SDCC 2016

中国软件开发者大会

SOFTWARE DEVELOPER CONFERENCE CHINA

Message passing concurrency made easy

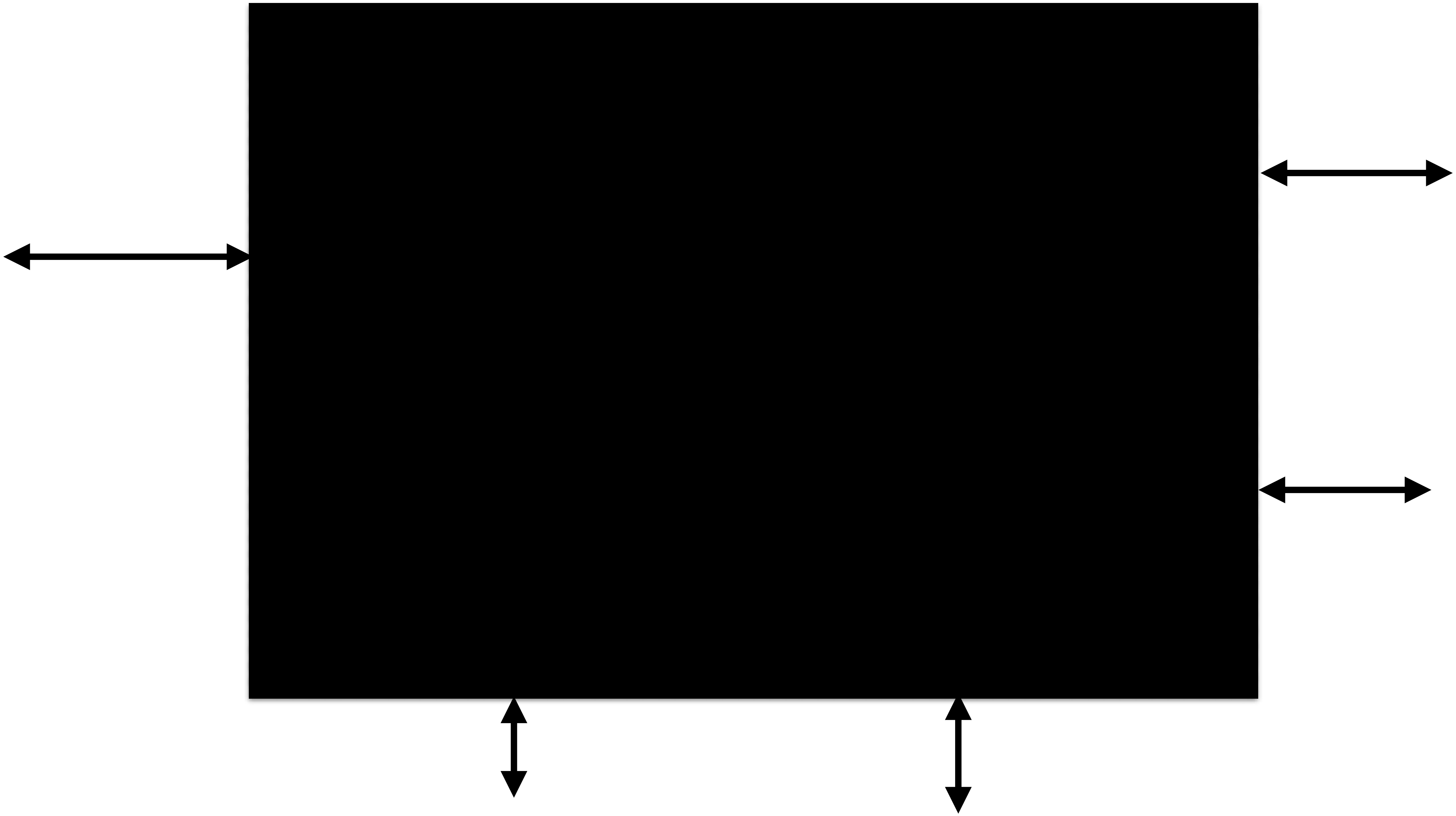
Joe Armstrong

Plan

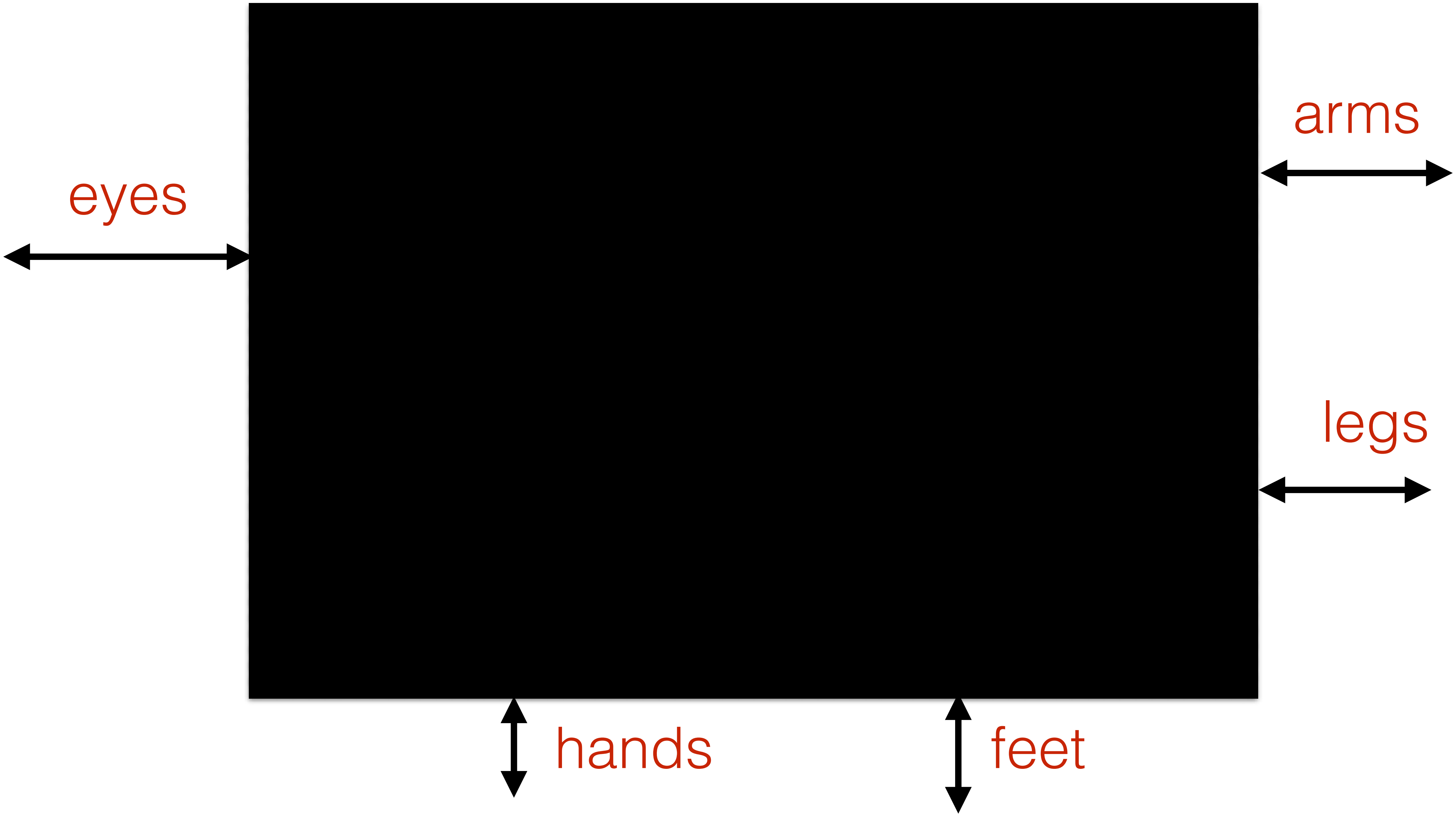
- Tell you why pure message passing is great
- Tell you how we implemented systems with pure message passing
- Tell you how you can do this
- Tell you some other good things to do
- You leave here and write better software

