Fault tolerance 101
Joe Armstrong

Fault

- “behaves as per specification”
- “does not crash”

Many systems have no specification

Programming is the act of turning an inexact description of something (the specification) into an exact description of the thing (the program)

A program is the most precise description of the problem that we have

What is fault tolerance?

- The ability to behave in a sensible manner in the presence of failure. *Consumer software, websites, ...*

- The ability to behave exactly as specified despite failures. *Air traffic control, nuclear power station control.*

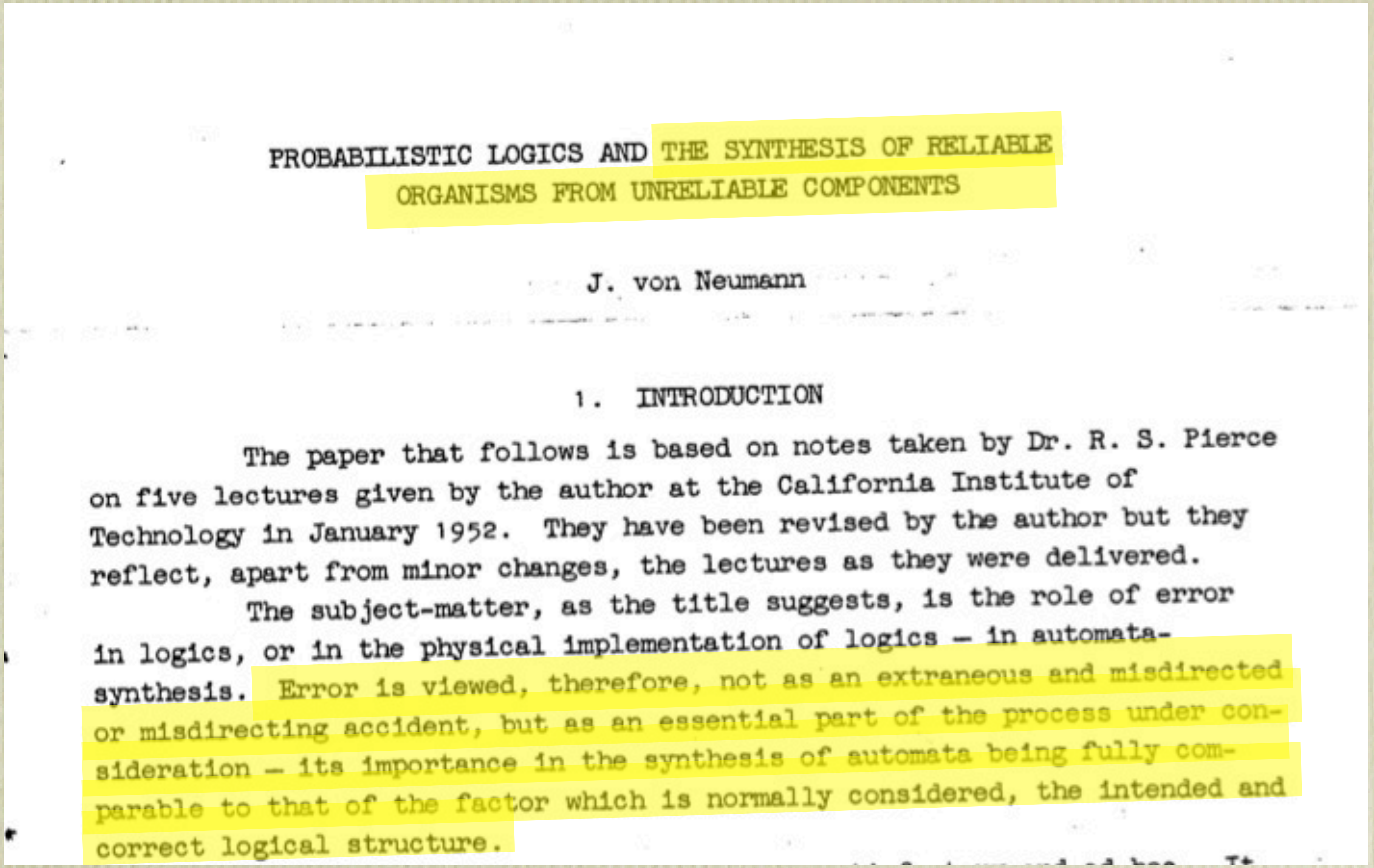
"In a sensible manner" is rather wooly

Exact specification is extremely difficult

When there is no spec - "in a sensible manner" means - does not crash

- History
- Hardware Fault Tolerance
- Software Fault Tolerance
- Specifications and code
- Erlang FT
- Demo

We cannot prevent failures



PROBABILISTIC LOGICS AND THE SYNTHESIS OF RELIABLE ORGANISMS FROM UNRELIABLE COMPONENTS

J. von Neumann

1. INTRODUCTION

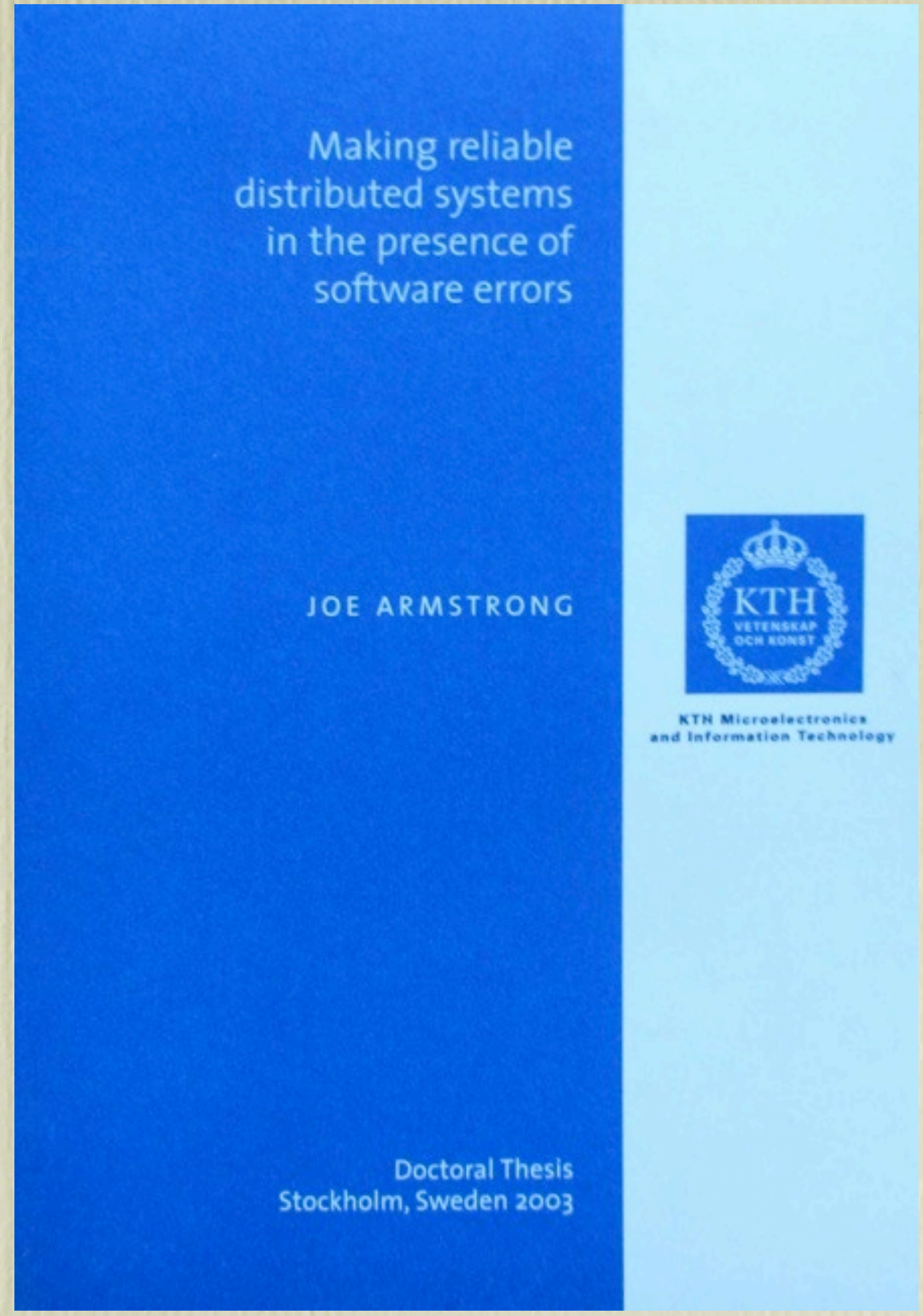
The paper that follows is based on notes taken by Dr. R. S. Pierce on five lectures given by the author at the California Institute of Technology in January 1952. They have been revised by the author but they reflect, apart from minor changes, the lectures as they were delivered.

