

All things are not equally important

Making reliable
distributed systems
in the presence of
software errors

Joe Armstrong
joe@sics.se

SYSTEMS 1

How do we correct hardware failures?

Replicate the hardware

How do we correct software errors?

Having two identical copies of the software
won't work - both will fail at the same time
and for the same reason

Why does your computer crash?

Which fails more often, hardware or software?

History

- 1986 - POTS Erlang (in Prolog)
- 1987 - ACS/Dunder
- 1988 - Erlang -> Strand (fails)
- 1989 - JAM (Joe's abstract machine)
- 1990 - Erlang syntax changes (70x faster)
- 1991 - Distribution
- 1992 - Mobility Server
- 1993 - Erlang Systems AB
- 1995 - AXE-N collapses. AXD starts
- 1996 - OTP starts
- 1998 - AXD deployed. Erlang Banned. Open Source Erlang.
Bluetail formed
- 1999 - BMR sold
- 2000 - Alteon buys Bluetail. Nortel buys Alteon
- 2002 - UBF. Concurrency Oriented Programming
- 2003 - Making reliable systems

How do we make systems?



Systems are made of black boxes (components)

Black boxes execute concurrently

Black boxes communicate

How the black box works internally is irrelevant

Failures inside one black box should not crash another black box

Plan

Problem Domain

Architecture

Philosophy

System requirements

Language

Models of programming

Programming for fault tolerance

Abstracting non-function behaviour

OTP

Case studies

APIs and protocols

Contracts

Wrappers

Problem domain

Highly concurrent (hundreds of thousands of parallel activities)

Real time

Distributed

High Availability (down times of minutes/year - never down)

Complex software (million of lines of code)

Continuous operation (years)

Continuous evolution

In service upgrade

Architecture

Philosophy
Way of doing things
Construction Guidelines
Programming examples

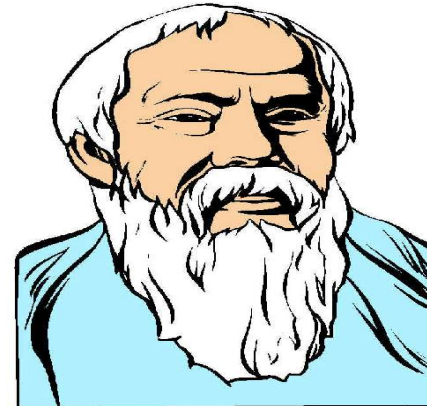
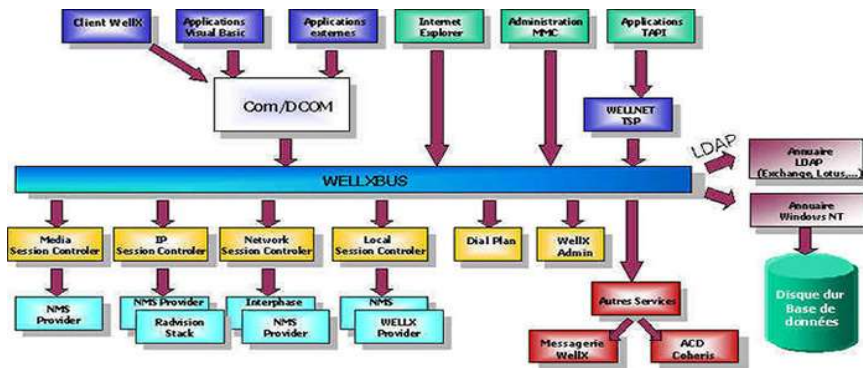
We start with the [bank_client.erl](#)

```
-module(bank_client).  
-export([deposit/2, withdraw/2, balance/1]).  
deposit(Who, X) -> simple_rpc({deposit, Who, X}).  
withdraw(Who, X) -> simple_rpc({withdraw, Who, X}).  
balance(Who) -> simple_rpc(balance, Who).  
  
simple_rpc(X) ->  
  case gen_tcp:connect("localhost", 3010,  
    {ok, Socket} -> [binary, {packet, 4096}] of  
    gen_tcp:send(Socket, [term_to_binary(X)]),  
    wait_reply(Socket);  
    E ->  
      E  
  end.  
  
wait_reply(Socket) ->  
  receive  
    {top, Socket, Bin} ->  
      Term = binary_to_term(Bin),  
      gen_tcp:close(Socket),  
      Term;  
    {tcp_closed, Socket} ->  
      Erase  
  end.
```

This is a simple "no frills" client, that accesses a bank server.