Not necessarily in this order

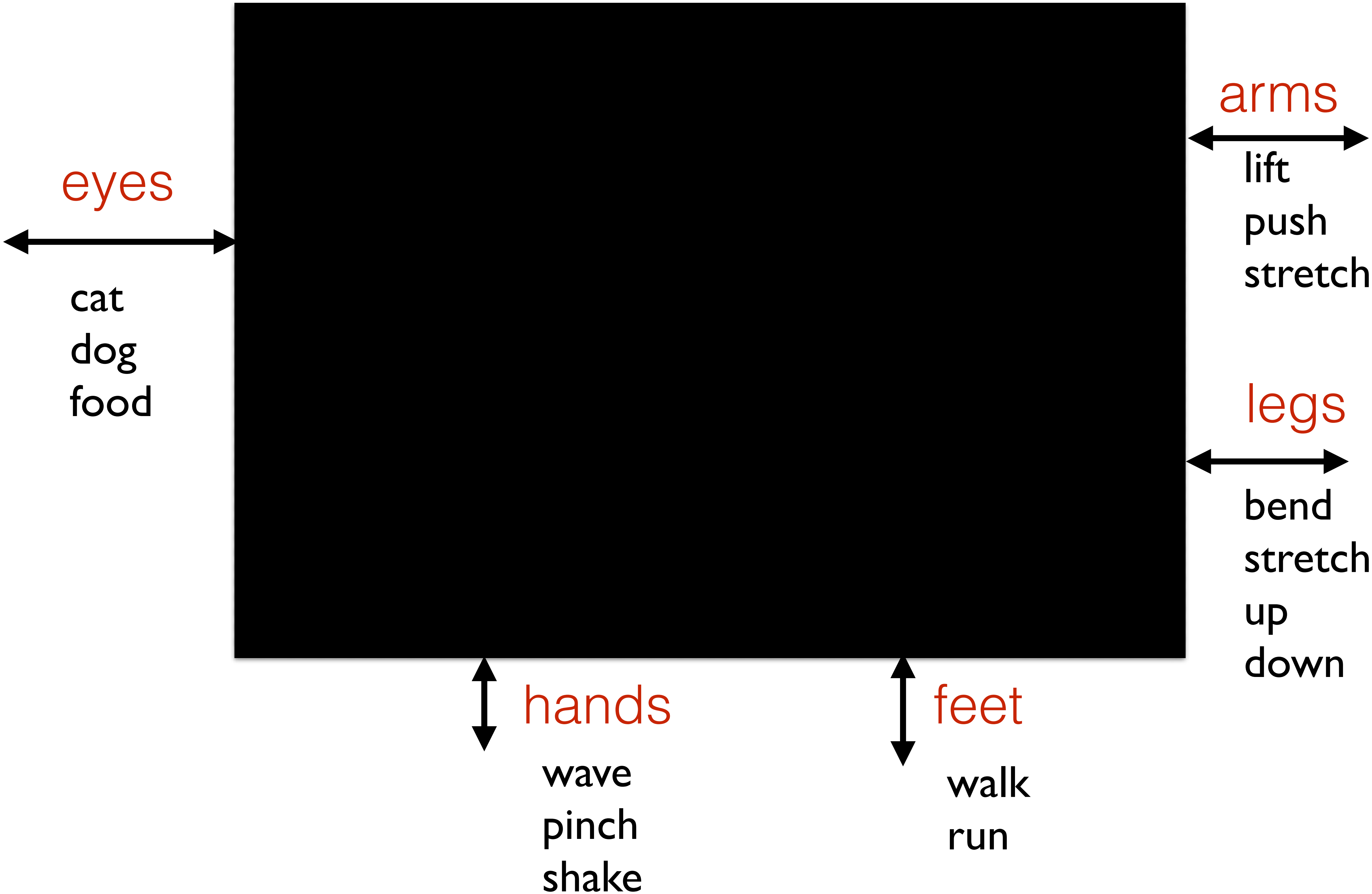
How can we understand complex software?