The subject-matter, as the title suggests, is the role of error in logics, or in the physical implementation of logics — in automata-synthesis. Error is viewed, therefore, not as an extraneous and misdirected or misdirecting accident, but as an essential part of the process under consideration — its importance in the synthesis of automata being fully comparable to that of the factor which is normally considered, the intended and correct logical structure.

Automata Studies
ed. C. Shannon
Princ. Univ. Press 1956

Q: Can we make reliable systems that behave reasonably from unreliable components?

A: Yes



Making reliable distributed systems in the presence of software errors

JOE ARMSTRONG

KTH Microelectronics and Information Technology

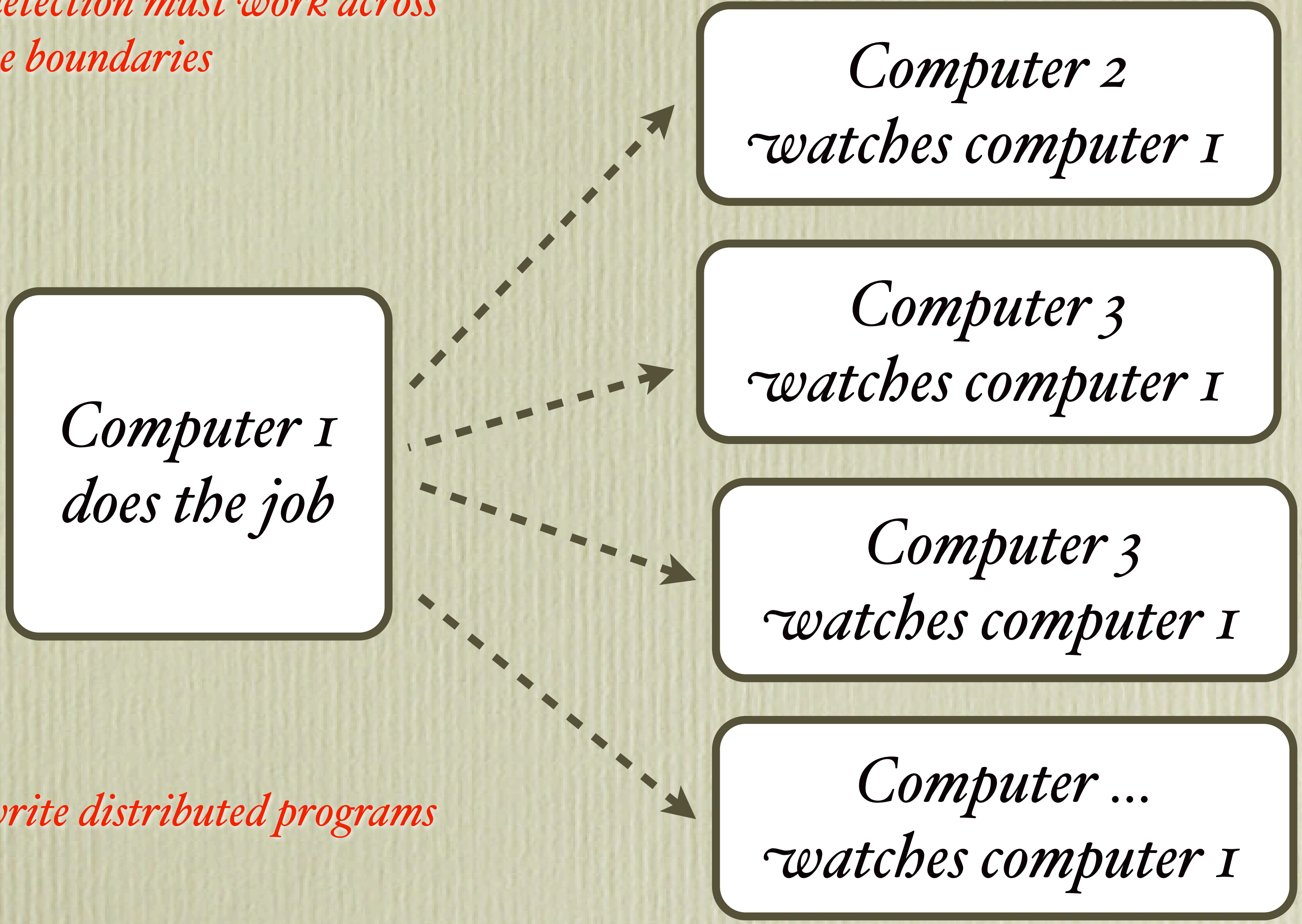
Doctoral Thesis
Stockholm, Sweden 2003

The Cornerstones of FT

- Detect Errors
- Correct Errors
- Stop Errors from Propagating

Needs > 1 computer

Error detection must work across machine boundaries


Computer 1 does the job
Computer 2 watches computer 1
Computer 3 watches computer 1
Computer 3 watches computer 1
Computer … watches computer 1

Must write distributed programs

Programs run in parallel

Decoupling and separation helps stop errors from propagating

Things to ponder

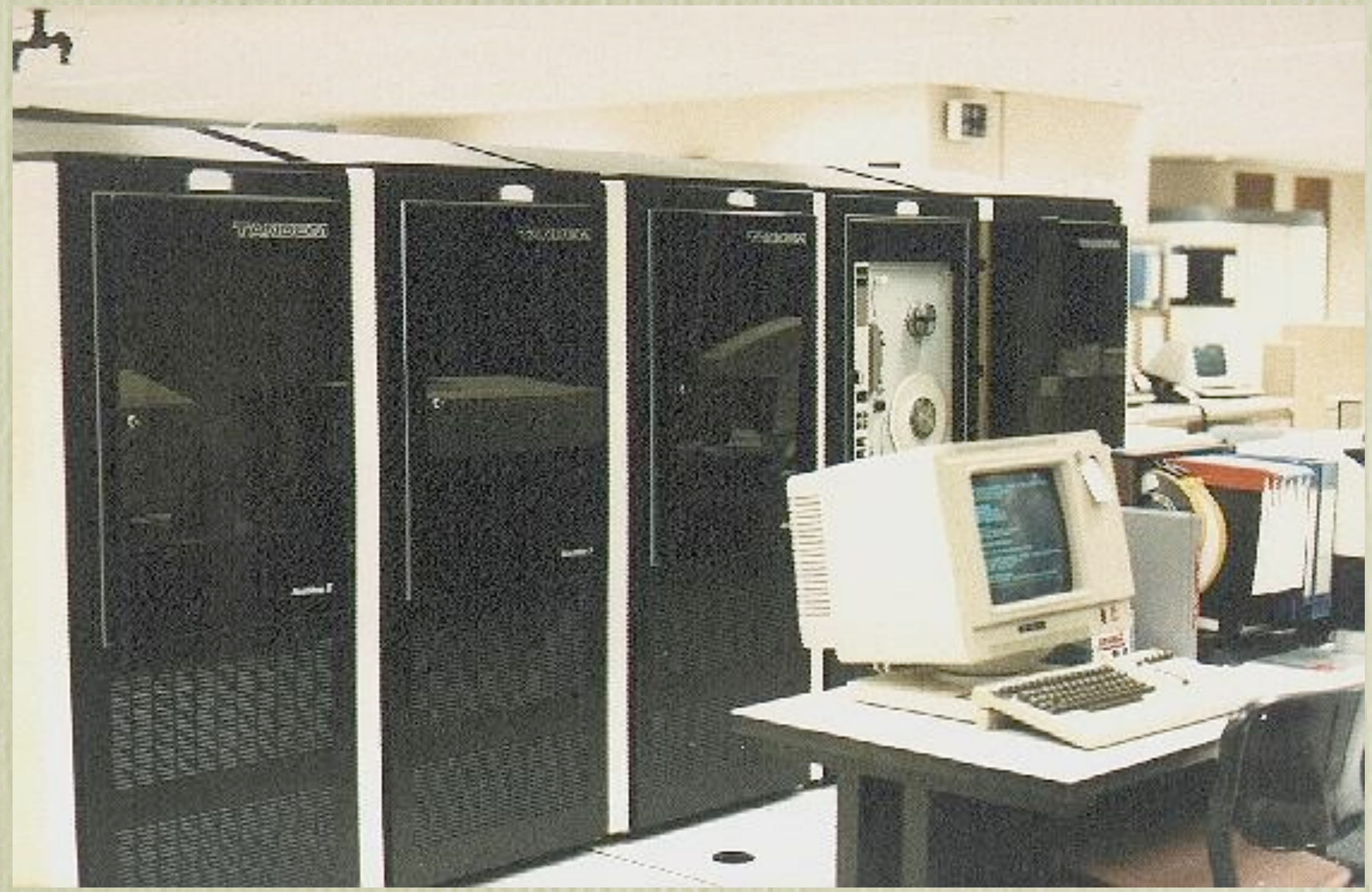
- Hardware can fail
- Software either complies with a spec = works or does not do what the spec says = fails
- What should the software do when the system behaves in a way that is not described in the spec?
- What do we do when we don't have a spec?
- Can we make reliable systems that behave reasonably from unreliable components?
- Detecting or masking errors?
- Correcting errors
- Propagation of errors
- Error firewalls
- Self-repairing zones
- Static/Dynamic error detection

Hardware fault tolerance

- System that mask (hide) errors and use redundancy to mask errors.