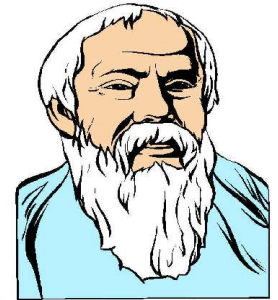
The address of the bank server is "hard wired" into the program at address `localhost` and port `3010`.

Since we are not using distributed Erlang we have to do all encoding and decoding of Erlang terms ourselves. This is achieved by using



Philosophy

Concurrency Oriented Programming



1. COPLs support processes
2. Processes are Isolated
3. Each process has a unique unforgeable Id
4. There is no shared state between processes
5. Message passing is unreliable
6. It should be possible to detect failure in another processes and we should know the reason for failure

System requirements

- | | |
|--------------------------|------------------|
| R1. Concurrency | processes |
| R2. Error encapsulation | isolation |
| R3. Fault detection | what failed |
| R4. Fault identification | why it failed |
| R5. Live code upgrade | evolving systems |
| R6. Stable storage | crash recovery |

Isolation

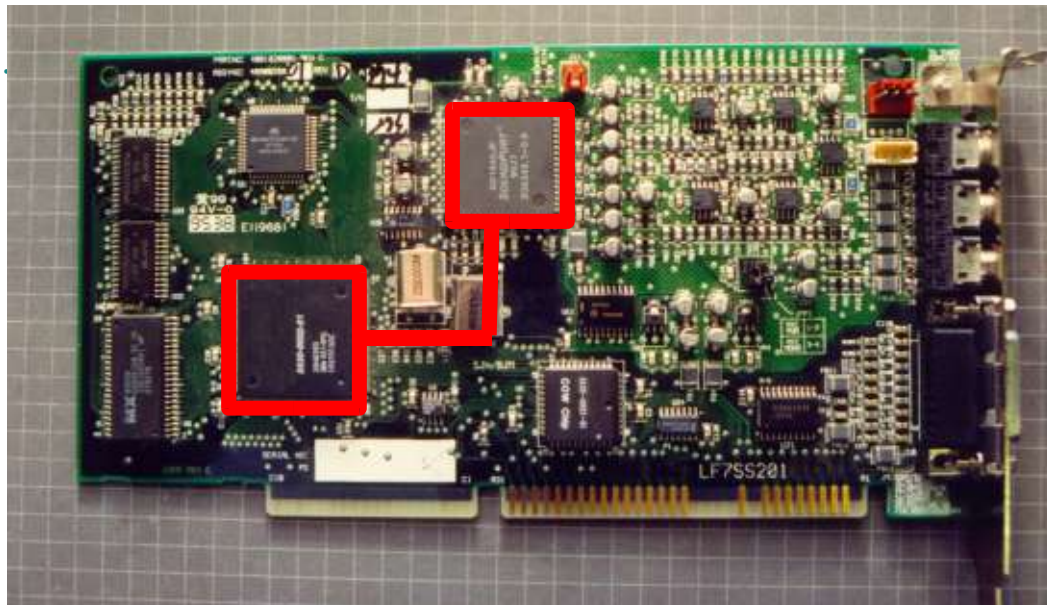
My program should not be able to crash your program.

This is the single most important property that a system component must have

Remember Slide 1?

All things are not equally important

Isolation



Hardware components operate concurrently are isolated and communicate by message passing

Remember slide 5?

