Robust Erlang

John Hughes

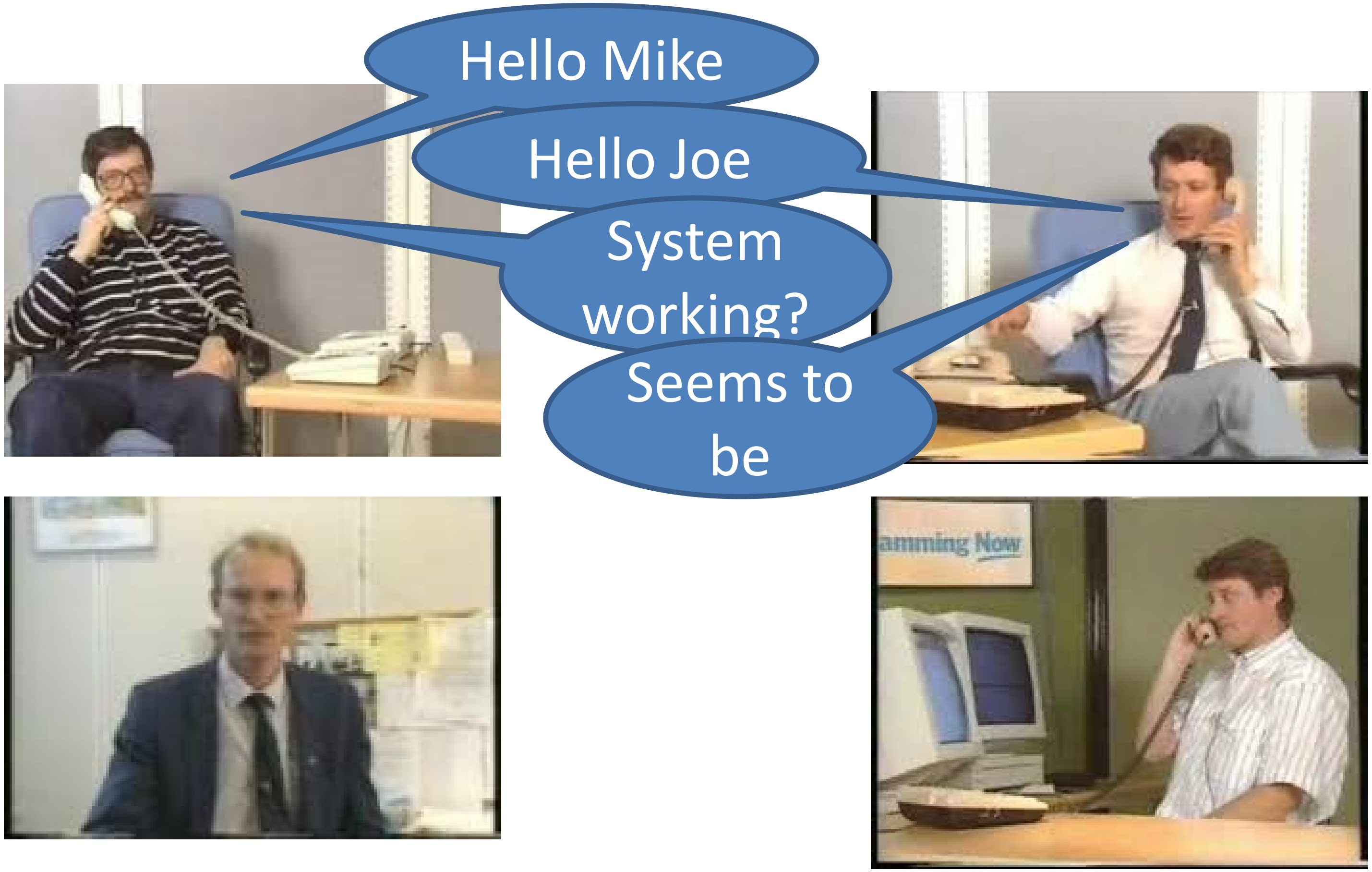
Genesis of Erlang

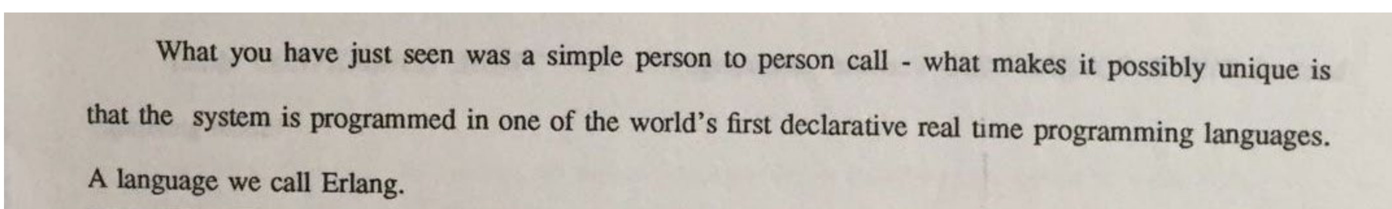
- **Problem:** telephony systems in the late 1980s
  - Digital
  - More and more complex
  - Highly concurrent
  - Hard to get right
  - High reliability needed
- **Approach:** a group at Ericsson research programmed POTS in different languages
- **Solution:** nicest was *functional programming*—but not concurrent
- Erlang designed in the early 1990s

”Plain Old Telephony System”

https://youtu.be/BXmOlCy0oBM

Erlang: The Movie





What you have just seen was a simple person to person call - what makes it possibly unique is that the system is programmed in one of the world's first declarative real time programming languages. A language we call Erlang.