  Examples: RAID disks, error correcting bits in memory hardware etc.

Tandem nonstop II (1981)



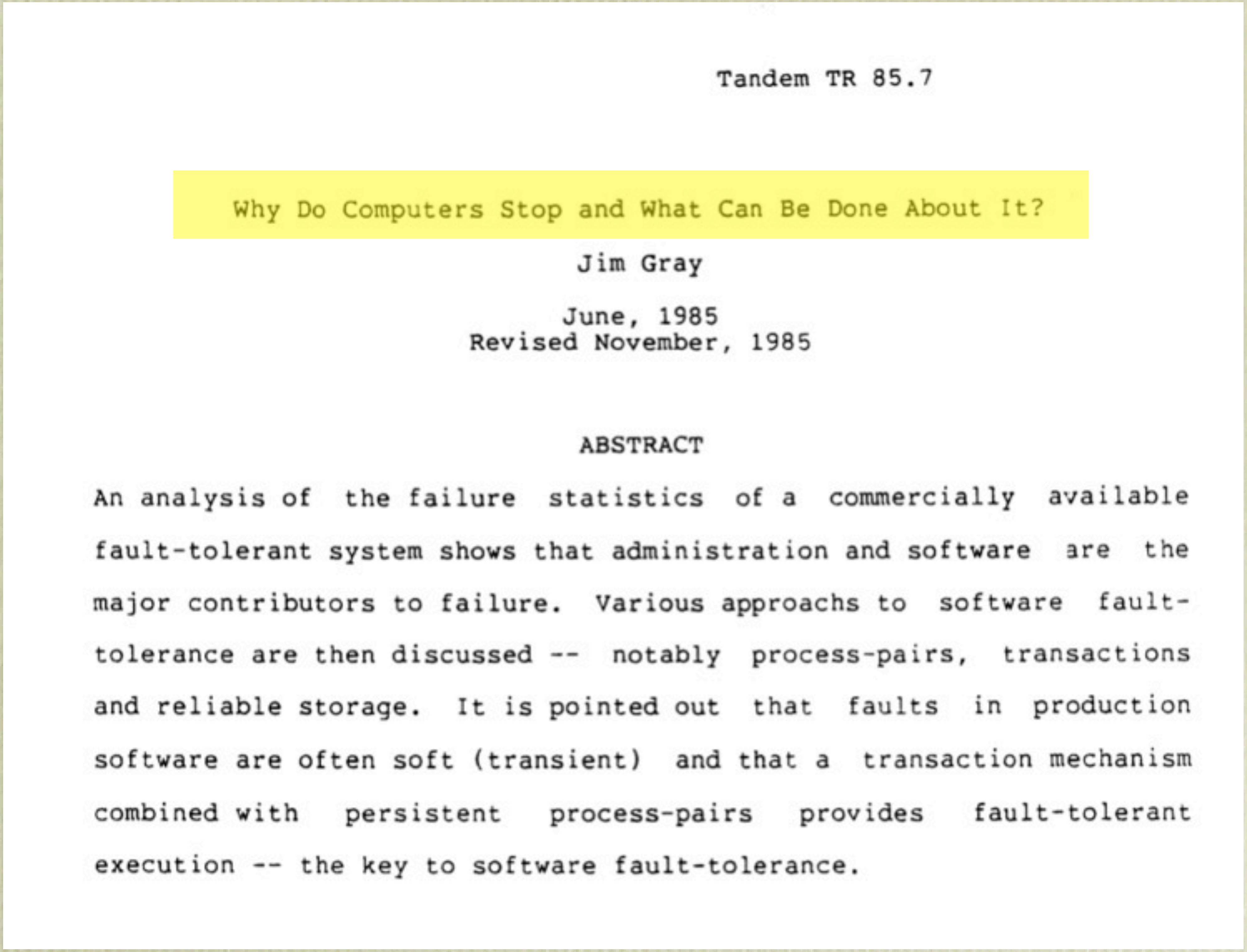
Tandem ...

Tandem Computers, Inc. was the dominant manufacturer of fault-tolerant computer systems for ATM networks,banks, stock exchanges, telephone switching centers, and other similar commercial transaction processing applications requiring maximum uptime and zero data loss.

To contain the scope of failures and of corrupted data, these multi-computer systems have no shared central components, not even main memory. Conventional multi-computer systems all use shared memories and work directly on shared data objects. Instead, NonStop processors cooperate by exchanging messages across a reliable fabric, and software takes periodic snapshots for possible rollback of program memory state.

Besides handling failures well, this "shared-nothing" messaging system design also scales extremely well to the largest commercial workloads. Each doubling of the total number of processors would double system throughput, up to the maximum configuration of 4000 processors. In contrast, the performance of conventional multiprocessor systems is limited by the speed of some shared memory, bus, or switch. Adding more than 4–8 processors that way gives no further system speedup. NonStop systems have more often been bought to meet scaling requirements than for extreme fault tolerance. They compete well against IBM's largest mainframes, despite being built from simpler minicomputer technology.

All quotes from Wikipedia



Tandem TR 85.7

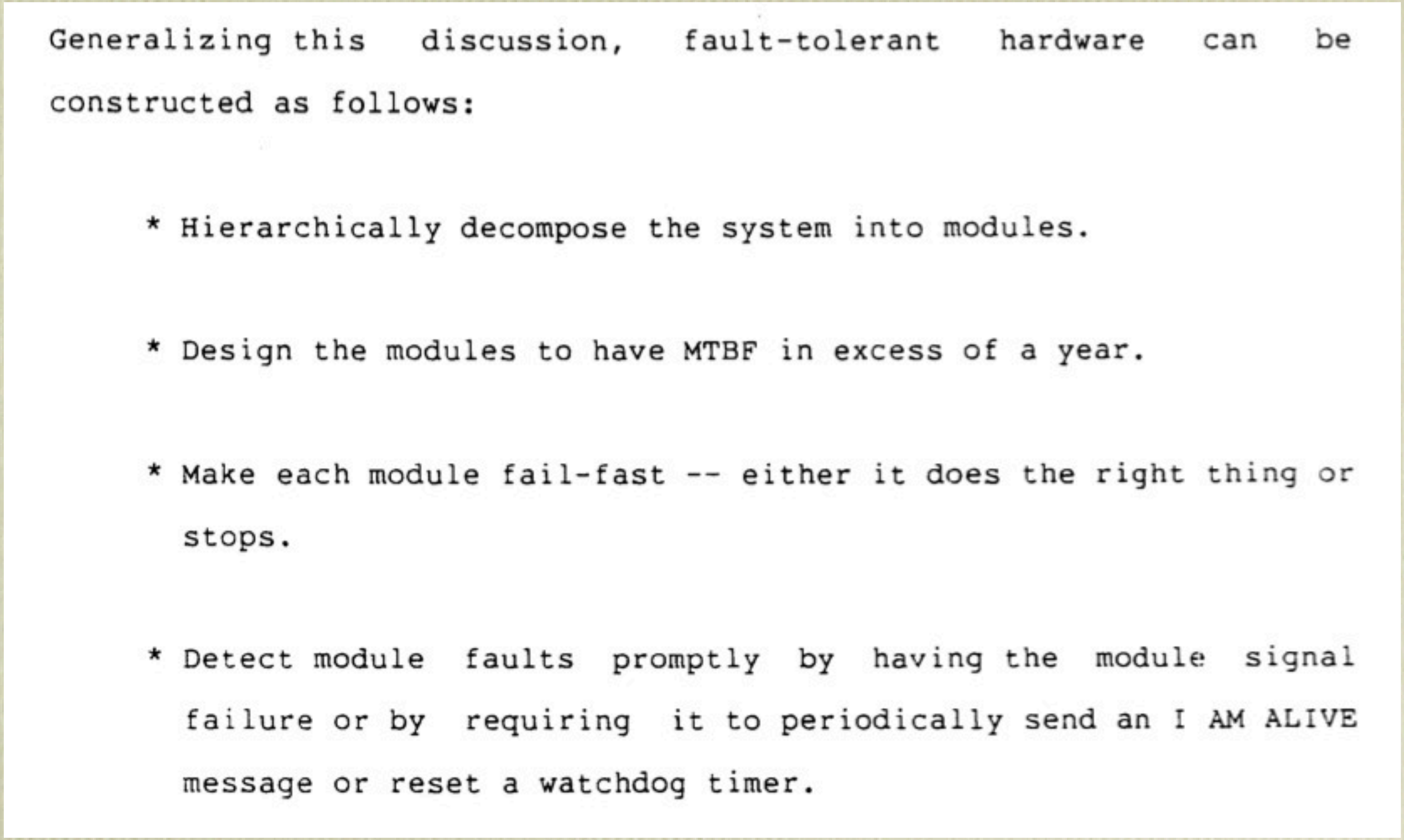
Why Do Computers Stop and What Can Be Done About It?