Consequences of Isolation

Processes have **share nothing** semantics and data must be copied

Message passing is the only way to exchange data

Message passing is asynchronous

GOOD STUFF

Processes

Copying

Message passing



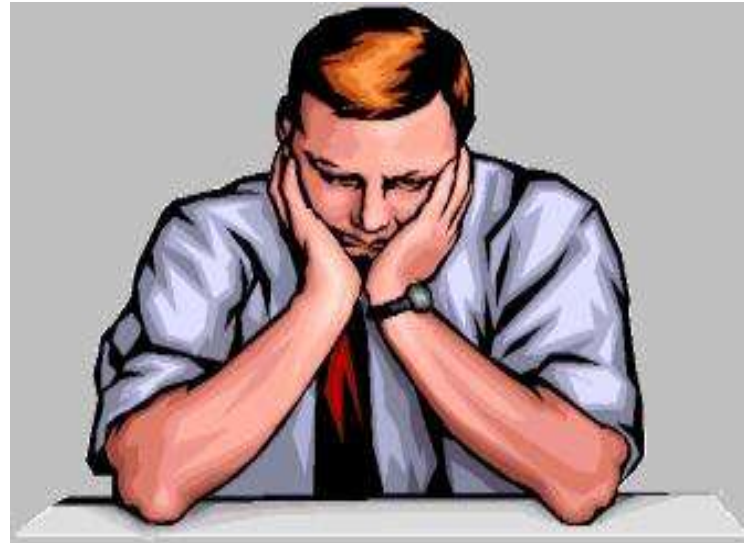
BAD SHIT

Threads

Sharing

Mutexes

Synchronized methods



Computers in Internet are (almost) Isolated

Processes execute on physically separated machines

Processes communicate by message passing

Remember slide 5?

Language

My program should not be able to crash your program

Need strong isolation and concurrency

Processes are OK - threads are not (threads have shared resources)

Can't use OS processes (Heavy - semantics depends on OS)

Java doesn't work

...The only safe way to execute multiple applications, written in the Java programming language, on the same computer is to use a separate JVM for each of them, and to execute each JVM in a separate OS process. This introduces various inefficiencies in resource utilization, which downgrades performance, scalability, and application startup time. The benefits the language can offer are thus reduced mainly to portability and improved programmer productivity. Granted these are important, but the full potential of language-provided safety is not realized. Instead there exists a curious distinction between ``language safety,`` and ``real safety``.

... tasks cannot directly share objects, and that the only way for tasks to communicate is to use standard, copying communication mechanisms

- Czajkowski, and Daynes, Sun Microsystems

Nor does C and C++

No processes (OS has processes but not C or C++)

Terrible isolation pointers etc.

Non-portable (word sizes, big-/little-endian problems)

No GC

What about Erlang?



Lightweight processes (lighter than OS threads)

Good isolation (not perfect yet ...)

Programs never lose control

Error detection primitives

Reason for failure is known

Exceptions

Garbage collected memory

Lots of processes

Functional



Agner Krarup Erlang (1878-1929)

Erlang

You can create a parallel process

```
Pid = spawn(fun() -> ... end).
```

then send it a message

```
Pid ! Msg
```

and then wait for a reply

```
receive
```

```
{Pid, Reply} ->
```

```
    Actions
```

```
end
```

*It typically takes 1 microsecond to
create a process or send a message*

*Processes are
isolated*

Generalisation

Client

```
Pid = spawn(fun() -> loop() end)
Pid ! {self(), 21},
receive
  {Pid, Val} -> ...
end
```

Server

```
loop() ->
  receive
    {From, X} ->
      From ! {self(), 2*X},
      loop()
  end.
```

A simple process

Client

```
Double = fun(X) -> 2 * X end,
Pid = spawn(fun() -> loop(Double) end)
Pid ! {self(), 21},
receive
  {Pid, Val} -> ...
end
```