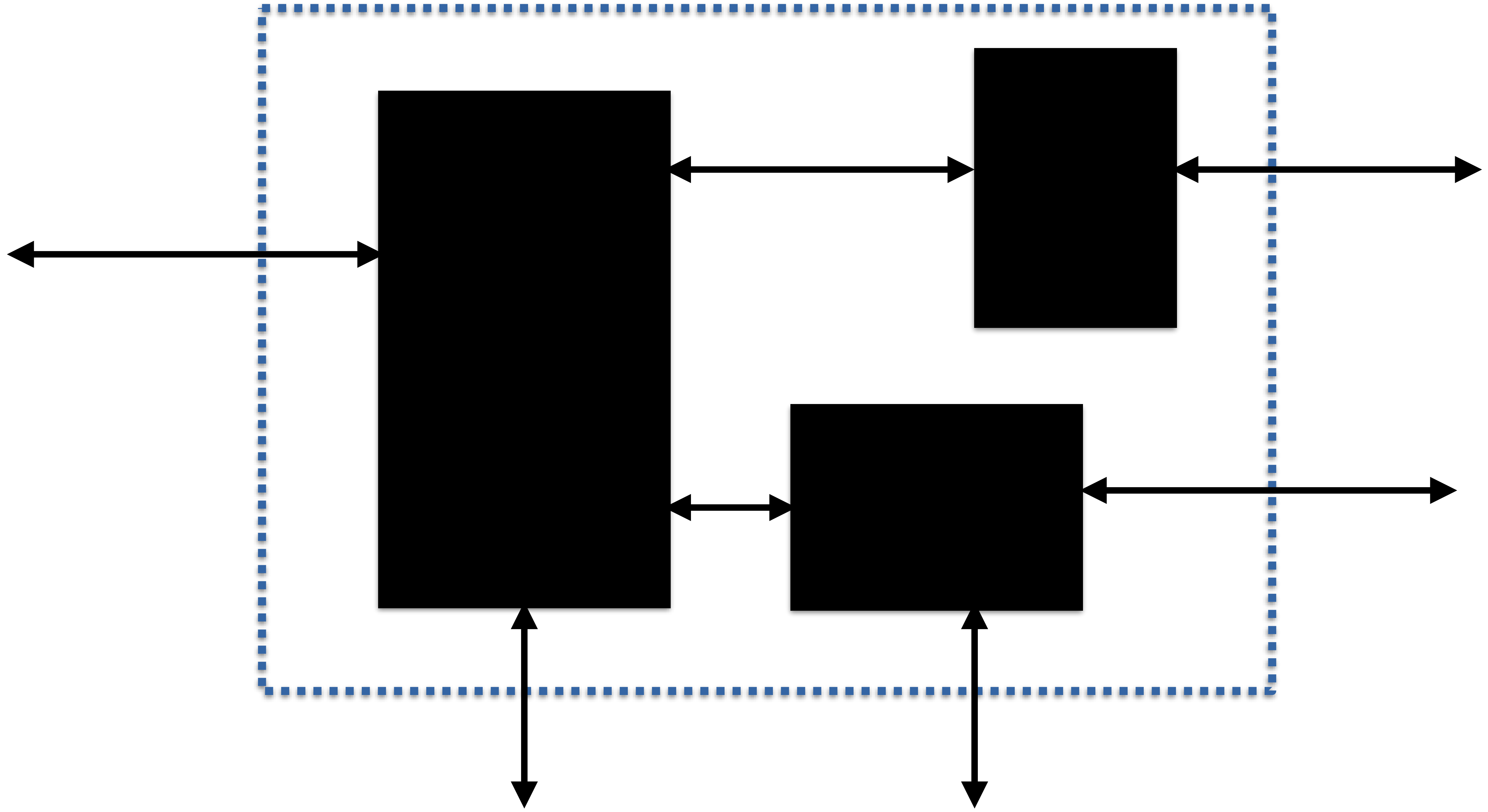
Name the channels



Add messages



Add the logic



Three parallel machines communicating with messages

Basic Properties

- Composable (build big things from small things)
- Parts must run in parallel
- Failure must be contained
- Messages must be well-defined
- Protocols must be well-defined
- Allow reasoning about behaviour to take place at different levels
- Observable
- Made from small validated parts