Jim Gray

June, 1985
Revised November, 1985

ABSTRACT

An analysis of the failure statistics of a commercially available fault-tolerant system shows that administration and software are the major contributors to failure. Various approachs to software fault-tolerance are then discussed -- notably process-pairs, transactions and reliable storage. It is pointed out that faults in production software are often soft (transient) and that a transaction mechanism combined with persistent process-pairs provides fault-tolerant execution -- the key to software fault-tolerance.

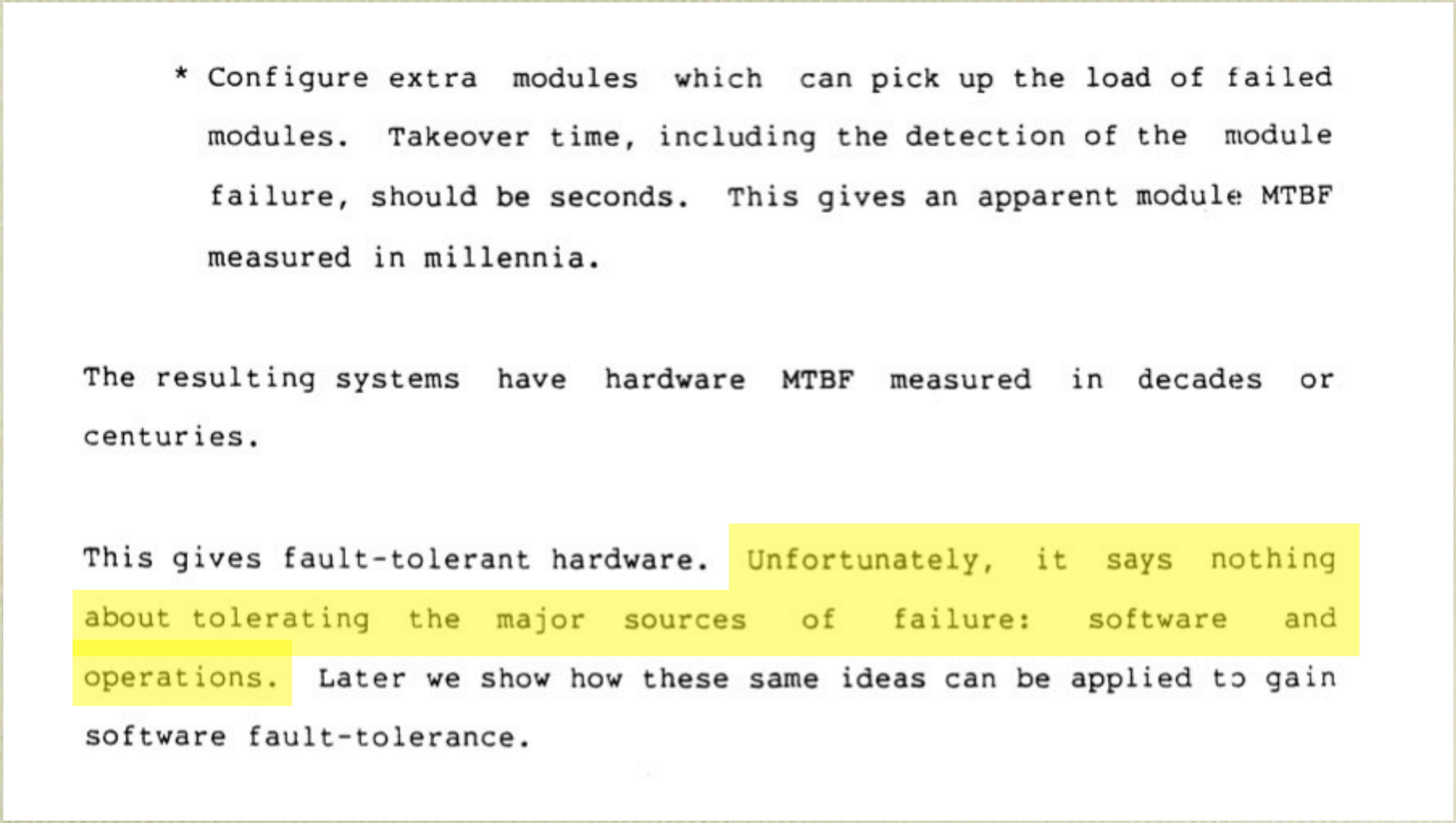

Generalizing this discussion, fault-tolerant hardware can be constructed as follows:

* Hierarchically decompose the system into modules.

* Design the modules to have MTBF in excess of a year.

* Make each module fail-fast -- either it does the right thing or stops.

* Detect module faults promptly by having the module signal failure or by requiring it to periodically send an I AM ALIVE message or reset a watchdog timer.



* Configure extra modules which can pick up the load of failed modules. Takeover time, including the detection of the module failure, should be seconds. This gives an apparent module MTBF measured in millennia.

The resulting systems have hardware MTBF measured in decades or centuries.

This gives fault-tolerant hardware. Unfortunately, it says nothing about tolerating the major sources of failure: software and operations. Later we show how these same ideas can be applied to gain software fault-tolerance.

What do we do when we detect an error?

- Mask it (try again)
- Do nothing (*crash later - not a totally brilliant idea)*
- *Or ...*

LET
IT
CRASH

Programming the Ericsson Diavox (1976)

If you're in a three-way call at any time you can press the # key then press 1 to talk to party 1
2 to talk to party 2
or * to enter a conference call

```
if(state == 3waycall && key == “#”){
    key = get_next_key();
    if(key==”1”){
      park(2);
      connect([self,1]);
   } elseif(key==”2”){
      park(1);
      connect([self,2]);
   } elseif (key==”*”){
       connect([self,1,2]);
   } elseif(key=”onhook”){
      /* Uuugh what do I do here */
 }
```

Defensive programming

Oh Dear

- The Spec tells what to do when things happen
- The Spec does not say what to do when the behavior goes “off-spec”
- The number of ways we can go “off spec” is huge
- Most specifications do not include failure analysis, and do not say what to do when you are “off spec”

Joe: “So what happens if we’re in a 3-way conference, and the guy processes hash and then puts the hook down, and doesn’t press 1 2 or star?”

Bernt: “So what you do is stop the conference, send the phone a ring tone and when they answer go back to the point where you were expecting them to enter 1 2 or star.”

Joe: “But that’s not in the spec.”

Bernt: “But everybody knows.”

Joe: “I didn’t know.”

Calls are “files”

- If a process crashes the OS closes all files opened by the process
- If a call crashes the OS closes all calls opened by the process

- The OS’s job is to “keep files safe” (ie it maintains invariants)

Let it crash philosophy

- If a processes crashes the OS detects this
- The OS protects the resources being used by the process
- Programs should crash when going off spec

```
if(state == 3waycall && key == “#”){
    key = get_next_key();
    if(key==”1”){
      park(2);
      connect([self,1]);
   } elseif(key==”2”){
      park(1);
      connect([self,2]);
   } elseif (key==”*”){
      connect([self,1,2]);
   } else{
      exit(out_of_spec1);
   }
 }
```

Defensive programming

```
confcall("#") ->
    case get_next_key() of
        "1" ->
            park(2);
            connect([self,1]);
        "2" ->
            park(1);
            connect([self,2]);
        "*" ->
           connect([self,1,2])
   end.
```

Failed Patten matching provides the exit

Non defensive programming - there is no error detection or correction code

Are hardware and software faults are fundamentally different?

Are there any pure functions?

Class (a) functions: If computing f(X) fails and f is a pure function computing f(X) will always fail.

Class (b) functions: If computing f(X) fails and f is a non-pure function it might succeed if we call f(X) again.

