Let It Crash...
Except When You Shouldn't

Steve Vinoski
Verivue, Inc.
Westford, MA USA
vinoski@ieee.org

QCon London
10 March 2011

About This Talk

- Explore Erlang's "Let It Crash" approach to failure handling
- I don't assume you know Erlang, so there'll be some explanation of some core Erlang concepts
- Focus on a couple problem areas that aren't well documented and that you usually learn the hard way

Fail Constantly

- Netflix “Chaos Monkey”
- Kills randomly kills things within Netflix’s AWS infrastructure to make sure things keep running even with failures
- “Best way to avoid failure is to fail constantly”
- http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html

Defensive Programming

- Write code to solve the actual problem
- Then try to think of everything that can go wrong, especially with inputs
- And then write defensive code to catch and handle all possible errors and exceptions

Defensive Holes

- The more code you have, the more bugs you have
- Obscures the business logic, making it hard to read, extend, and maintain
- Error handling code is often incomplete and inadequately tested
- It's hard to defend against every possibility

Let It Crash

- From Joe Armstrong's doctoral thesis:
  - *Let some other process do the error recovery.*
  - *If you can't do what you want to do, die.*
  - *Let it crash.*
  - *Do not program defensively.*

Erlang's Better Way

- Provides features that let you address fault tolerance from the start
- Cheap lightweight processes
- Process linking and monitoring
- Workers and supervisors
- Hierarchical supervision
- Distribution/clustering (not covered)

Cheap Processes

- It’s practical to have hundreds of thousands in a single Erlang VM
- Fast starting
- Small footprint
- Isolated, reachable by message passing

Process Linking

- Erlang supports bidirectional links between processes
- If a process dies abnormally, linked processes receive an exit signal and by default also die
- Processes can trap exits to avoid dying when a linked process dies