Server

```
loop(F) ->
  receive
    {From, X} ->
      From ! {self(), F(X)},
      loop(F)
  end.
```

Generalised

A generic server

```
-module(gserver).  
-export([start/1, rpc/2, code_change/2]).
```

```
start(Fun) ->  
    spawn(fun() -> loop(Fun) end).
```

```
rpc(Pid, Q) ->  
    Pid ! {self(), Q},  
    receive  
        {Pid, Reply} ->  
            Reply  
    end.
```

```
code_change(Pid, Fun1) ->  
    Pid ! {swap_code, Fun1}.
```

```
loop(F) ->  
    receive  
        {swap_code, F1} ->  
            loop(F1);  
        {Pid, X} ->  
            Pid ! {self(), F(X)},  
            loop(F);  
    end.
```

```
Double = fun(X) -> 2*X end,  
Pid = gserver:start(Double),  
...  
Triple = fun(X) -> 3*X end,  
gserver:code_change(Pid, Triple)
```

A generic server with data

```
-module(gserver).
-export([start/2, rpc/2, code_change/2]).

start(Fun, Data) ->
    spawn(fun() -> loop(Fun, Data) end).

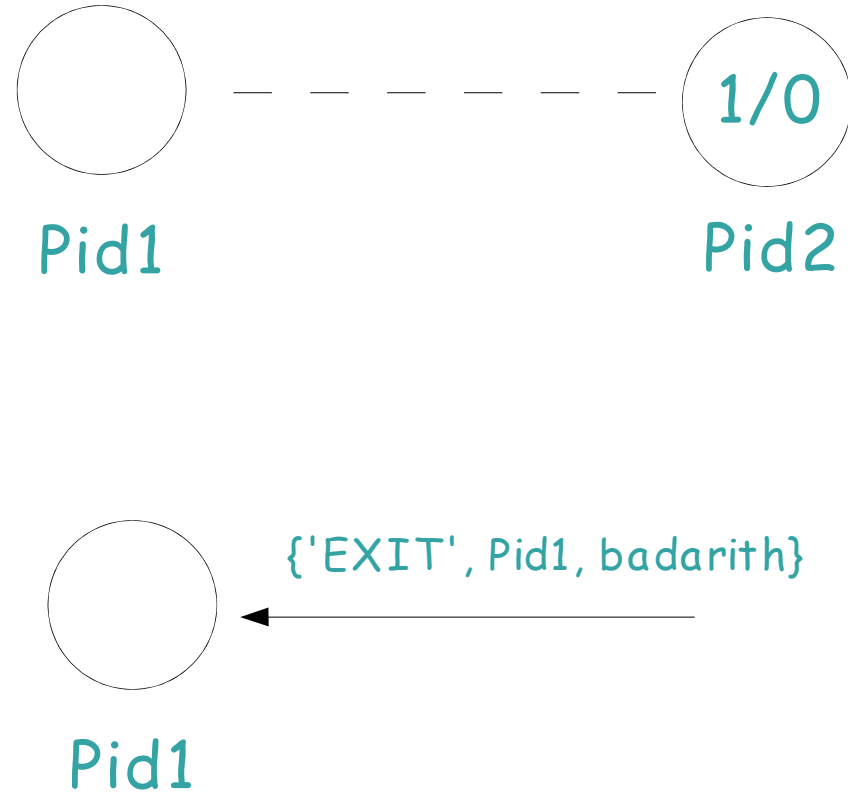
rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} ->
            Reply
    end.

code_change(Pid, Fun1) ->
    Pid ! {swap_code, Fun1}.

loop(F, Data) ->
    receive
        {swap_code, F1} ->
            loop(F1);
        {Pid, X} ->
            {Reply, Data1} = F(X),
            Pid ! {self(), Reply},
            loop(F, Data1);
    end.
```