This is how we make hardware not software

To program
systems of communicating
objects we need to make
it easy to write parallel
programs

Why do we want to write parallel programs?

- World is parallel
- We want ONE way to program
- We want to reduce complexity

- but ….

It’s actually difficult to write parallel programs so …

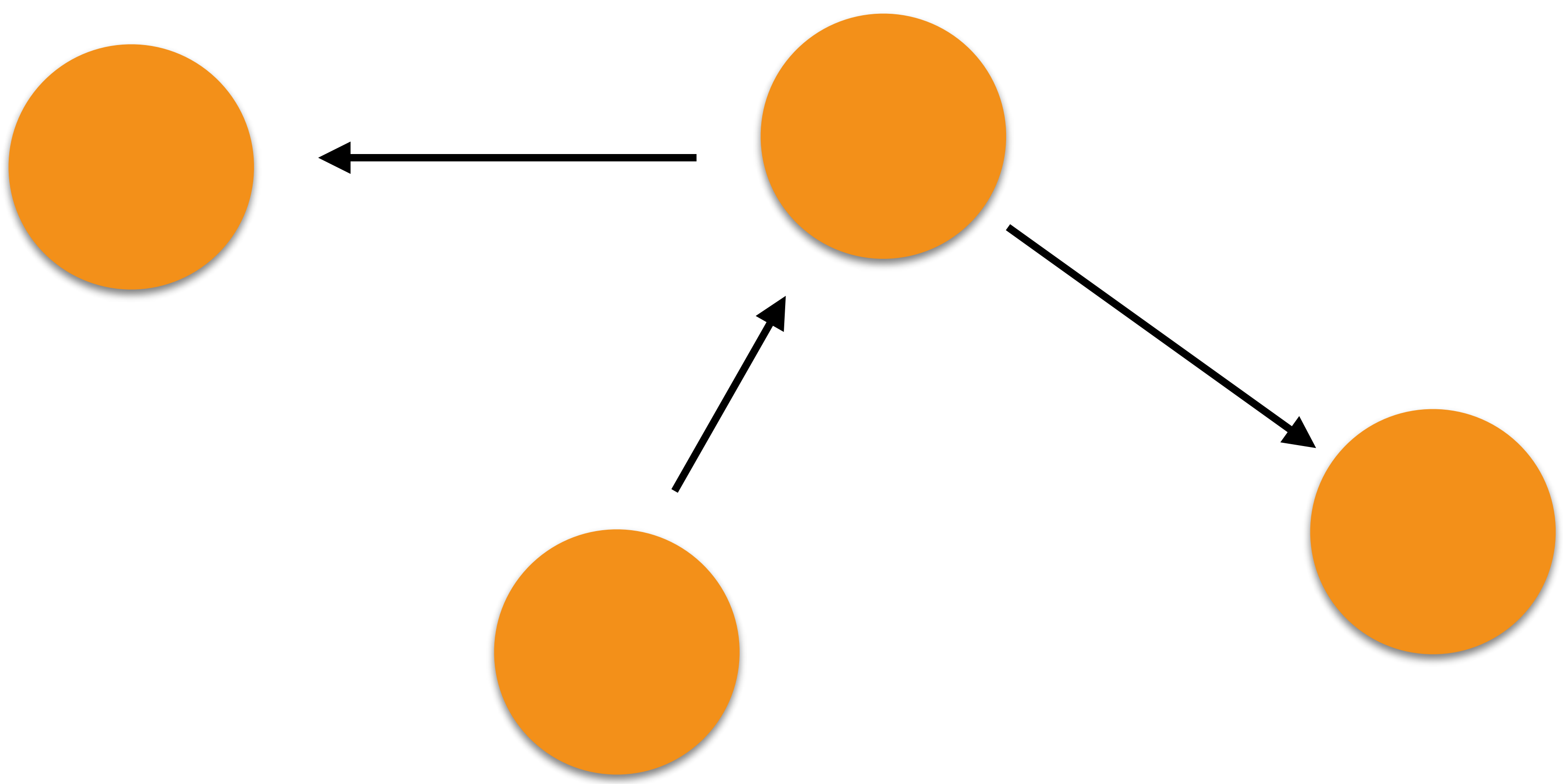
Make it easy to write ~~parallel~~ concurrent programs