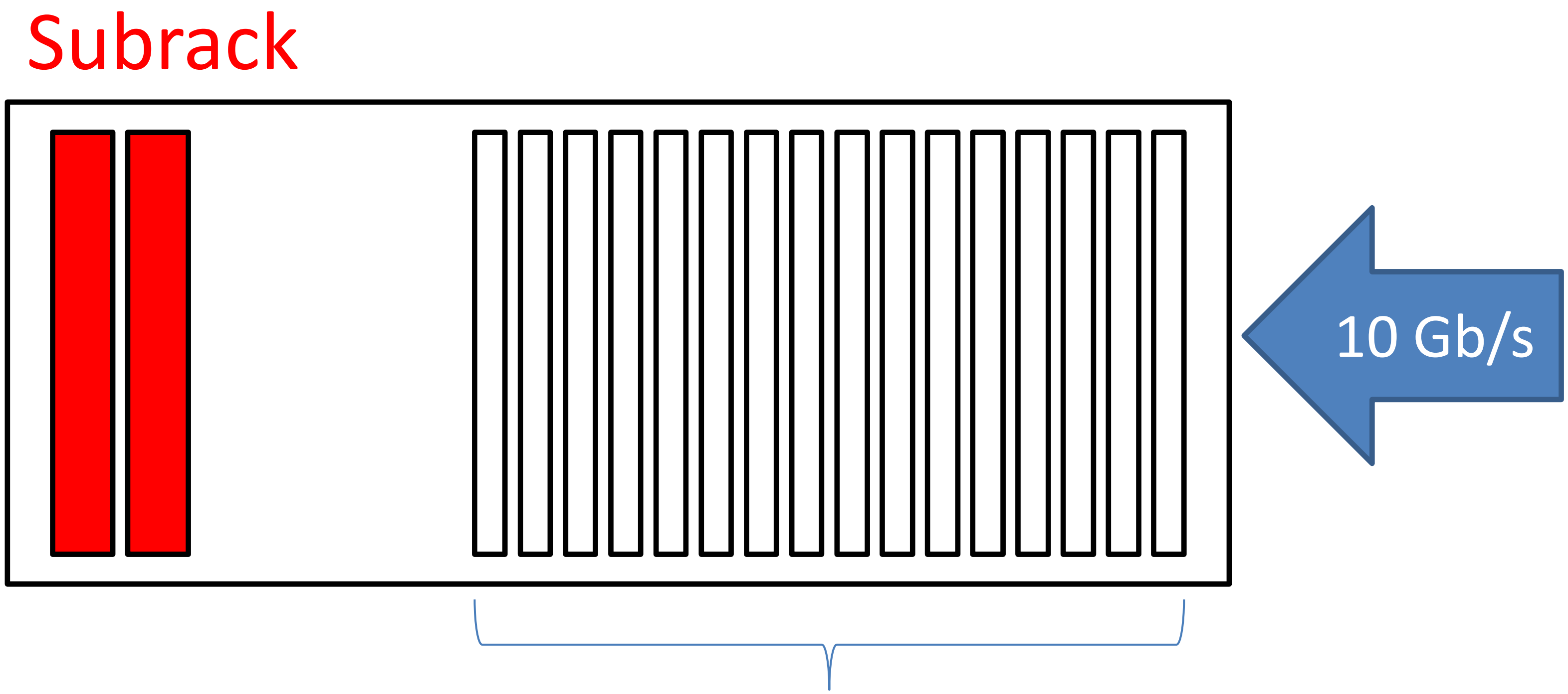
Mid 1990s: the AXD 301

- ATM switch (telephone backbone), released in 1998
- First *big* Erlang project (over a million LOC)
- Born out of the ashes of a disaster!

“The AXE-N venture was to be the most expensive industrial project in Sweden after Saab’s JAS fighter. One calculation estimates that it cost Ericsson SEK 10 billion. The project has often been described as a total failure.”

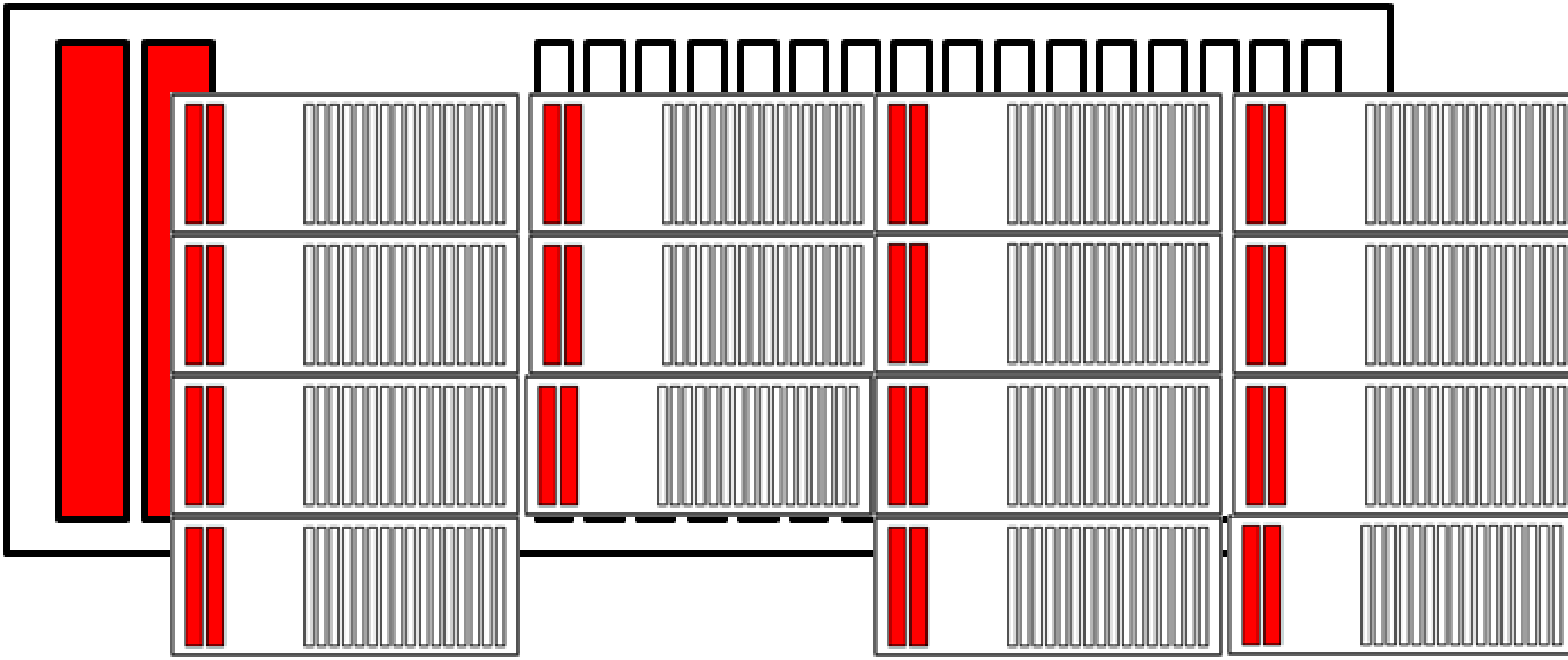
https://www.ericsson.com/en/about-us/history/changing-the-world/big-bang/axe-n