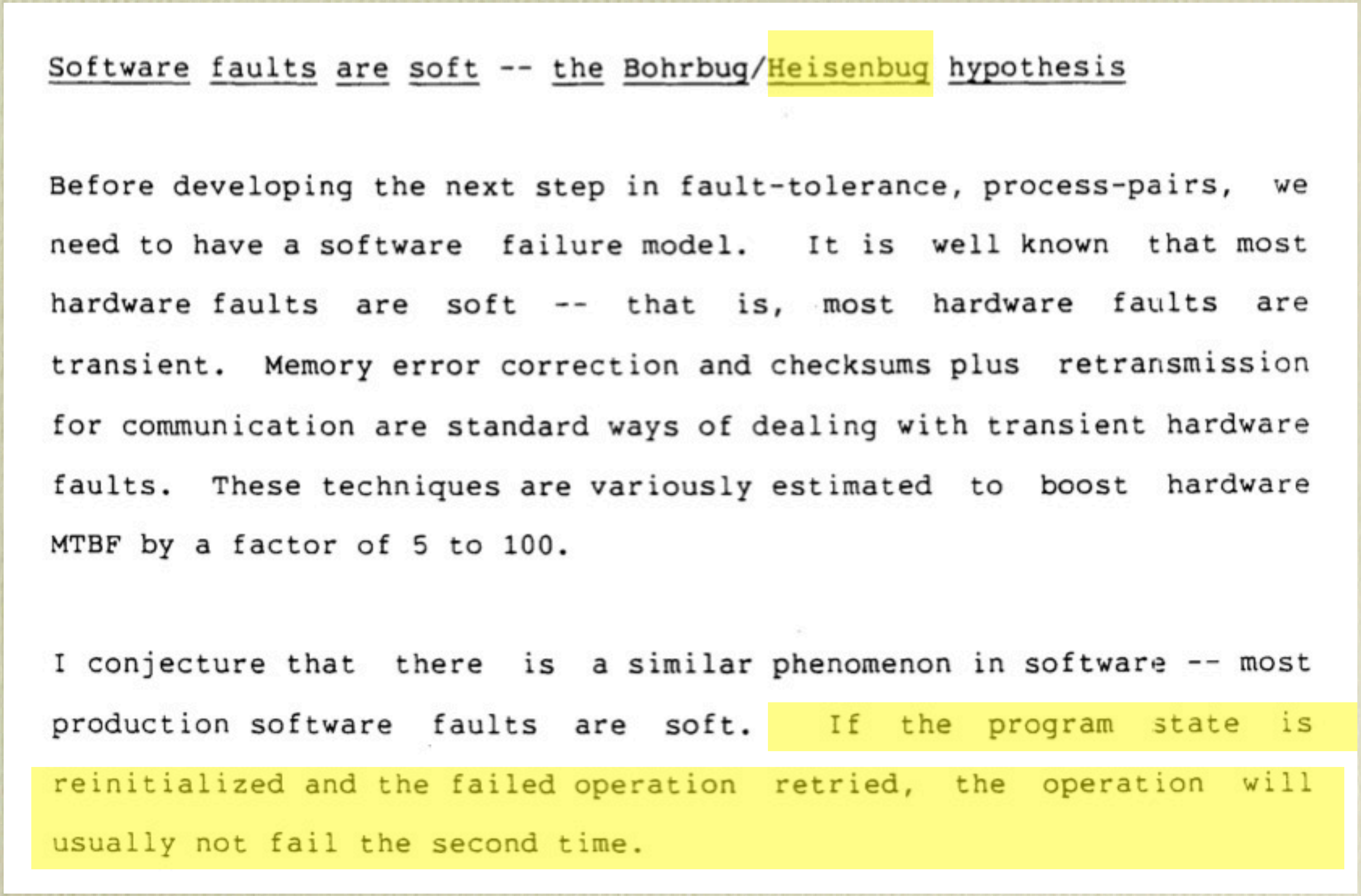
Is this a pure function?

```
function f(){
    int a = 10,
    int b = 2,
    return a/b
}
```

Cosmic ray hits the memory cell where b is stored and changes the 2 into zero

```
function f(){
    int a = 10,
    int b = 2,
    return a/b
}
```

A heisenbug



Software faults are soft -- the Bohrbug/Heisenbug hypothesis

Before developing the next step in fault-tolerance, process-pairs, we need to have a software failure model. It is well known that most hardware faults are soft -- that is, most hardware faults are transient. Memory error correction and checksums plus retransmission for communication are standard ways of dealing with transient hardware faults. These techniques are variously estimated to boost hardware MTBF by a factor of 5 to 100.

I conjecture that there is a similar phenomenon in software -- most production software faults are soft. If the program state is reinitialized and the failed operation retried, the operation will usually not fail the second time.

- **Heisenbug** - Bug that that seems to disappear or alter its behavior when one attempts to study it
- **Bohrbug** - A "good, solid bug". Like the deterministic Bohr atom model, they do not change their behavior and are relatively easily detected.
- **Mandelbug** - (named after Benoît Mandelbrot's fractal) is a bug whose causes are so complex it defies repair, or makes its behavior appear chaotic or even non-deterministic.
- **Schrödinbug** (named after Erwin Schrödinger and his thought experiment) is a bug that manifests itself in running software after a programmer notices that the code should never have worked in the first place.
- **Hindenbug** (named after Hindenburg disaster) is a bug with catastrophic behavior.

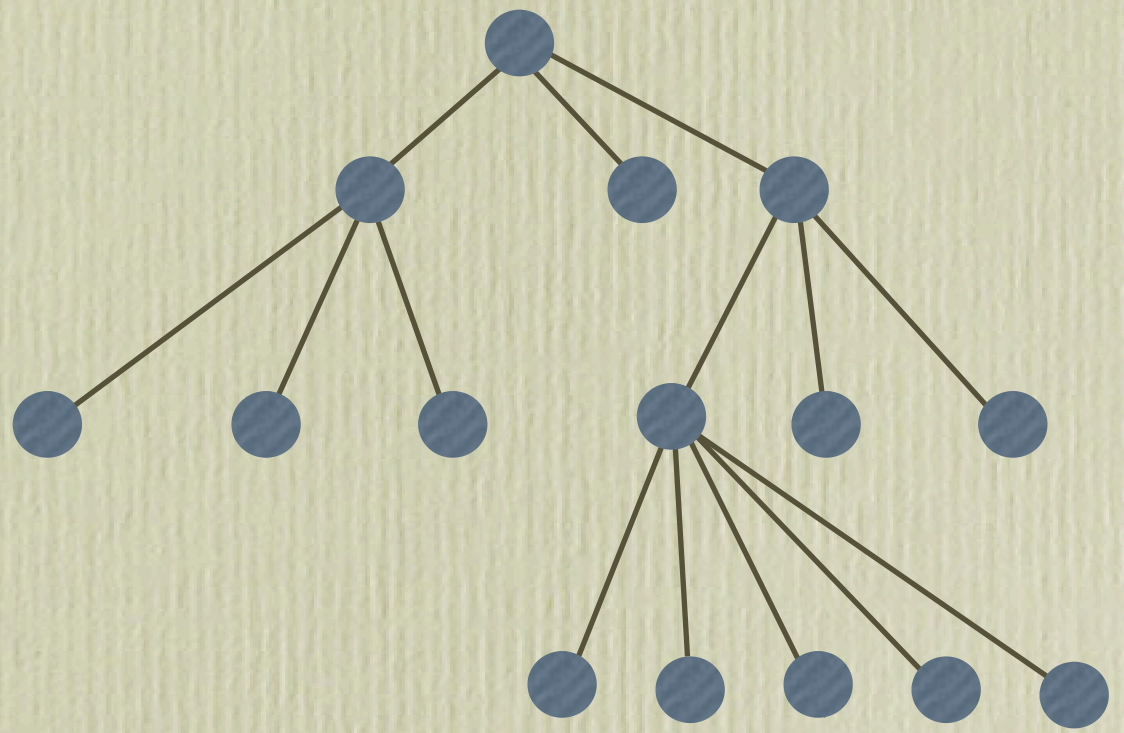
Source: wikipedia

- If a process fails restart it (*fixes many heisenbugs, especially those due to subtle timing errors*)

- If you have tried restarting a process more than N times in K seconds, then give up. Try and do something simpler instead.

- Build trees of processes, if low-level nodes fail and cannot be restarted fail higher up the tree

Supervision trees



supervisors

workers

Don't forget the manual backup :-)

The failure model
is part of the specification
(especially for air-traffic
control software etc.)

The customer should
understand the failure model

I want fault tolerant storage

That's impossible

We'll make three copies of your data, on three different machines. We'll guarantee that if one machine crashes you'll never lose any data

what happens if 2 machines crash at the same time

You can still save data on the third machine, but it will be unsafe. Our guarantee will not apply.

But I want more safety

We'll make five copies of your data, on five different machines. We'll guarantee that if two machines crashes you'll never lose any data

what happens if 3 machines crash at the same time

You can still save data on machine 4 and 5, but it will be unsafe. Our guarantee will not apply.