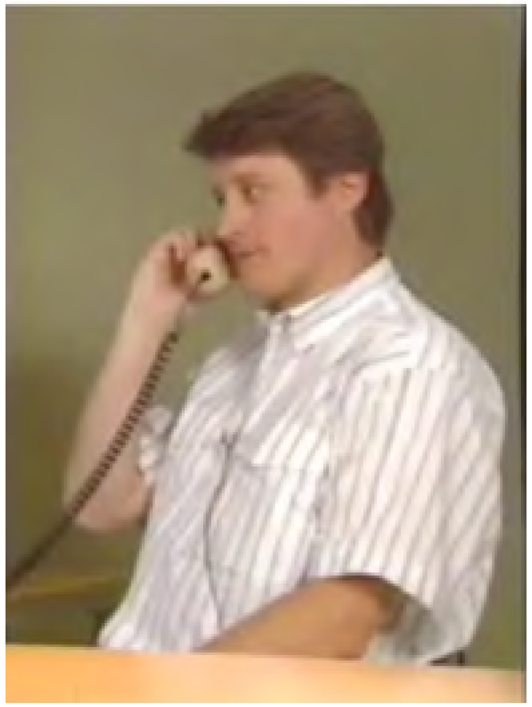
Automate
the parellelization
of concurrent
programs

Programmer decides the process model

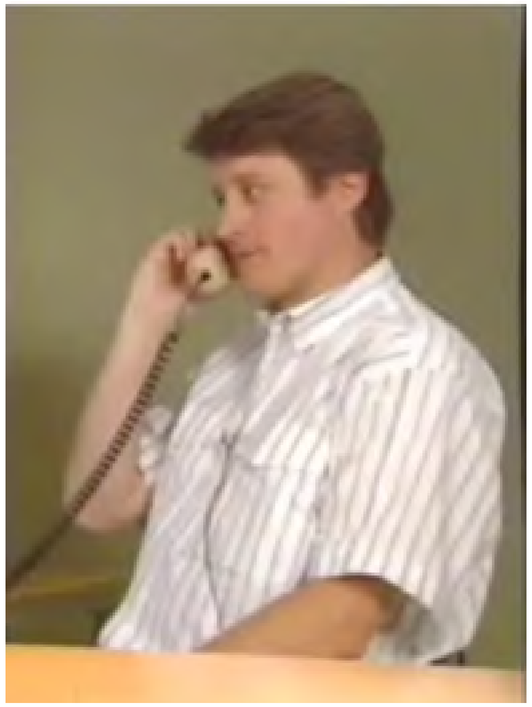
"Process model" = "units of concurrency"

Observe the world and the communication channels





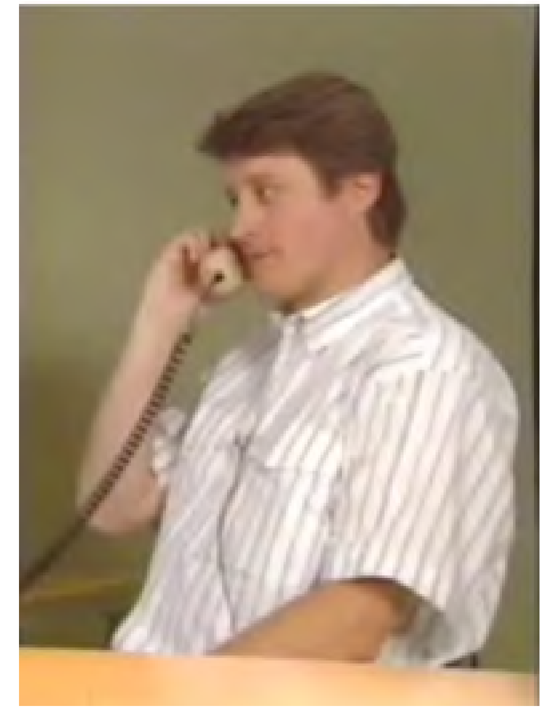
How many processes?



How many channels?

What are the messages?

“Hello Robert” →



← “Hello Joe”

“Hello Mike” ↘

```
receive
    Msg ->
        joe ! “Hello Joe”
end,
```

```
robert ! “Hello Robert”
receive
    Msg ->
        mike ! “Hello Mike”
end
```

Code is based on OBSERVATION

What are the messages?