AXD301 Architecture



1,5 million LOC
of Erlang

16 data boards
2 million lines of C++



- 160 Gbits/sec (240,000 simultaneous calls!)
- 32 distributed Erlang nodes
- Parallelism vital from the word go

Typical Applications Today

Invoicing services for web shops—European market leader, in 14 countries

Distributed no-SQL database serving e.g. Denmark and the UK’s medicine card data

Messaging services. See http://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/

What do they all have in common?

- Serving *huge* numbers of clients through parallelism

- Very high demands on *quality of service:* these systems should work *all* of the time

AXD 301 Quality of Service

- 7 nines reliability!
  - Up 99,99999% of the time
- Despite
  - Bugs
    - (10 bugs per 1000 lines is *good*)
  - Hardware failures
    - Always something failing in a big cluster
    - Avoid *any* SPOF

Example: Area of a Shape

```
area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y.
```

```
8> test:area({rectangle,3,4}).
12
9> test:area({circle,2}).
** exception error: no function clause matching
test:area({circle,2}) (test.erl, line 16)
10>
```

What do we do about it?

Defensive Programming

Anticipate a possible error

```
area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y;
area(_) -> 0.
```

Return a plausible result.

```
11> test:area({rectangle,3,4}).
12
12> test:area({circle,2}).
0
```

No crash any more!

Plausible Scenario

- We write lots more code manipulating shapes
- We add circles as a possible shape
  - But we forget to change area!

<LOTS OF TIME PASSES>

- We notice something doesn't work for circles
  - We silently substituted the wrong answer
- We write a special case *elsewhere* to "work around" the bug

Handling Error Cases

- Handling errors often accounts for > ⅔ of a system's code
  - Expensive to construct and maintain
  - Likely to contain > ⅔ of a system's bugs
- Error handling code is often poorly tested
  - Code coverage is usually << 100%

```
ghc: panic! (the 'impossible' happened)
  (GHC version 8.0.1 for x86_64-unknown-linux):
        Prelude.chr: bad argument: 2600468483

Please report this as a GHC bug:  http://www.haskell.org/ghc/reportabug
```

Don't Handle Errors!

LET IT CRASH!

Stopping a malfunctioning program

…is better than …

Letting it continue and wreak untold damage

Let it crash… locally

- **Isolate** a failure within one process!
  - No shared memory between processes
  - No mutable data
  - One process cannot cause another to fail

- *One* client may experience a failure… but the rest of the system keeps going

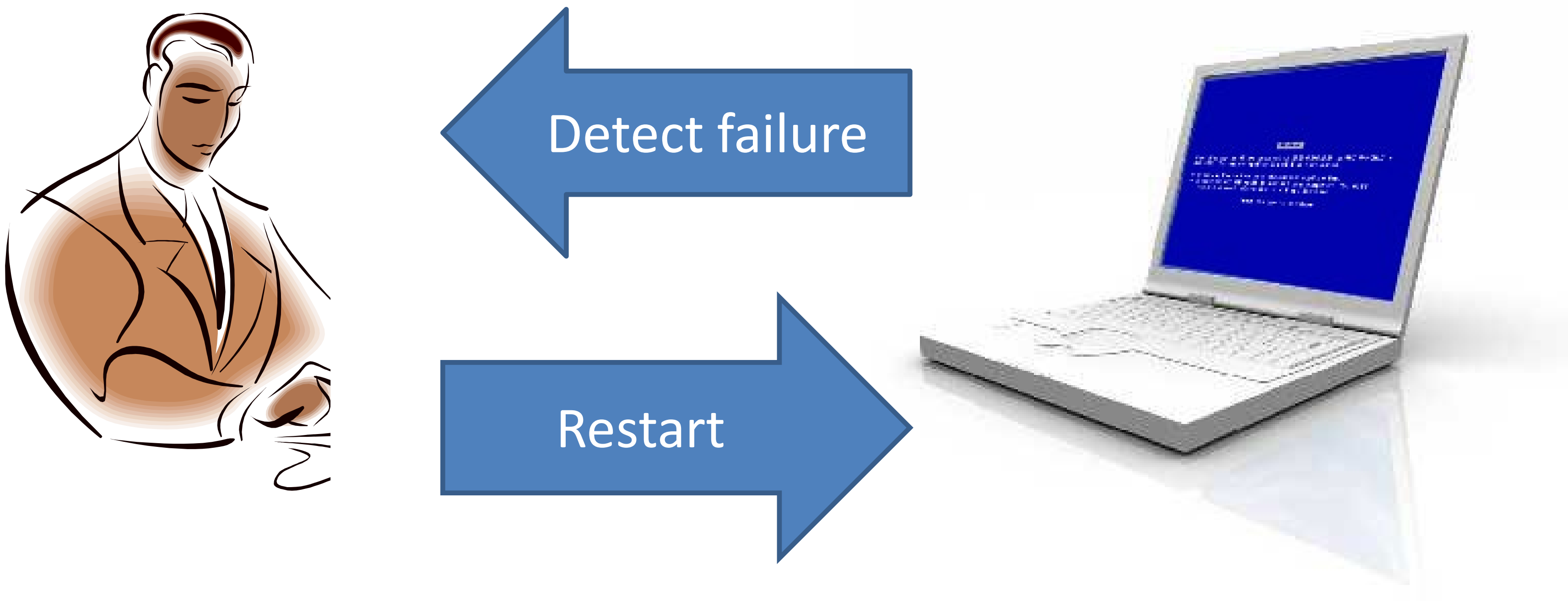
Windows

A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) + 00010E36. The current application will be terminated.