Why is it unsafe? - it's stored on two machines

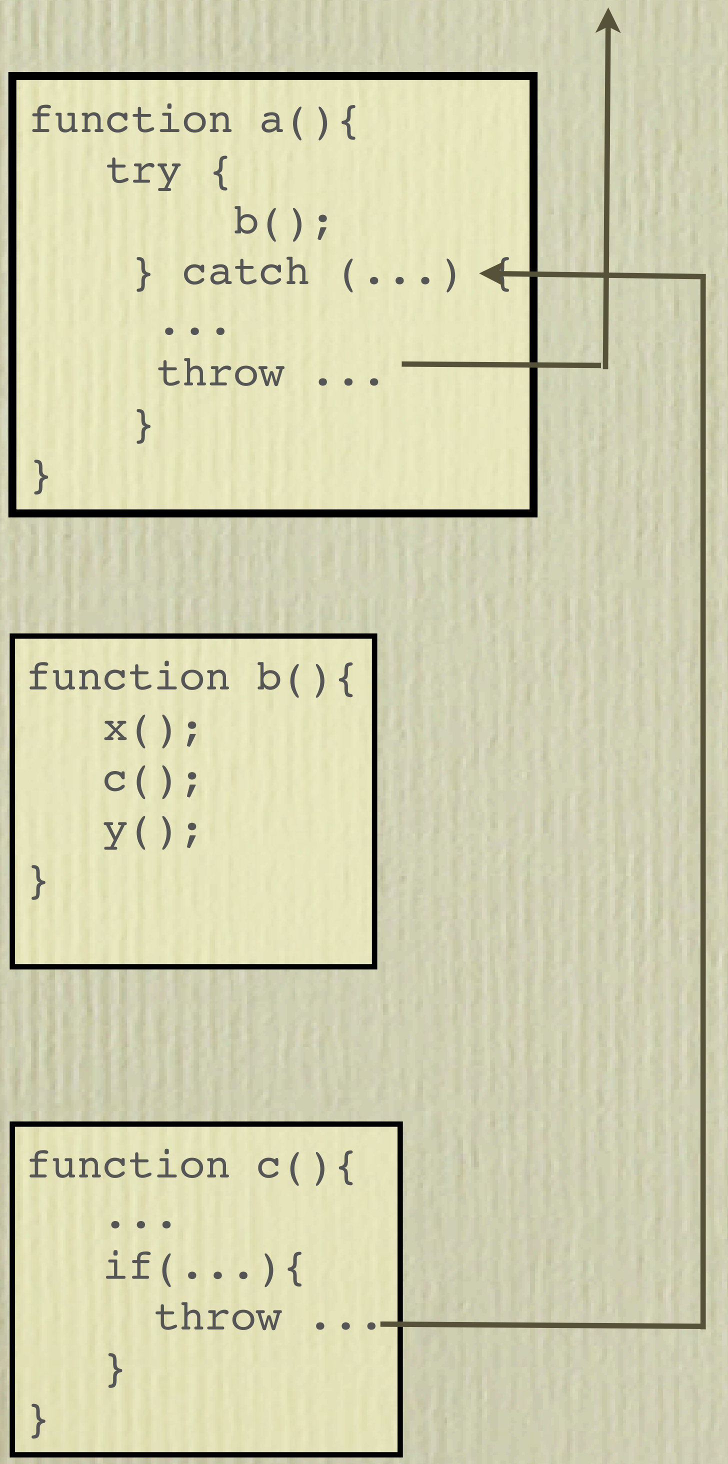
Because when machines 1,2,3 come back to life they might outvote the changes on machines 4 and 5

You have to explain in the contract the failure assumptions and what will happen if these failures occur. If a failure occurs that is not planned it is not covered by the contract.

"act of God"

Detecting Errors

Sequential Languages



```
function a(){
   try {
        b();
    } catch (...)
     ...
     throw ...
    }
}
```

```
function b(){
   x();
   c();
   y();
}
```

```
function c(){
   ...
   if(...){
     throw ...
   }
}
```

- Function calls put call frames on the stack
- Try instruction put catchpoints on the stack
- Exceptions unwind the stack to the last catchpoint

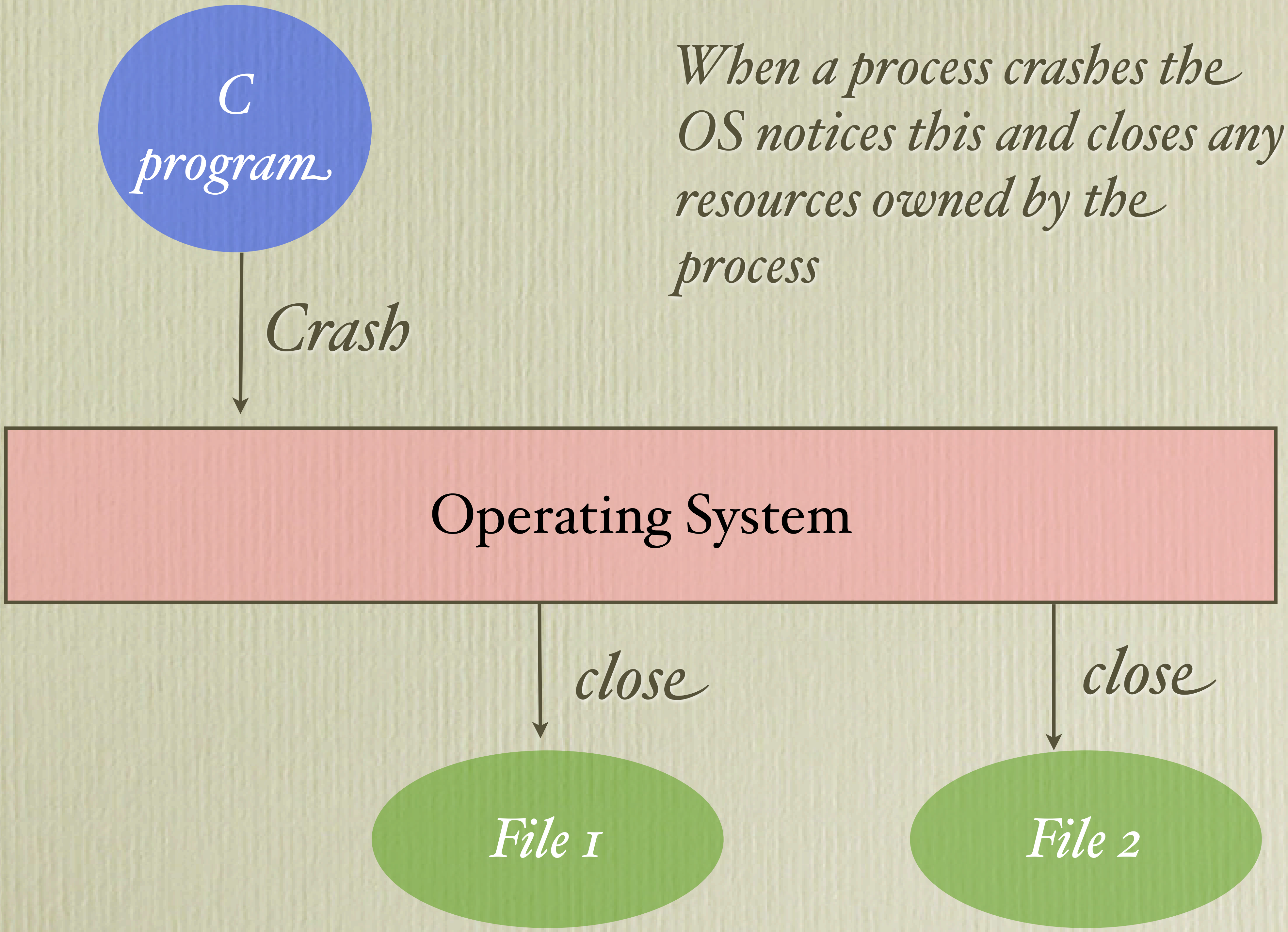
Uncaught Exceptions

- What happens if the exception gets to the top of the stack and no catchpoint handlers is found?

  Java: print a stack trace and exit
  C: core dumped

  Erlang: Process dies some other process on the same or some other machine possibly catches the error