How do we receive the messages?

```
joe ! {self(), "....."}

mike ! {self(), "....."}

robert ! {self(), "..."}
```

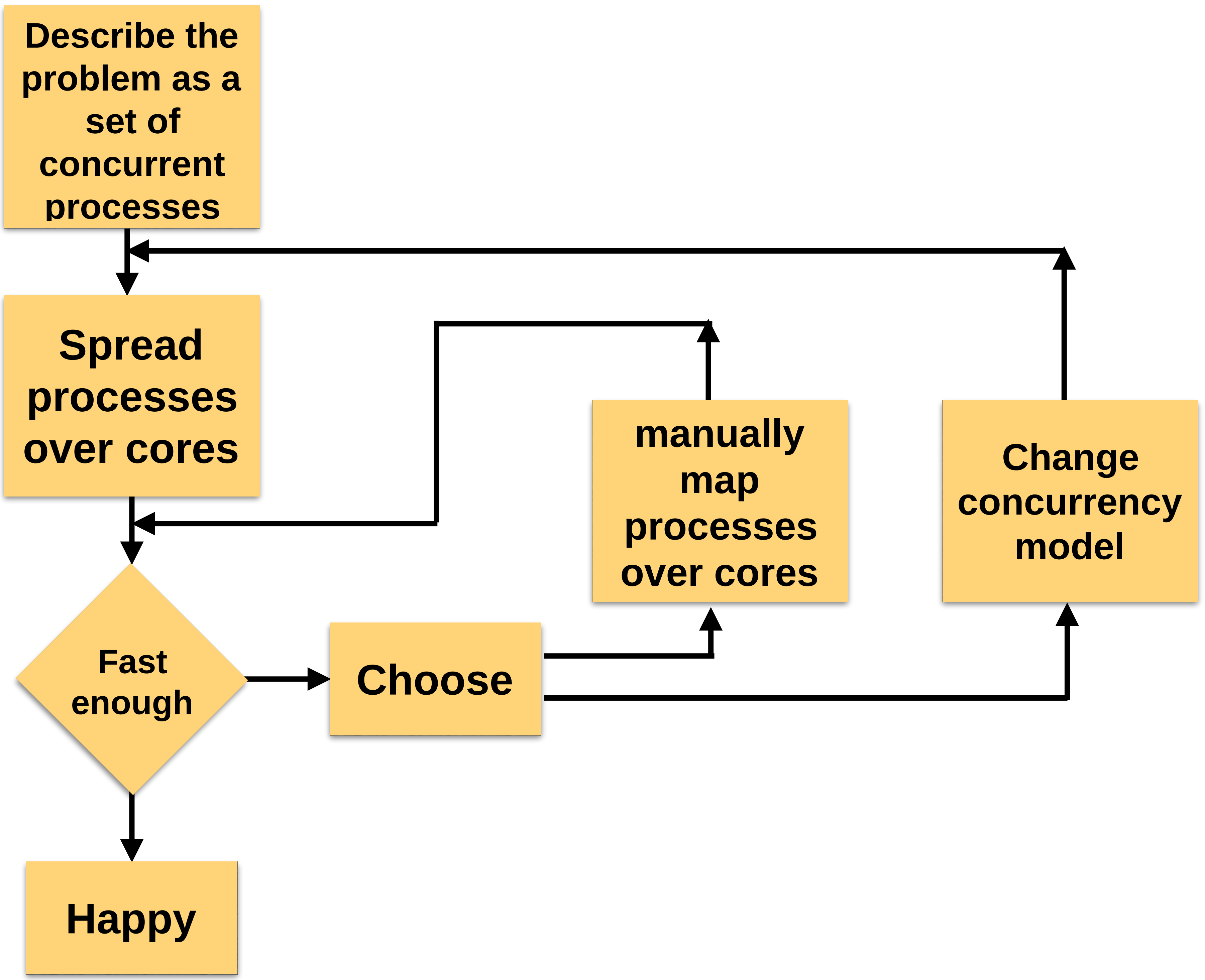
```
receive
    {Joe, Msg} ->
        ...
end
```

One parallel operation in real world
=
One process

Carl Hewett calls this "physics modelling" (as opposed to computation based on mathematical logic)

One parallel operation in real world
=
One process

Carl Hewett calls this "physics modelling" (as opposed to computation based on mathematical logic)


Describe the problem as a set of concurrent processes
Spread processes over cores
Fast enough
Choose
manually map processes over cores
Change concurrency model
Happy

How do we make it easy to write concurrent programs?

have a language
with only 3 concurrency
primitives

Easy to
remember

spawn
send
receive

Spawn creates a parallel process

No

shared memory
semaphores
mutexes
monitors
spin locks
critical regions
futures
locks
caches
threads
thread-safety

Pure message passing

Why Pure Message Passing?