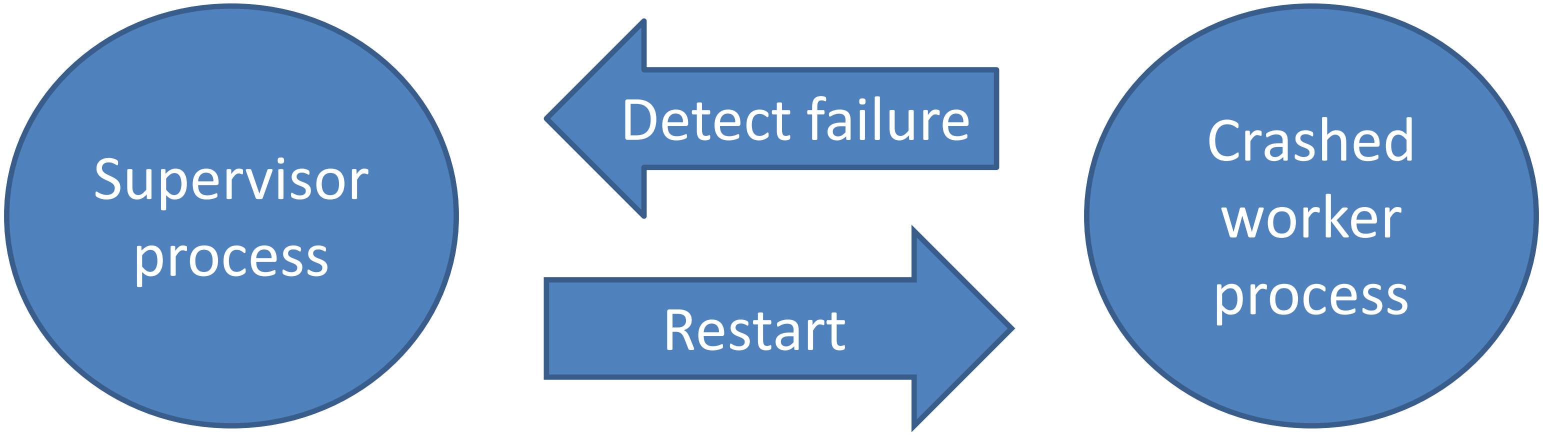
* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue