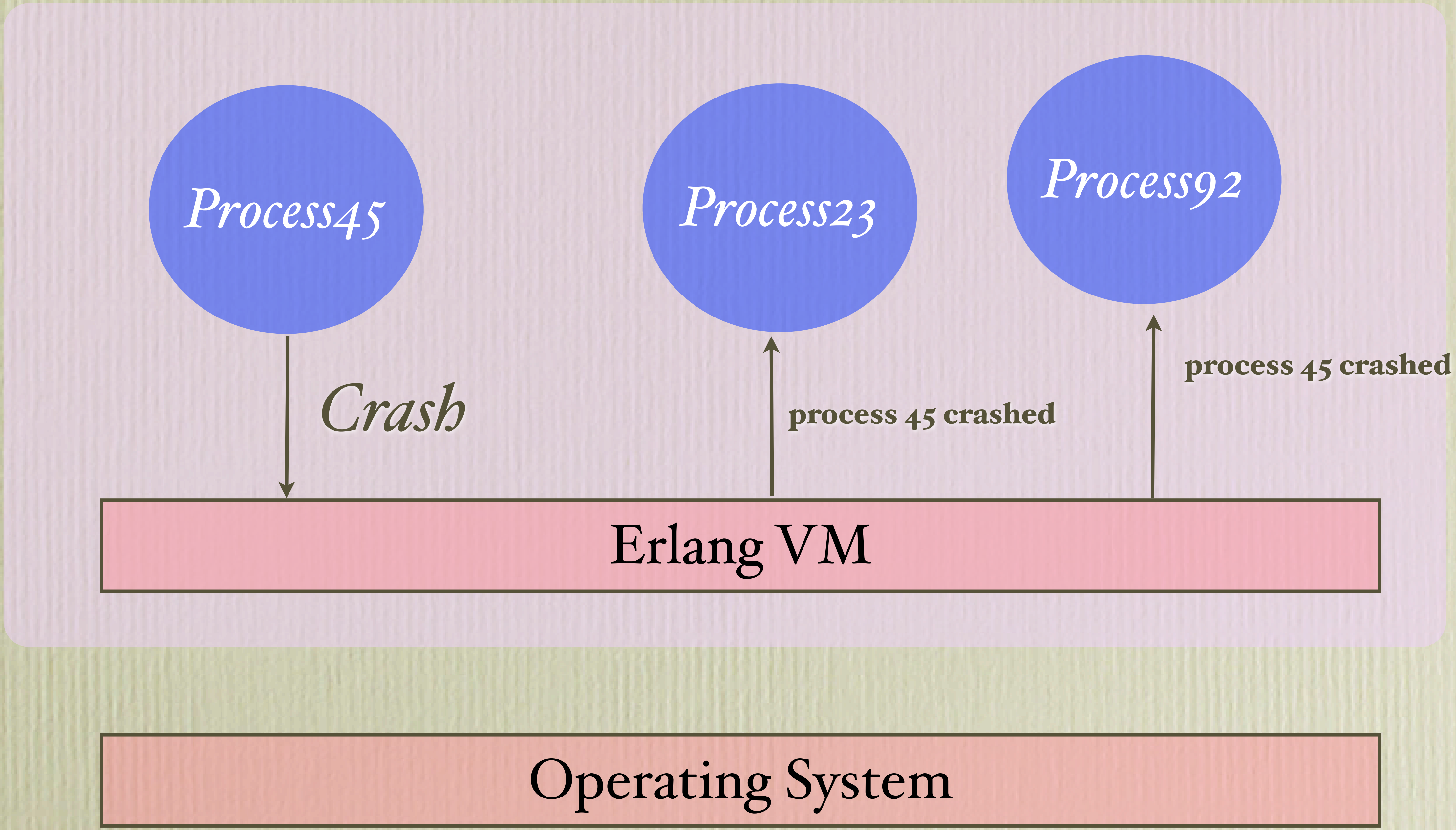
Sequential Languages



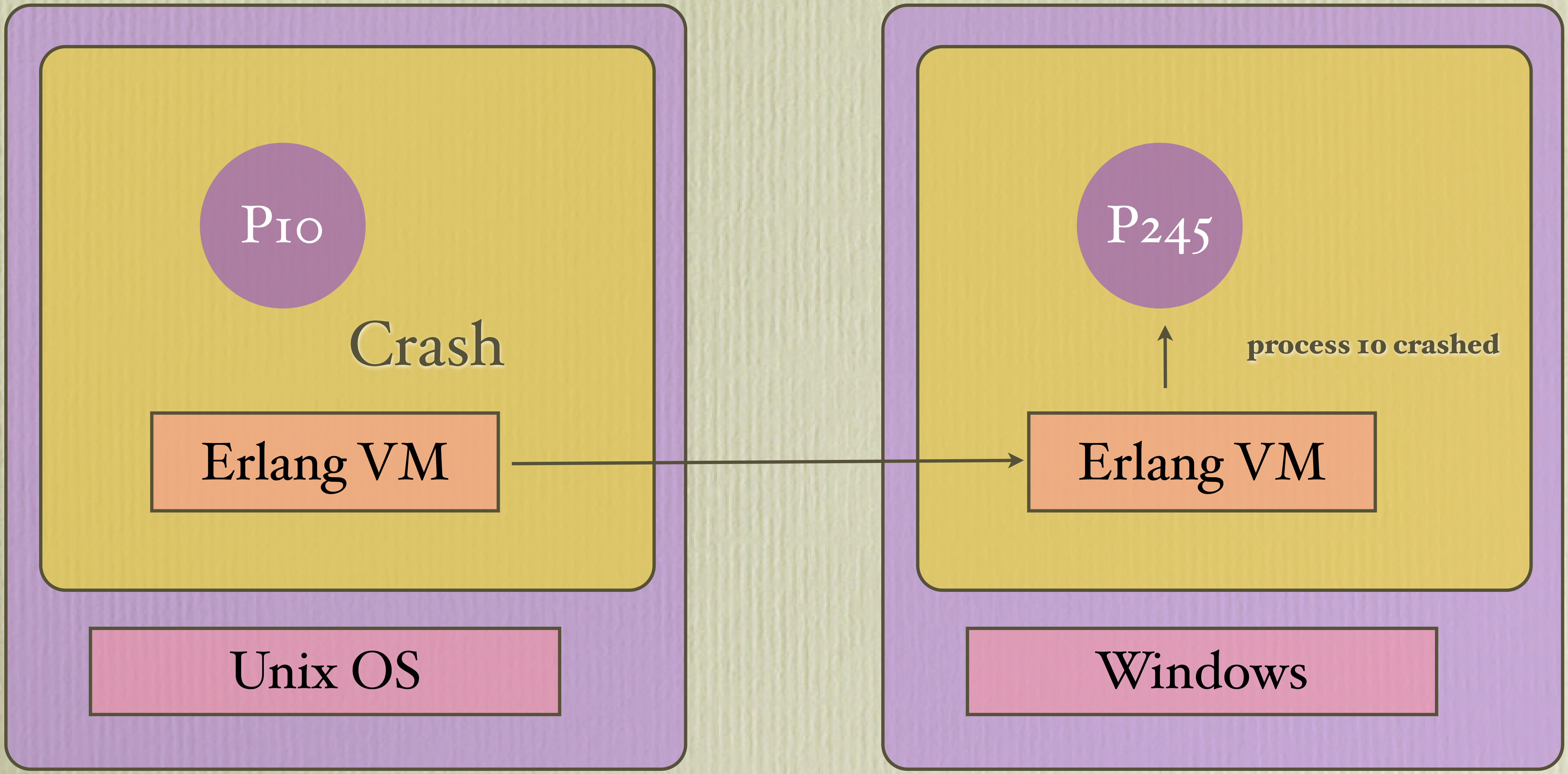
When a process crashes the OS notices this and closes any resources owned by the process

Erlang

When an Erlang process crashes the Erlang VM notices this and sends messages to any linked processes



Erlang


P10
Crash
Erlang VM
Unix OS
P245
process 10 crashed
Erlang VM
Windows

Demo

1. Start a process on one machine. Send it a message so it crashes.

2. Start a process on one machine. Send it a message so it crashes. Detect the crash

3. Start a process on a remote machine. Send it a message so it crashes. Detect the error on a remote machine.

prog1.erl

```
-module(prog1).
-export([loop/0]).

loop() ->
  receive
    N ->
      io:format("node=~p 1/~p = ~p~n",
         [node(), N, 1/N]),
      loop()
  end.
```

One machine

```
$ erl
Eshell V5.10.1  (abort with ^G)
1> P = spawn(prog1, loop, []).
<0.34.0>
2> P ! 12.
node=nonode@nohost 1/12 = 0.08333333333333333
12
3> P ! 0.
0
4>
=ERROR REPORT==== 29-Nov-2013::13:07:26 ===
Error in process <0.34.0> with exit value:
  {badarith,[{prog1,loop,0,[{file,"prog1.erl"},{line,7}]}]}
4> P ! 12.
12
```

monitor.erl

```erlang
-module(monitor).
-export([process/1]).

process(Pid) ->
  spawn(fun() ->
          process_flag(trap_exit, true),
          link(Pid),
          monitor(Pid)
        end).

monitor(Pid) ->
 receive
   Any ->
       io:format("Monitor ~p received ~p~n",[Pid,Any]),
       monitor(Pid)
 end.
```

One machine + Monitor

```
Eshell V5.10.1  (abort with ^G)
1> P = spawn(prog1, loop, []).
<0.34.0>
2> monitor:process(P).
<0.36.0>
3> P ! 12.
node=nonode@nohost 1/12 = 0.08333333333333333
12
4> P ! 0.
Monitor <0.34.0> received
 {'EXIT',<0.34.0>,
    {badarith,
      [{prog1,loop,0,
     [{file,"prog1.erl"},{line,7}]}]}}
```