and isolation

It’s PURE OO

Alan Kay On Messaging

From: Alan Kay <alank@wdi.disney.com>
Date: 1998-10-10 07:39:40 +0200
To: squeak@cs.uiuc.edu
Subject: Re: prototypes vs classes was: Re: Sun's HotSpot

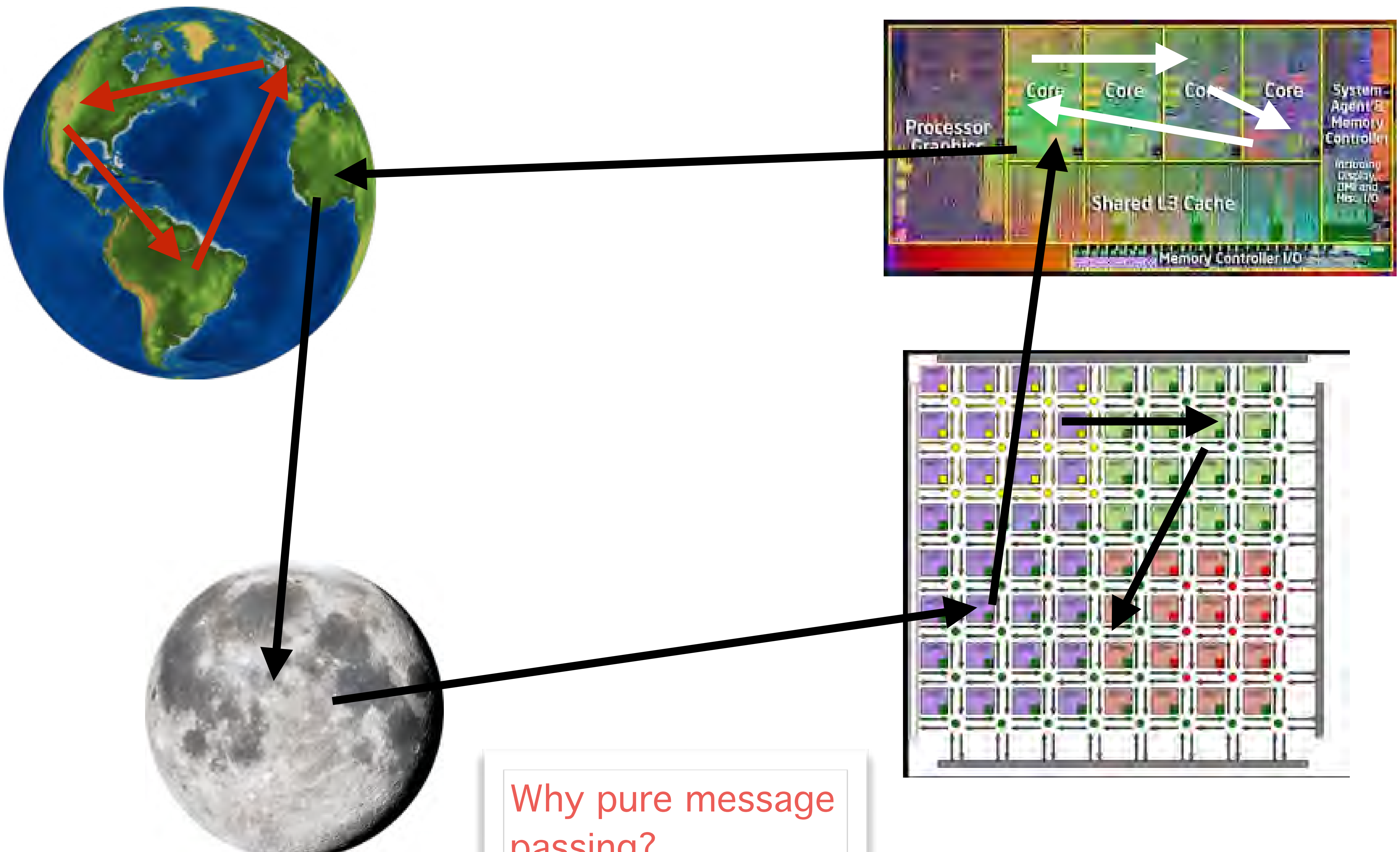
Folks --

Just a gentle reminder that I took some pains at the last OOPSLA to try to remind everyone that Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.

The big idea is "messaging" - that is what the kernal of Smalltalk/Squeak is all about (and it's something that was never quite completed in our Xerox PARC phase). The Japanese have a small word - ma - for "that which is in between" - perhaps the nearest English equivalent is "interstitial". The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be. Think of the internet - to live, it (a) has to allow many different kinds of ideas and realizations that are beyond any single standard and (b) to allow varying degrees of safe interoperability between these ideas.

Why pure message passing?

One programming model



Why pure message passing?

One Programming Model

- Cannot do distributed programming without message passing it's impossible
- Want same way to do distributed and non-distributed programming
- **Must use message passing to do non-distributed programming**

Obeys the laws of physics

Why pure message passing?

Which laws of physics?