We know what to do…

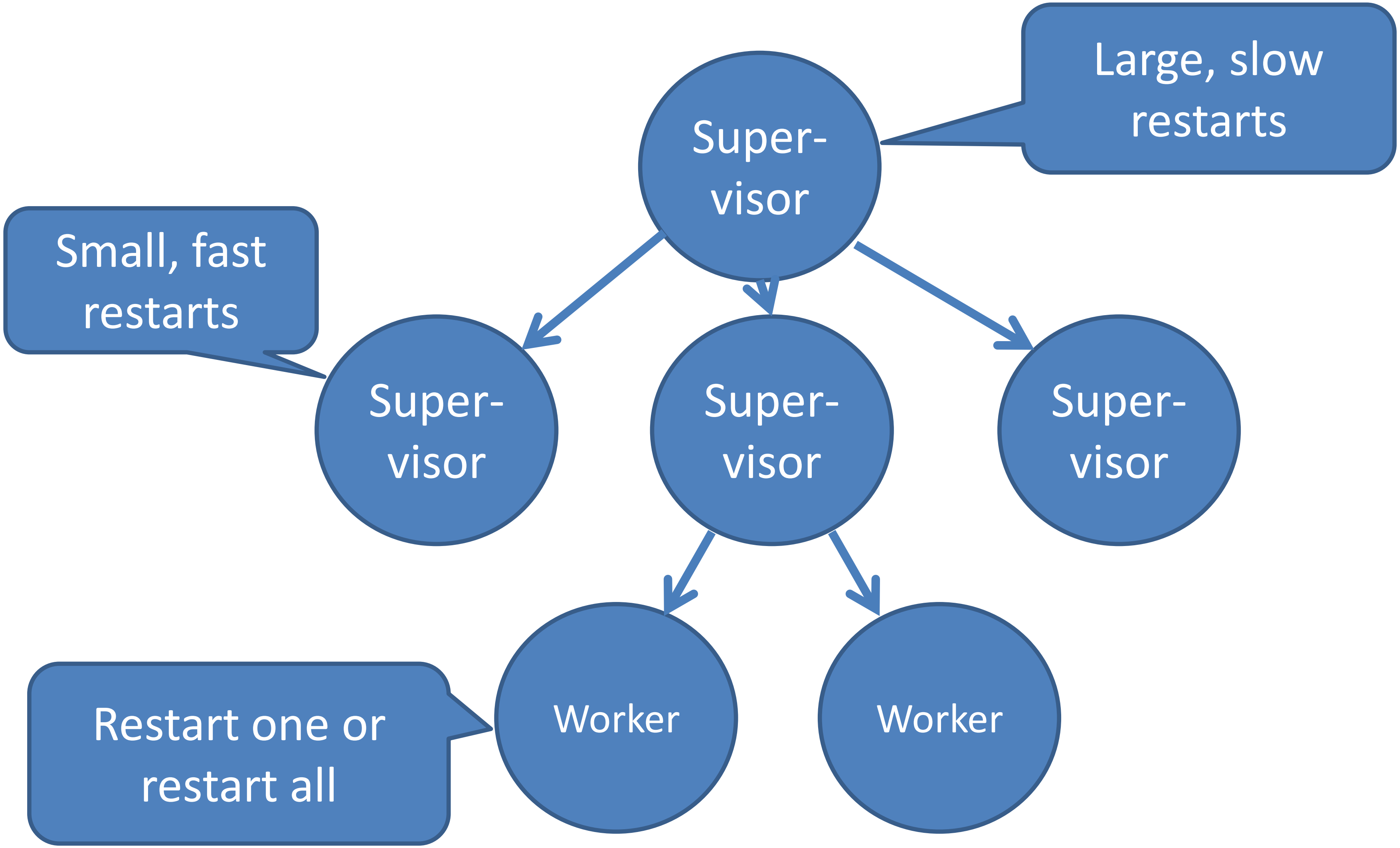


Using Supervisor Processes

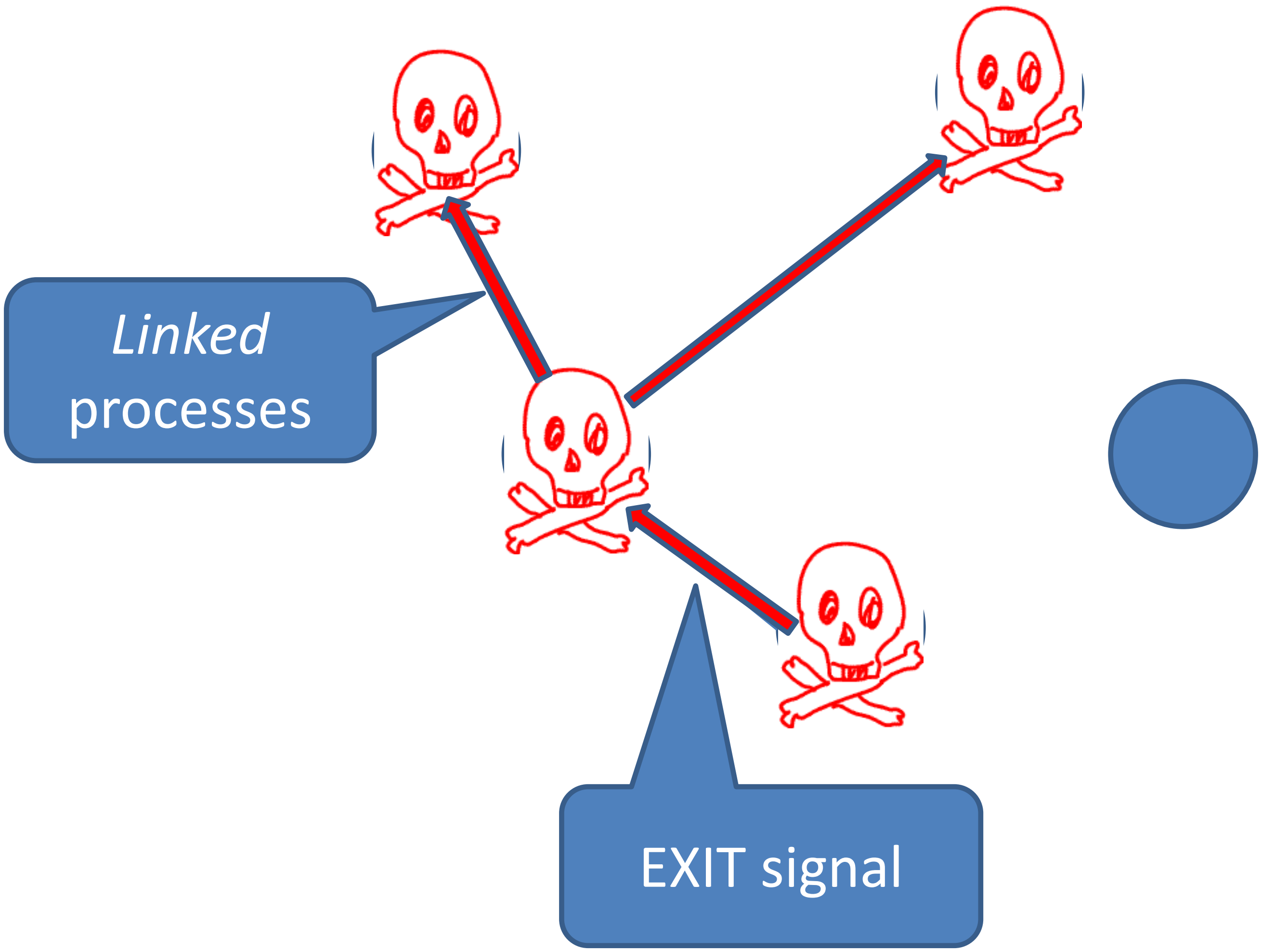


- Supervisor process is *not* corrupted
  - One process *cannot* corrupt another
- Large grain error handling
  - simpler, smaller code