The process dies and a message is sent to the monitor process

Two machines and a monitor

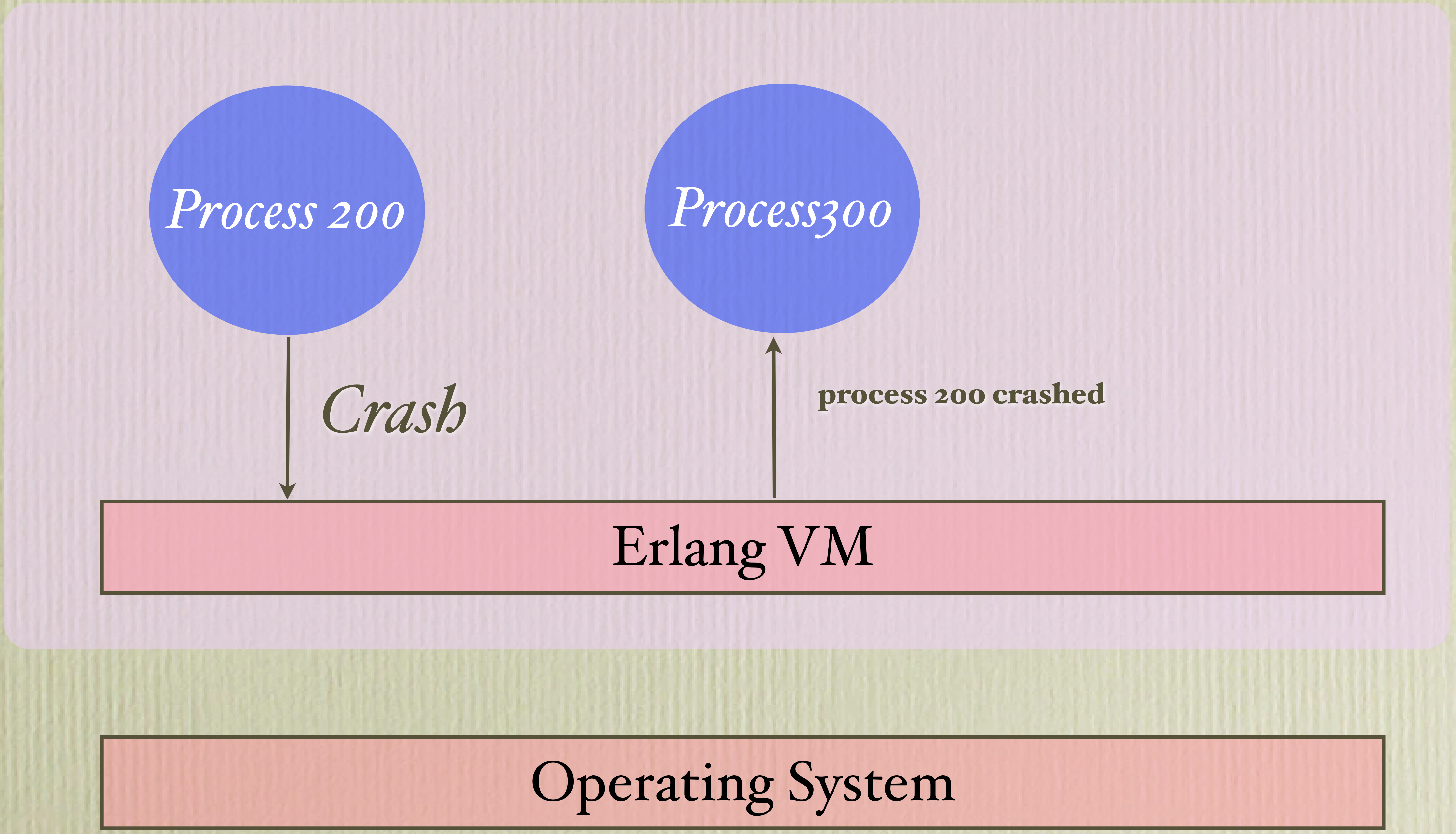
```
$ erl -sname two
(two@joe)1>
```

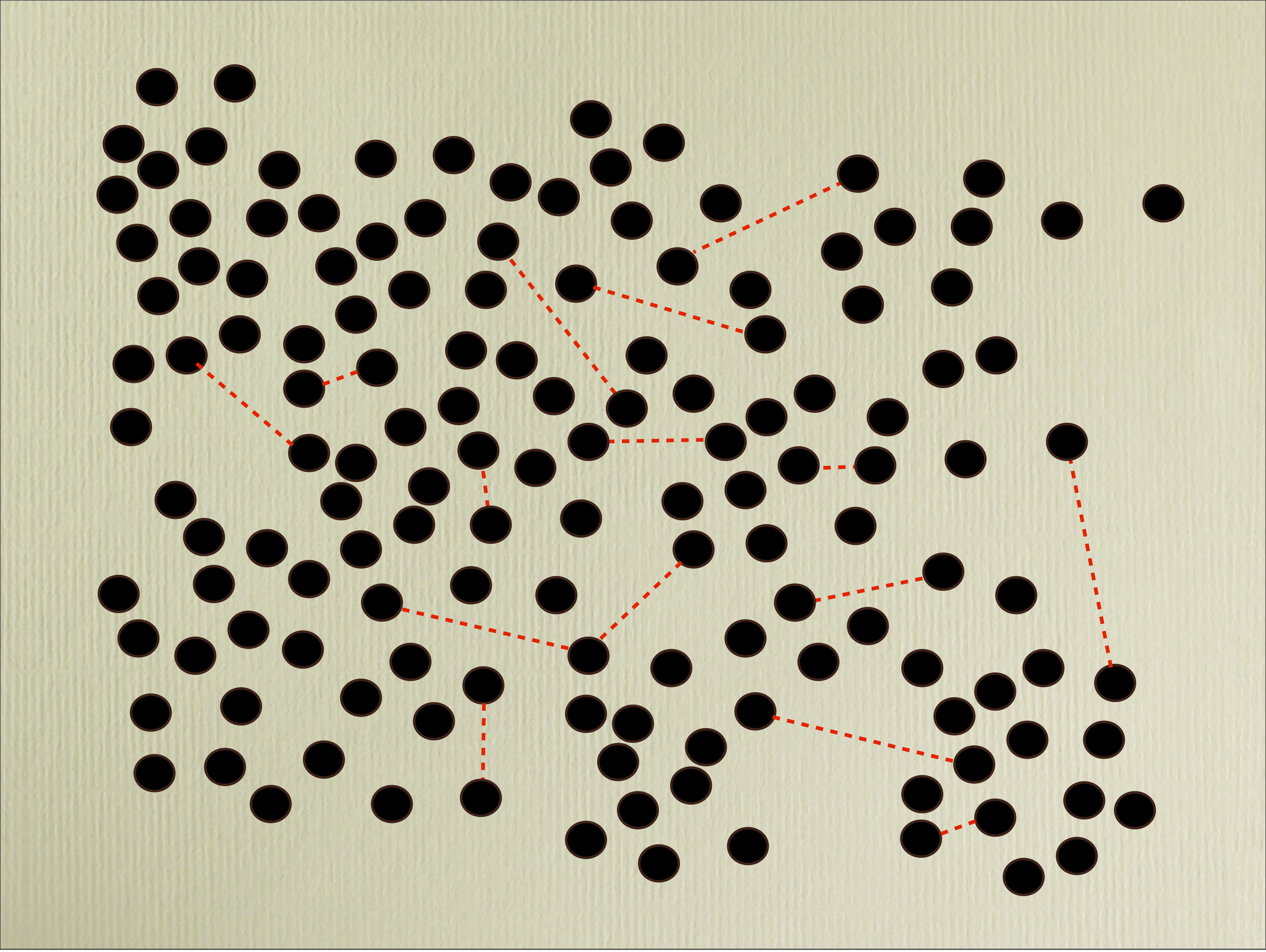
```
$ erl -sname one
(one@joe)1> P = spawn('two@joe', prog1, loop, []).
<6803.43.0>
(one@joe)2> monitor:process(P).
<0.47.0>
(one@joe)4> P ! 10.
10
node=two@joe 1/10 = 0.1
(one@joe)5> P ! 0.
0
Monitor <6803.43.0> received
  {'EXIT',<6803.43.0>,
             {badarith,
                [{prog1,loop,0,
                     [{file,"prog1.erl"},{line,7}]}]}}
```

Or we could kill the machine?

Reminder

When an Erlang process crashes the Erlang notices this and tells and linked processes





Defensive programming is a consequence of a bad concurrency model

We’ve detected an error what do we do next?

I’ve detected an error, what should I do?

Try again - it might be a heisenbug

I tried again ten time but it didn’t help

Ok - give up, and tell you’re boss you gave up. You did your best, nobody will blame you.

We have a problem Huston

.... *@!%$!!**&%%%!!!%$#@*** #$@

Do not fail silently
if you cannot do exactly what
you are supposed to do crash.
Somebody else will fix the
problem

Summary

- No shared memory
- Pure message passing
- Remote Error Detection
- Replicated hardware and software on separated machines
- Crash when you get an error
- Do not fail silently
- Some other process fixes the error

Does this strategy work?