Trapping errors

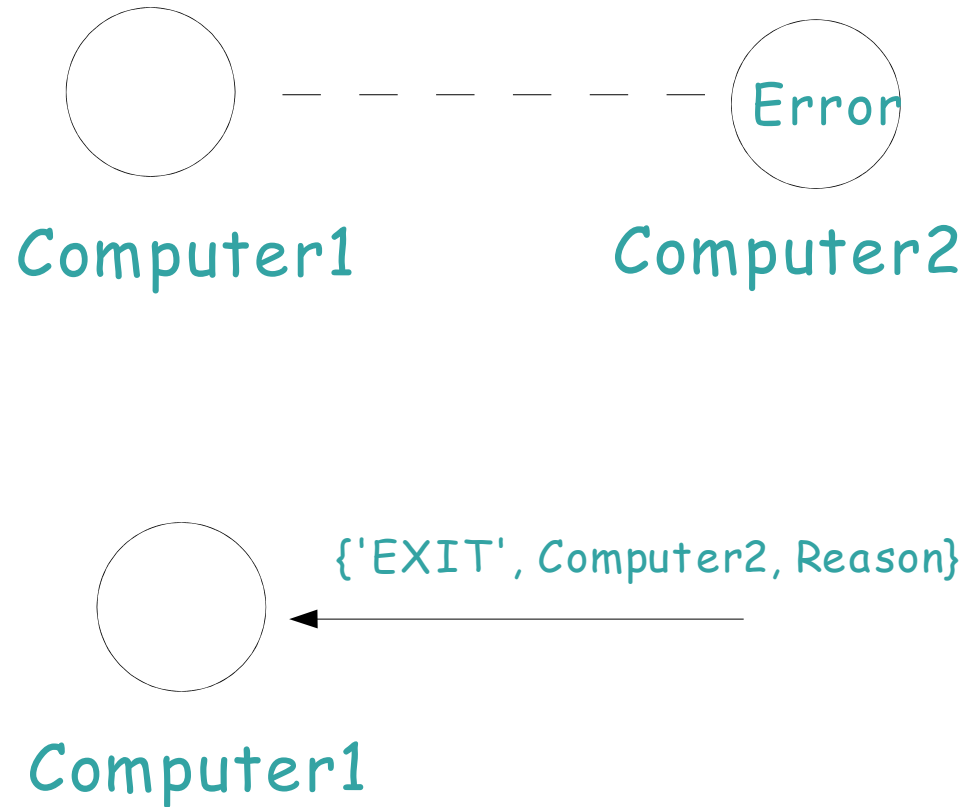
```
In Pid1 ...  
Pid2 = spawn_link(fun() -> ... end).  
process_flag(trap_exit, true)  
...  
receive  
  {'EXIT', Pid, Why} ->  
    Actions  
end.
```



error detection + reason for failure (slide 10)

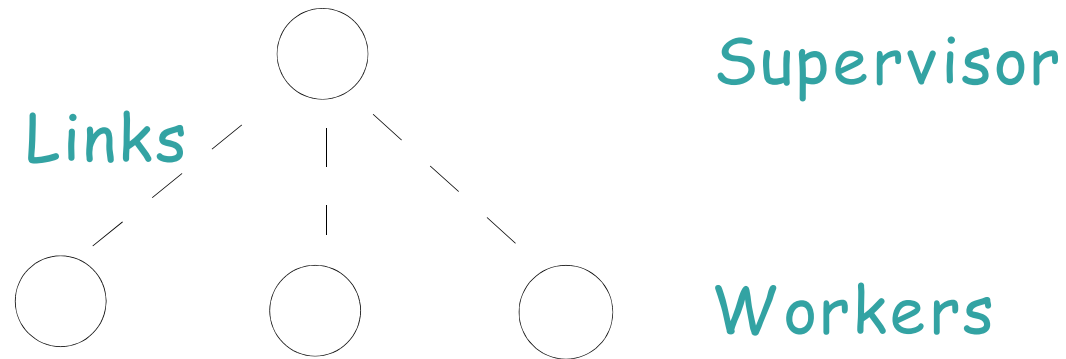
Why remote trapping of errors?