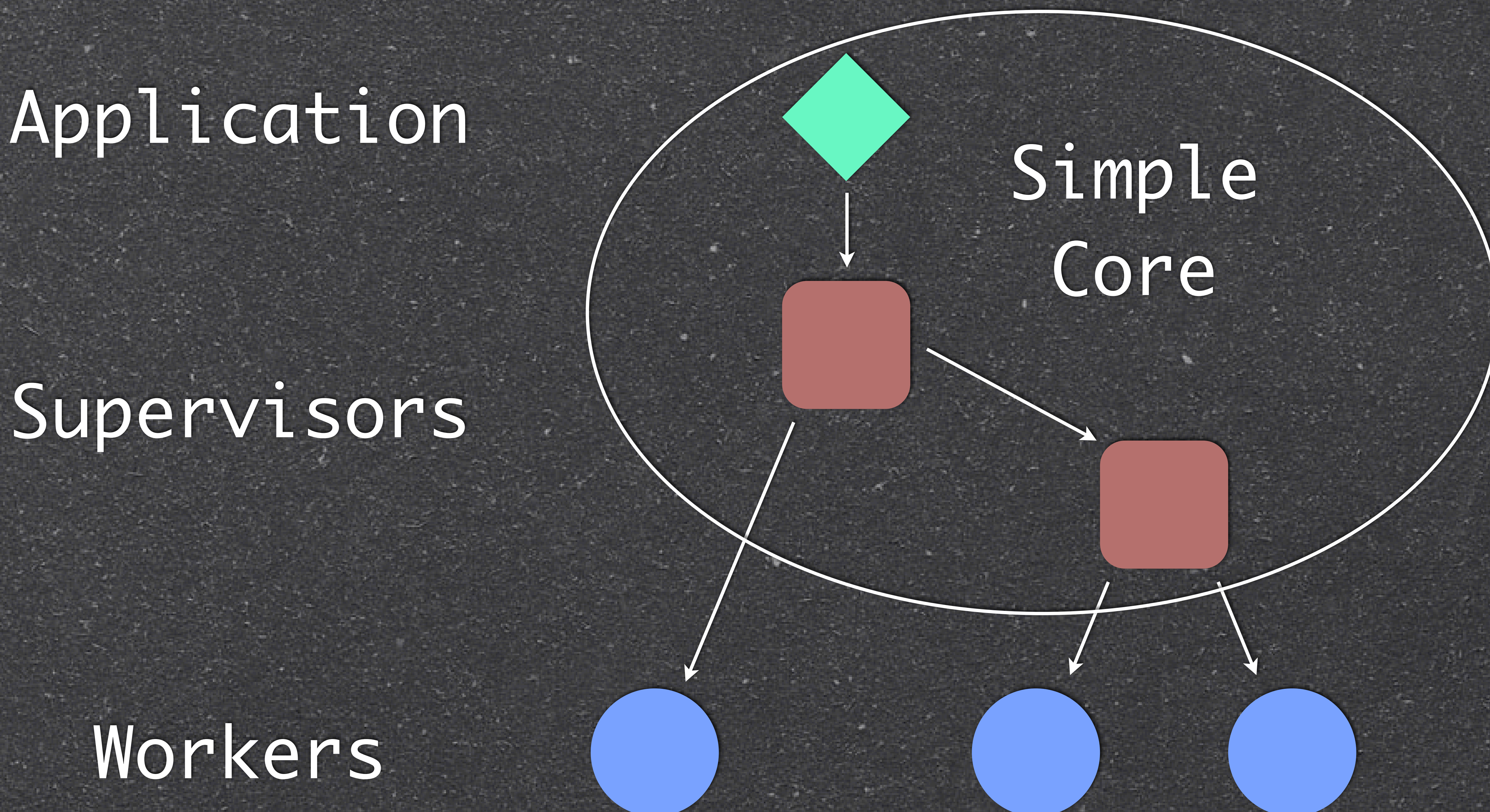
Workers & Supervisors

- Workers implement application logic
- Supervisors:
  - start child workers and supervisors
  - link to the children and trap exits
  - take action when a child dies, typically restarting one or more children

Startup Sequence

- Hierarchical sequence
- Application controller starts the app
- App starts supervisor
- Supervisor starts children
- Workers are typically instances of OTP “behaviors,” frameworks that support an “init” function called during startup

Application, Supervisors, Workers



“Let It Crash” Gone Wrong

“Let It Crash” Gone Wrong

- Production web video delivery system
- Tracking paid video subscriber usage
- During an interactive debug session, looked up a random subscriber several times
- When that subscriber logged out, the lookup crashed the whole data table.
- All usage data lost. Oops.

Moral of the Story

- Failed to follow the "Principle of Least Surprise"
- Probably not what Joe Armstrong meant
- "Let It Crash" is not:
  - a (long-term) design crutch
  - an excuse for losing vital data

Handle what you can,
and let someone else
handle the rest.

Erlang Term Storage (ets)

- In-memory key-value storage for Erlang terms
- Concurrency safe, very fast
- Each ets table is owned by a process
- Not garbage collected, either deleted explicitly or destroyed when owner dies

What Went Wrong?

- Subscriber data stored in ets table
- Subscriber tracking process did not handle a failed ets lookup
- Resulting exception took down the tracking process
- When the process died, it took the subscriber data table down with it

Avoid Losing ets Data

- When you just “Let It Crash” you lose your ets tables by default
- If this isn’t what you want, the alternatives are straightforward

Option: Name an Heir

- When creating the table, specify a process to inherit the table if the owner dies
- Heir process receives this message if owner dies:

  {'ETS-TRANSFER', TableId, Owner, HeirData}

Option: Give It Away

- A process creating an ets table can give it away to another process
- New owner gets the message below:

{'ETS-TRANSFER',Tab,Owner,GiftData}

Option: Table Manager

- Have the supervisor create a process whose sole job is to manage the ets table
- Process is doing so little that failure is extremely unlikely
- Table can be public to allow other processes to read and write

Or, a Combination

- Table manager links to the table user process, and traps exits
  - creates the table and makes itself the heir
  - gives it away to the user process
- If failure, manager gets the table back
- Rinse and repeat

Combination Example

```
1> process_flag(trap_exit, true).
false
```

Combination Example

```
1> process_flag(trap_exit, true).
false
2> T = ets:new(foo, [{heir, self(), undefined}]).
16400
```

Combination Example

```
1> process_flag(trap_exit, true).
false
2> T = ets:new(foo, [{heir, self(), undefined}]).
16400
3> P = spawn_link(fun() -> F = fun(Fn) -> receive exit -> ok;
3> M -> io:format("~p~n", [M]), Fn(Fn) end end, F(F) end).
<0.36.0>
```

Combination Example

```
1> process_flag(trap_exit, true).
false
2> T = ets:new(foo, [{heir, self(), undefined}]).
16400
3> P = spawn_link(fun() -> F = fun(Fn) -> receive exit -> ok;
3> M -> io:format("~p~n", [M]), Fn(Fn) end end, F(F) end).
<0.36.0>
4> ets:give_away(T, P, undefined).
{'ETS-TRANSFER',16400,<0.31.0>,undefined}
true
```