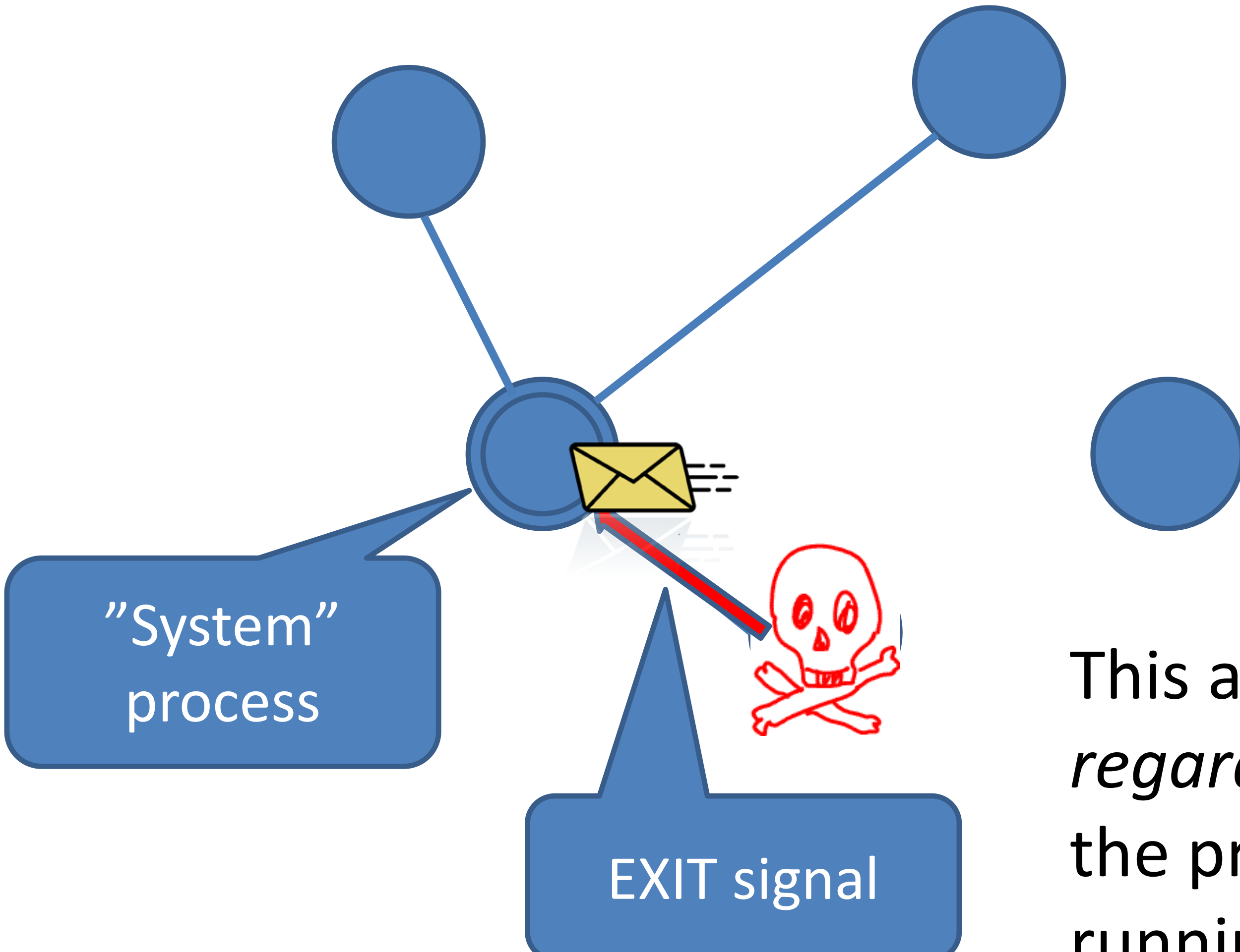
Supervision Trees



Detecting Failures: Links



Linked Processes



This all works *regardless* of where the processes are running☺

Creating a Link

- link(Pid)
  - Create a link between self() and Pid
  - When one process exits, an *exit signal* is sent to the other
  - Carries an *exit reason* (`normal` for successful termination)

- unlink(Pid)
  - Remove a link between self() and Pid

Two ways to spawn a process

- spawn(F)
  - Start a new process, which calls F().

- spawn_link(F)
  - Spawn a new process *and link to it atomically*

Trapping Exits

- An exit signal causes the recipient to exit also
  - Unless the reason is `normal`
- …unless the recipient is a *system process*
  - Creates a message in the mailbox: `{'EXIT',Pid,Reason}`
  - Call `process_flag(trap_exit,true)` to become a system process

An On-Exit Handler

- Specify a function to be called when a process terminates

```
on_exit(Pid,Fun) ->
    spawn(fun() -> process_flag(trap_exit,true),
                   link(Pid),
                   receive
                       {'EXIT',Pid,Why} -> Fun(Why)
                   end
          end).
```

Testing on_exit

```
5> Pid = spawn(fun()->receive N -> 1/N end end).
<0.55.0>
6> test:on_exit(Pid,fun(Why)->
           io:format("***exit: ~p\n",[Why]) end).
<0.57.0>
7> Pid ! 1.
***exit: normal
1
8> Pid2 = spawn(fun()->receive N -> 1/N end end).
<0.60.0>
9> test:on_exit(Pid2,fun(Why)->
         io:format("***exit: ~p\n",[Why]) end).
<0.62.0>
10> Pid2 ! 0.
=ERROR REPORT==== 25-Apr-2012::19:57:07 ===
Error in process <0.60.0> with exit value:
{badarith,[{erlang,'/',[1,0],[]}]}
***exit: {badarith,[{erlang,'/',[1,0],[]}]}
0
```

A Simple Supervi

Real supervisors won’t restart too often—pass the failure up the hierarchy

- Keep a server alive at all times
  - Restart it whenever it terminates

```
keep_alive(Fun) ->
    Pid = spawn(Fun),
    on_exit(Pid,fun(_) -> keep_alive(Fun) end).
```

- Just one problem…

How will anyone ever communicate with Pid?

The Process Registry

- Associate *names* (atoms) with pids
- Enable other processes to find pids of servers, using
  - register(Name,Pid)
    - Enter a process in the registry
  - unregister(Name)
    - Remove a process from the registry
  - whereis(Name)
    - Look up a process in the registry