To do fault-tolerant computing you need at least TWO computers



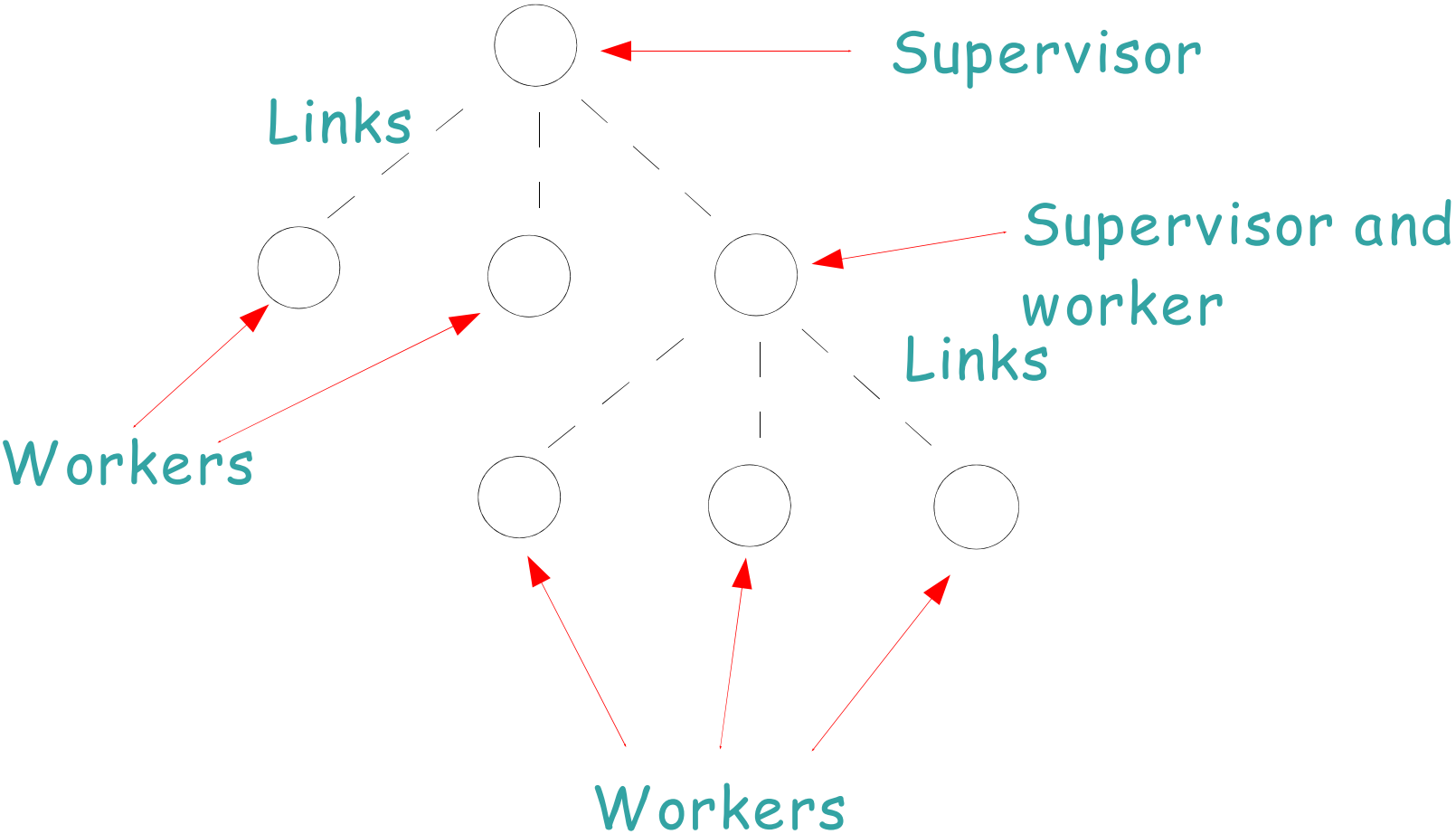
Programming for errors

If you can't do what you want to do try
and do something simpler



The supervisor monitors the
workers and restarts them if
they fail

A supervision hierarchy



OTP behaviours

Generic libraries for building components of a real-time system.

