic.com/
- HA backend systems

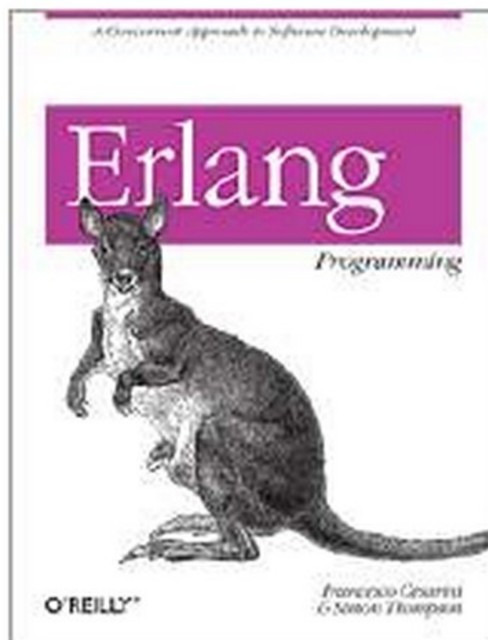
Preventing stuff like this →
:-)

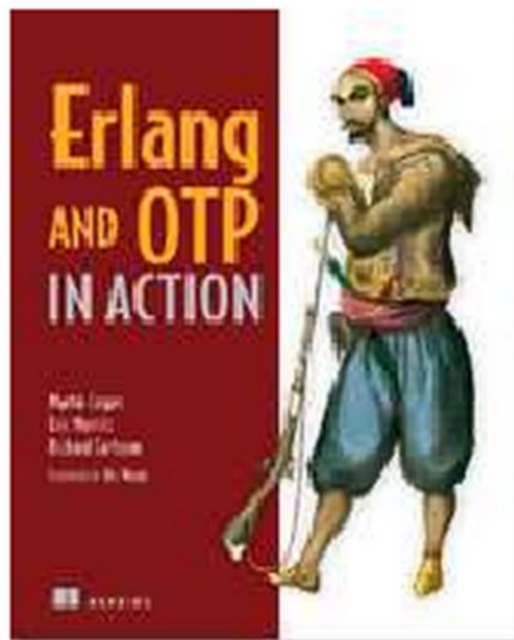
Tools

- IDE:
  - Emacs, Distel, Plugins for Eclipse and Netbeans
- Testing:
  - e-unit, proper, meck, triq, QuickCheck (paid)
- Static code analyzer
  - Dialyzer
- Debug and trace
  - Built-in in VM
- Build:
  - emake, rebar
- Package Managers:
  - rebar, agner, sinan, CEAN

Books







Sources of Inspiration

- Erlang - the ghetto
- Intro to erlang (slideshare)
- Erlang - message passing concurrency (slideshare)
- Erlang concepts (slideshare)
- Concurrency and Paralellism in Erlang (hopkins univ)

Thanks!

Arjan Scherpenisse
arjan@miraclethings.nl

http://twitter.com/acscherp
http://miraclethings.nl/