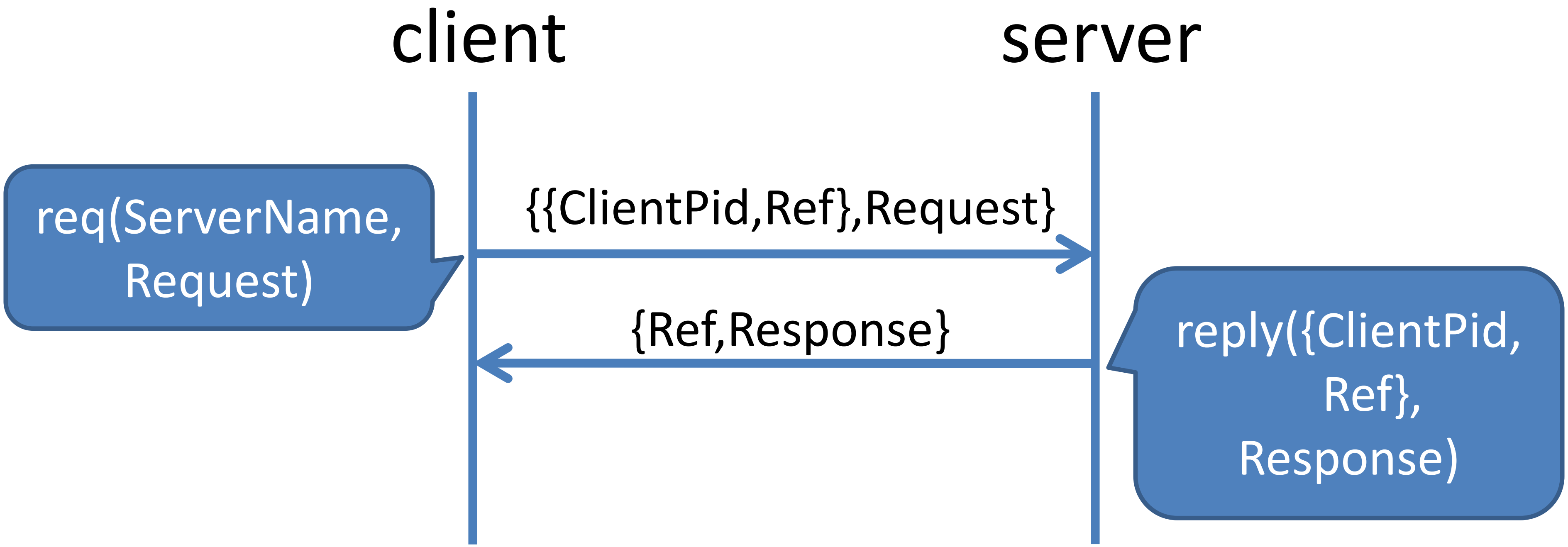
A Supervised Divider

```
divider() ->
   keep_alive(fun() -> register(divider,self()),
                       receive
                             N ->  io:format(”\n~p\n",[1/N])
                       end
              end).
```

```
4> divider ! 0.
=ERROR REPORT==== 25-Apr-2012::20:05:20 ===
Error in process <0.43.0> with exit value:
{badarith,[{test,'-divider/0-fun-0-',0,
          [{file,"test.erl"},{line,34}]}]}
0
5> divider ! 3.
0.3333333333333333
3
```

Supervisors supervise servers

- At the leaves of a supervision tree are processes that service requests
- Let’s decide on a protocol



req/reply

```
req(ServerName,Request) ->
    Ref = make_ref(),
    ServerName ! {{self(),Ref},Request},
    receive
        {Ref,Response} ->
            Response
    end.

reply({ClientPid,Ref},Response) ->
    ClientPid ! {Ref,Response}.
```

Example Server

```
account(Name,Balance) ->
   receive
       {Client,Msg} ->
              case Msg of
                  {deposit,N} ->
                         reply(Client,ok),
                         account(Name,Balance+N);
                 {withdraw,N} when N=<Balance ->
                         reply(Client,ok),
                         account(Name,Balance-N);
                 {withdraw,N} when N>Balance ->
                         reply(Client,{error,insufficient_funds}),
                         account(Name,Balance)
              end
   end.
```

eply

Change the state

A Generic Server

- Decompose a server into…
  - A *generic* part that handles client—server communication
  - A *specific* part that defines functionality for this particular server
- Generic part: receives requests, sends replies, recurses with new state
- Specific part: *computes* the replies and new state

A Factored Server

```
server(State) ->
  receive {Client,Msg} -> {Reply,NewState} = handle(Msg,State),
                          reply(Client,Reply),
                          server(NewState)
  end.
```

How do we parameterise the server on the callback?

```
handle(Msg,Balance) ->
  case Msg of
      {deposit,N}                          -> {ok, Balance+N};
      {withdraw,N} when N=<Balance -> {ok, Balance-N};
      {withdraw,N} when N>Balance   ->
        {{error,insufficient_funds}, Balance}
  end.
```

Callback Modules

- Remember:

foo:baz(A,B,C)

Call function baz in module foo

Mod:baz(A,B,C)

Call function baz in module Mod (a variable!)

- Passing a module *name* is sufficient to give access to a collection of ”callback” functions

A Generic Server

```
server(Mod,State) ->
    receive {Client,Msg} ->
            {Reply,NewState} = Mod:handle(Msg,State),
            reply(Client,Reply),
            server(Mod,NewState)
    end.
```

```
new_server(Name,Mod) ->
    keep_alive(fun() -> register(Name,self()),
                    server(Mod,Mod:init()) end).
```

The Bank Account Module

```
handle(Msg,Balance) ->
   case Msg of
          {deposit,N}                                    -> {ok, Balance+N};
          {withdraw,N} when N=<Balance -> {ok, Balance-N};
          {withdraw,N} when N>Balance   ->
                                  {{error,insufficient_funds}, Balance}
   end.
init() -> 0.
```

- This is *purely sequential* (and hence easy) code
- This is all the application programmer needs to write

What Happens If…

- The client makes a bad call, and…
- The handle callback crashes?

Is this what we want?

- The *server* crashes
- The *client* waits for ever for a reply

- Let’s make the *client* crash instead

Erlang Exception Handling

catch <expr>

- Evaluates to V, if <expr> evaluates to V
- Evaluates to {’EXIT’,Reason} if expr throws an exception with reason Reason

Generic S

```
server(Mod,State) ->
    receive
        {Pid,Msg} ->
            case catch Mod:hand
                {'EXIT',Reason} ->
                    reply(Name,Pid, {crash,Reason}),
                    server(Mod, State   );
                {Reply,NewState} ->
                    reply(Name,Pid, {ok,Reply}),
                    server(Mod,NewState)
            end
    end.
```

```
req(Name,Msg) ->
    …
    receive
        {Ref,{crash,Reason}} ->
            exit(Reason);
        {Ref,{ok,Reply}} ->
            Reply
    end.
```

What should we put here?

We don't *have* a new state!

Transaction Semantics

- The Mk II server supports *transaction semantics*
  - When a request crashes, the *client* crashes…
  - …but the server state is restored to the state before the request

- Other clients are unaffected by the crashes

Hot Code Swapping

- Suppose we want to *change the code* that the server is running
  - It's sufficient to change the *module* that the callbacks are taken from

```
server(Mod,State) ->
    receive
        {Client, {code_change,NewMod}} ->
            reply(Client,{ok,ok}),
            server(NewMod,State);
        {Client,Msg} -> …
    end.
```

The State is not lost

Two Difficult Things Before Breakfast

- Implementing transactional semantics in a server
- Implementing dynamic code upgrade *without losing the state*

Why was it easy?