Combination Example

```
1> process_flag(trap_exit, true).
false
2> T = ets:new(foo, [{heir, self(), undefined}]).
16400
3> P = spawn_link(fun() -> F = fun(Fn) -> receive exit -> ok;
3> M -> io:format("~p~n", [M]), Fn(Fn) end end, F(F) end).
<0.36.0>
4> ets:give_away(T, P, undefined).
{'ETS-TRANSFER',16400,<0.31.0>,undefined}
true
5> P ! exit.
exit
```

Combination Example

```
1> process_flag(trap_exit, true).
false
2> T = ets:new(foo, [{heir, self(), undefined}]).
16400
3> P = spawn_link(fun() -> F = fun(Fn) -> receive exit -> ok;
3> M -> io:format("~p~n", [M]), Fn(Fn) end end, F(F) end).
<0.36.0>
4> ets:give_away(T, P, undefined).
{'ETS-TRANSFER',16400,<0.31.0>,undefined}
true
5> P ! exit.
exit
6> flush().
Shell got {'ETS-TRANSFER',16400,<0.36.0>,undefined}
Shell got {'EXIT',<0.36.0>,normal}
```

Combination Example

```
1> process_flag(trap_exit, true).
false
2> T = ets:new(foo, [{heir, self(), undefined}]).
16400
3> P = spawn_link(fun() -> F = fun(Fn) -> receive exit -> ok;
3> M -> io:format("~p~n", [M]), Fn(Fn) end end, F(F) end).
<0.36.0>
4> ets:give_away(T, P, undefined).
{'ETS-TRANSFER',16400,<0.31.0>,undefined}
true
5> P ! exit.
exit
6> flush().
Shell got {'ETS-TRANSFER',16400,<0.36.0>,undefined}
Shell got {'EXIT',<0.36.0>,normal}
```

Another Example

TCP Connections

```
{ok, Socket} = gen_tcp:connect(...),
```

- Q: What happens if connect fails?
- A: It returns {error, Reason}

Result

{ok, Socket} = gen_tcp:connect(...)

if failure, means

{ok, Socket} = {error, Reason}

- In Erlang “assignment” is actually matching, so this assignment results in a badmatch exception
- The exception causes process death

Is This Good Code?

- Networks can fail
- Remote hosts can fail
- Remote server apps can fail
- So, gen_tcp:connect must be expected to fail sometimes

Crash or Not?

- If the process
  - must connect now
  - must connect to a particular server instance
  - can’t operate at all without the connection
- Then maybe it’s OK to crash

Crash or Not?

- If the process
  - can defer the connection
  - can try to connect to a different server instance
  - can still offer other capabilities that don't depend on the connection
- Then no, maybe it shouldn't crash

Handle It Elsewhere?

- If we choose to crash when we can't connect, then
  - who will deal with the crash?
  - what will they do to handle it?
  - is it worth logging?
  - what if the alternative doesn't work?

Startup Sequence

- Hierarchical sequence
- Application controller starts the app
- App starts supervisor
- Supervisor starts children
- Workers are typically instances of OTP "behaviors"

OTP Behaviors

- Erlang frameworks that support
  - storage of state in a tail-recursive loop
  - handling of system messages for status
  - code upgrades
- e.g., gen_server and gen_fsm are behaviors
- Developers write behavior impls that fulfill certain callbacks
- One such callback is the “init” function called during behavior process startup

Behavior Init Function

```
init([]) ->
    {ok, Sock} = gen_tcp:connect(...),
    {ok, #state{socket = Sock}}.
```

- Call connect
- Store returned socket in our behavior loop state

Problems in App Startup

- If a child process blocks in init, the supervisor, app, and app controller are blocked as well
- gen_tcp:connect can take a long time to timeout on error
- What happens if connect returns {error, Reason} instead?

More Startup Problems

- Exception in init can cause the supervisor to restart the child
- If the exception occurs repeatedly, the supervisor’s max child restart frequency might be exceeded
- This can cause the app or even the whole Erlang node to die

Crash in Init?

- Q: In general, is “Let It Crash” appropriate within the init function?
- A: Would having its supervisor restart it make it work next time around?