Includes

Client-server (slides 21-23 only much more advanced)

Finite State machine (thesis)

Supervisor (slides 26 and 27)

Event Handler (thesis)

Applications (thesis)

Systems (OTP documentation)

OTP

Set of behaviours

Documentation

Courses

Libraries

Open source Erlang release

Available from <http://www.erlang.org/>

Used in.

Ericsson AXD301 (in Engine)

Alteon (Nortel) SSL accelerator and
VPN

Teba bank South Africa

What's next?

~~How do we program?~~

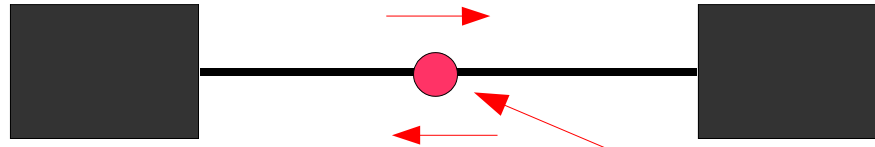
Solved

How to we make systems?

How do we store stuff?

How to we find stuff?

How do we make systems?



Systems are made of black boxes
(components)

Protocol checker

Black boxes execute concurrently

Black boxes communicate with defined
(universal) protocols

The protocol is checked externally

How the black box works internally is
irrelevant

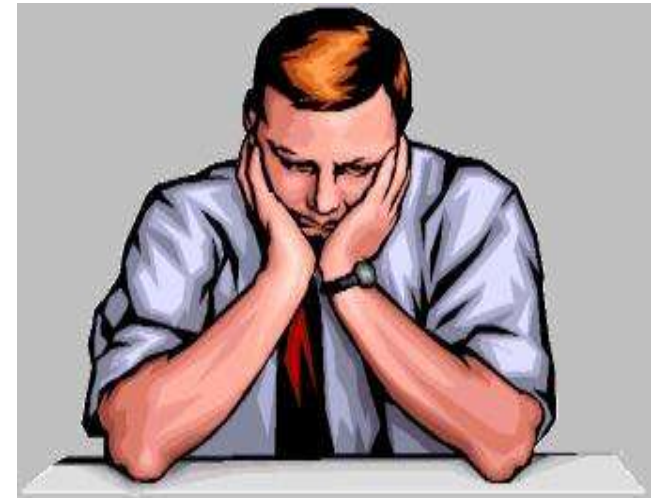
APIs done wrong

```
+type file:open(fileName(), read | write) ->  
  {ok, fileHandle()}  
  | {error, string()}.
```

```
+type file:read_line(fileHandle()) ->  
  {ok, string()} | eof.
```

```
+type file:close(fileHandle()) ->  
  true.
```

```
+deftype fileName() = [int()]  
+deftype string()   = [int()].  
+deftype fileHandle() = pid().
```



```
silly() ->  
{ok, H} = file:open("foo.dat", read),  
file:close(H),  
file:read_line(H).
```

APIs done better

```
+type start x file:open(fileName(), read | write) ->  
  {ok, fileHandle()} x ready  
  | {error, string()} x stop.
```

```
+type ready x file:read_line(fileHandle()) ->  
  {ok, string()} x ready  
  | eof x atEof.
```

```
+type atEof | ready x file:close(fileHandle()) ->  
  true x stop.
```

```
+type atEof | ready x file:rewind(fileHandle()) ->  
  true x ready.
```



```
silly() ->  
  {ok, H} = file:open("foo.dat", read),  
  file:close(H),  
  file:read_line(H).
```