- Because all of the state is captured in a single value…
- …and the state is updated by a pure function

gen_server for real

- 6 call-backs
  - init
  - handle_call
  - handle_cast—messages with no reply
  - handle_info—timeouts/unexpected messages
  - terminate
  - code_change
- Tracing and logging, supervision, system messages…
- 70% of the code in real Erlang systems

OTP

- A handful of generic behaviours
  - gen_server
  - gen_fsm/gen_statem—traverses a graph of states
  - gen_event—event handlers
  - supervisor—tracks supervision tree+restart strategies
- And there are other more specialised behaviours…
  - gen_leader—leader election
  - …

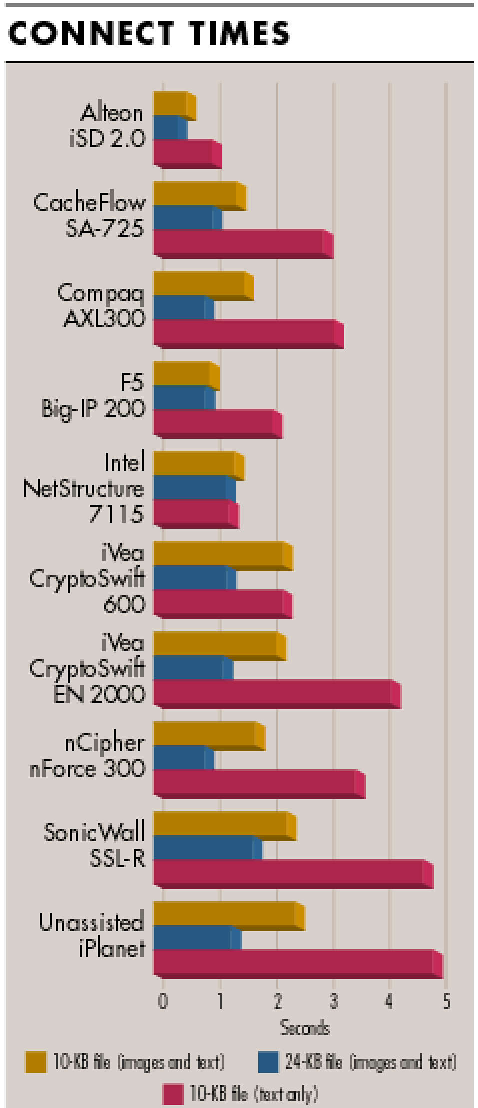
Erlang's Secret

- Highly robust
- Highly scalable
- **Ideal for internet servers**

- 1998: Open Source Erlang (banned in Ericsson)
- First Erlang start-up: Bluetail
  - Bought by Alteon Websystems
    - Bought by Nortel Networks

$140 million in <18 months

SSL Accelerator



- ”Alteon WebSystems' SSL Accelerator offers phenomenal performance, management and scalability.”
  – *Network Computing*

Virding’s Law

- ”Any sufficiently complicated concurrent program in another language contains an ad-hoc informally-specified bug-ridden slow implementation of half of Erlang”.

Erlang Today

- Scales well on multicores
  - 64 cores, no problem!
- Many companies, large and small
  - Amazon/Meta/Nokia/Motorola/HP…
  - Ericsson recruiting Erlangers
  - Many many start-ups
- ”Erlang style concurrency” widely copied
  - Akka in Scala (powers Twitter), Akka.NET, Cloud Haskell…
- Elixir (”Ruby-like syntax”) on the BEAM

Erlang Events

- Code BEAM (Stockholm, San Francisco),
  formerly Erlang User Conference/Erlang Factory
  - (btw: Youtube ”John Hughes Why Functional Programming Matters Erlang Factory 2016”)
- Code MESH (London)
- Code BEAM lite (Amsterdam, Bangalore, Berlin, Budapest, Bologna, Milan, Munich), ErlangCamp (Chicago…)…

Joe Armstrong. 2007. A history of Erlang. In Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III). Association for Computing Machinery, New York, NY, USA, 6-1-6-26

A History of Erlang

Joe Armstrong

Ericsson AB

joe.armstrong@ericsson.com

Abstract

Erlang was designed for writing concurrent programs that "run forever." Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language, not the operating system. Erlang has mechanisms to allow programs to change code "on the fly" so that programs can evolve and change as they run. These mechanisms simplify the construction of software for implementing non-stop systems.

This paper describes the history of Erlang. Material for the paper comes from a number of different sources. These include personal recollections, discussions with colleagues, old newspaper articles and scanned copies of Erlang manuals, photos and computer listings and articles posted to Usenet mailing lists.

1. A History of Erlang

1.1 Introduction

Erlang was designed for writing concurrent programs that "run forever." Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight

operations occur. Telephony software must also operate in the "soft real-time" domain, with stringent timing requirements for some operations, but with a more relaxed view of timing for other classes of operation.

When Erlang started in 1986, requirements for virtually zero down-time and for in-service upgrade were limited to rather small and obscure problem domains. The rise in popularity of the Internet and the need for non-interrupted availability of services has extended the class of problems that Erlang can solve. For example, building a non-stop web server, with dynamic code upgrade, handling millions of requests per day is very similar to building the software to control a telephone exchange. So similar, that Erlang and its environment provide a very attractive set of tools and libraries for building non-stop interactive distributed services.

From the start, Erlang was designed as a practical tool for getting the job done—this job being to program basic telephony services on a small telephone exchange in the Lab. Programming this exchange drove the development of the language. Often new features were introduced specifically to solve a particular problem that I encountered when programming the exchange. Language features that were not used were removed. This was such a rapid process that many of the additions and removals from the language were never recorded. Appendix A gives some idea of the rate at which changes were made to the language. Today, things are much

Summary

- Erlang's fault-tolerance mechanisms and design approach reduce complexity of error handling code, help make systems robust
- OTP libraries simplify building robust systems
- Erlang fits internet servers like a glove—as many start-ups have demonstrated
- Erlang's mechanisms have been widely copied
  - See especially Akka, a Scala library based on Erlang