Example: Connect in Init

- Connect is slow: blocks app startup
- Connect fails: supervisor restarts it
  - network is down: restart fails
  - remote server is down: restart fails
  - out of file descriptors: restart fails

Connect Failures in Init

- Bottom line: supervisor restarts are unlikely to make connect work
  - at best, useful for quick temporary network or remote server glitches
- Need different failure handling tactics, such as alternative servers, multiple network interfaces

Failure Example

- Example code calls connect for a non-existent server in the child init function
- Attempt to run it from the Erlang shell: application dies

```
1> application:start(example).
{error,{shutdown,{example_app,start,[normal,[]]}}}
=INFO REPORT==== 1-Mar-2011::21:29:27 ===
    application: example
    exited: {shutdown,{example_app,start,[normal,[]]}}
    type: temporary
```

Failure of Permanent App

- When a permanent app dies, the entire Erlang node dies too

```
2> application:start(example, permanent).
=INFO REPORT==== 1-Mar-2011::21:30:00 ===
    application: example
    exited: {shutdown,{example_app,start,[normal,[]]}}
    type: permanent
{error,{shutdown,{example_app,start,[normal,[]]}}}
{"Kernel pid
terminated",application_controller,"{application_start_failure,example,
{shutdown,{example_app,start,[normal,[]]}}}"}
Crash dump was written to: erl_crash.dump
Kernel pid terminated (application_controller)
({application_start_failure,example,{shutdown,{example_app,start,
[normal,[]]}}})
```

Lessons So Far

- Blocking in init bad, crashing worse
  - breaks thru the Simple Core
  - can cause app and node restarts
- Crash only if someone else can *actually* handle it
- Keep init functions fast, simple, nonblocking

Node Restarts

- If the node hangs or dies, the “heart” program can restart it
- On a hang, heart issues kill -9, so reason for hang is lost
- Heart has no max restart count
  - but see http://steve.vinoski.net/blog/2009/02/22/controlling-erlangs-heart/

Production Logging

- Crashes should be logged to see if any code corrections are necessary
- Production systems typically enable Erlang's System Application Support Libraries (SASL)
- SASL creates better crash messages, allowing for easier triage

SASL Messages

- SASL message for our connect failure: 38 lines (I don't expect you to be able to read it)
- Shows crash reason and stack trace
- Shows state of process when it died

```
{error,{shutdown,{example_app,start,[normal,[]]}}}
=CRASH REPORT==== 3-Mar-2011::18:01:24 ===
  crasher:
    initial call: example:init/1
    pid: <0.55.0>
    registered_name: []
    exception exit: {{badmatch,{error,etimedout}},
                    [{example,init,1},
                     {gen_server,init_it,6},
                     {proc_lib,init_p_do_apply,3}]}
      in function  gen_server:init_it/6
    ancestors: [example_sup,<0.53.0>]
    messages: []
    links: [<0.54.0>]
    dictionary: []
    trap_exit: false
    status: running
    heap_size: 233
    stack_size: 24
    reductions: 282
  neighbours:
=SUPERVISOR REPORT==== 3-Mar-2011::18:01:24 ===
     Supervisor: {local,example_sup}
     Context:    start_error
     Reason:     {{badmatch,{error,etimedout}},
                  [{example,init,1},
                   {gen_server,init_it,6},
                   {proc_lib,init_p_do_apply,3}]}
     Offender:   [{pid,undefined},
                  {name,example},
                  {mfargs,{example,start_link,[]}},
                  {restart_type,permanent},
                  {shutdown,5000},
                  {child_type,worker}]
=INFO REPORT==== 3-Mar-2011::18:01:24 ===
    application: example
    exited: {shutdown,{example_app,start,[normal,[]]}}
    type: temporary
```

Downsides of SASL

- Your QA department will report each crash as a serious bug
  - won't know a "bug crash" from a "crash by design" (unless they know Erlang)
- SASL can use huge amount of memory
  - use Basho's riak_err instead

Lots More To Cover, But Not Today

- ets and tcp_connect are just two simple examples
- Supervision hierarchies, child restart specifications and strategies
- Application start phases
- Process monitoring and the noproc exception
- Debugging common crash causes

Summary

- Erlang’s hierarchical approach of workers, supervisors, and nodes allows each worker process to:
  - handle what it can
  - let someone else handle the rest
- Erlang makes you think hard about production error handling, but also gives you tools to solve the problems

For More Info

- Joe Armstrong: “Programming Erlang”
- Cesarini & Thompson: “Erlang Programming”
- Logan, Merritt, Carlsson: “Erlang and OTP in Action”
- Joe’s thesis: http://www.erlang.org/download/armstrong_thesis_2003.pdf
- erlang-questions mailing list and archives

Thanks