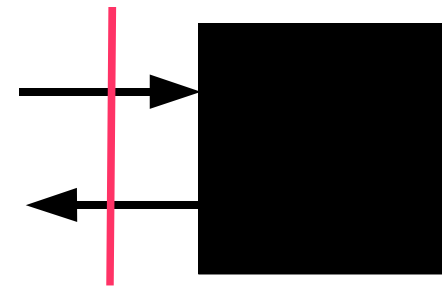
Protocols or APIs

```
+state start x {open, fileName(), read | write} ->  
    {ok, fileHandle()} x ready  
    | {error, string()} x stop.
```

```
+state ready x {read_line, fileHandle()} ->  
    {ok, string()} x ready  
    | eof x atEof.
```

```
+state ready | atEof x {close, fileHandle()} ->  
    true x stop.
```

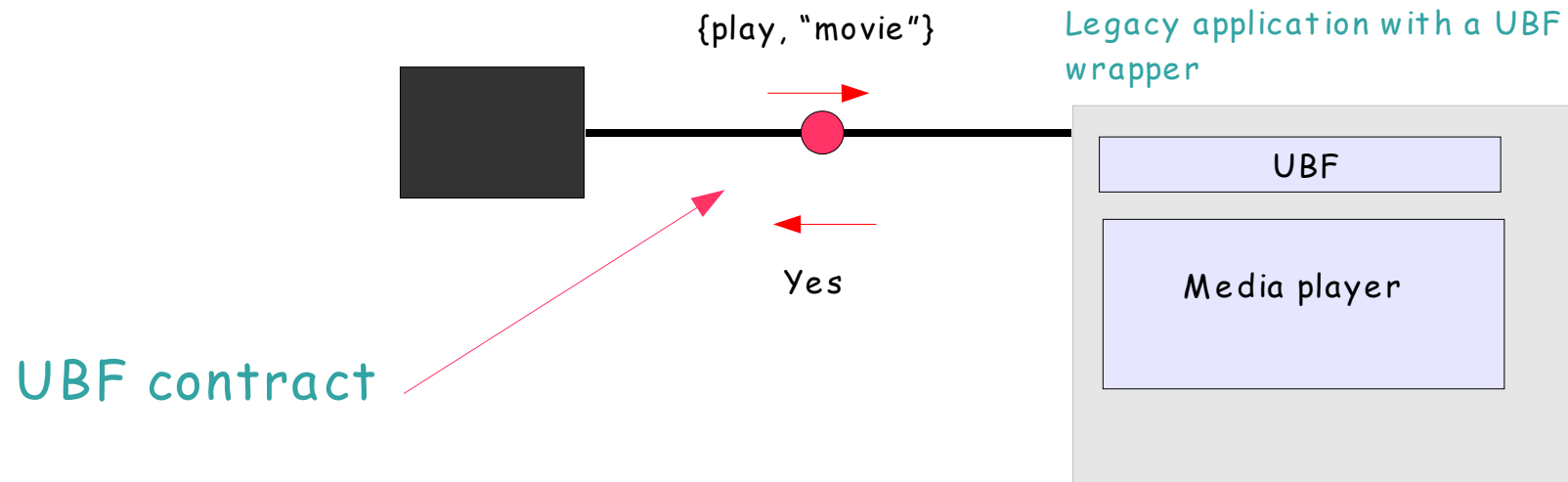
```
+state ready | atEof x {rewind, fileHandle()} ->  
    true x ready
```



How things work inside the black box is irrelevant

Check the protocol at the boundaries to the black box

Putting things together



+state active

```
{play, string()} => yes    x  playing;  
                  |  no    x  active
```

What's next?

~~How do we program?~~

Solved

~~How to we make systems?~~

Solved

How do we store stuff?

How to we find stuff?

What's left to do?

Write loads of wrappers for legacy applications

Implement UBF in all languages known to man

Increase the property of "isolation" in Erlang

Make decent OS's that obey principle of isolation

Add !! to core Erlang

Rewrite kernel libraries

Figure out how to store things

Figure out how to find things

- end -