Messages travel at <= Speed of light

Causality: If B depends upon the state of A, and A and B are separated in space, then A must send a message to B before B can do anything

We only know how things were
not how things are

Details

spreading processes
is difficult

OTP team at Ericsson (2-3 people know the multi-core part)

100K programmers?? know nothing about multi-core

Their programs should run 0.75 x N times faster on N core computers

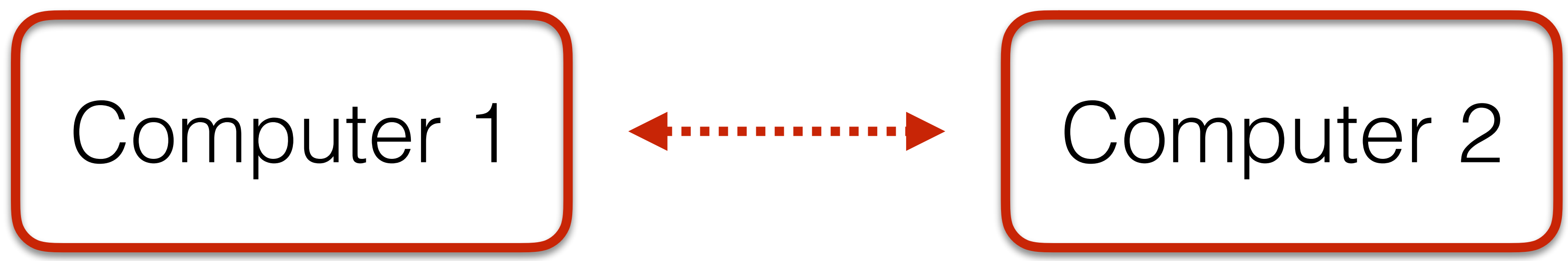
Why Pure Message Passing?

and isolation

Failure

Computer

What happens if the entire computer crashes?



If computer 1 crashes computer 2 takes over

If computer 2 crashes computer 1 takes over

Impossible with shared memory and dangling pointers

Reliability

P(fail) = 10^-3

P(fail)^2 = 10^-6

100 9's reliability with 34 computers

The key is independence

No dangling pointers

No shared memory

No synchronous events

Scalability

Can easily scale horizontally if processes are independent

Solves “massively parallel” problems which are very common

Key is independent isolated computations

One of 6 pre-conditions
read my PhD thesis

Shared memory is evil

Does this work?

Yes	No
WhatsApp, Klarna, Ericsson, …	Nitty gritty in-memory stuff (a few % of all SW)

Erlang is also

- Functional
- Dynamically typed
- Has Immutable values (data)

These are GOOD properties

Values are immutable

X = ~~7~~

↓

9

7 means 7

Immutable values are cacheable

Fault tolerance with immutable values

```
loop(State) ->
   receive
      F ->
        try F(State) of
           error:Why ->
             loop(State);
          NewState ->
            loop(NewState)
    end
```

Immutability Changes Everything

Pat Helland
Salesforce.com
One Market Street, #300
San Francisco, CA 94105 USA
01(415) 546-5881
phelland@salesforce.com

ABSTRACT

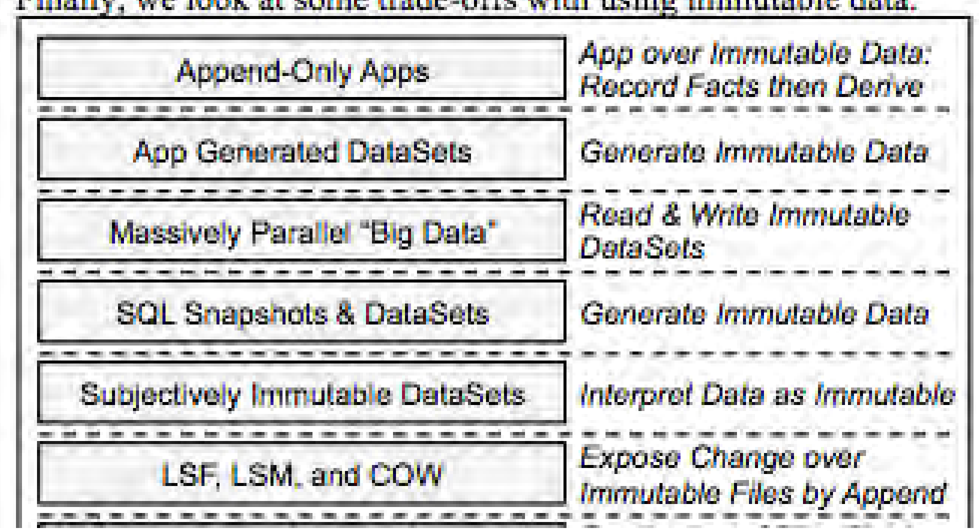
There is an inexorable trend towards storing and sending immutable data. We *need immutability* to coordinate at a distance and we *can afford immutability*, as storage gets cheaper.

This paper is simply an amuse-bouche on the repeated patterns of computing that leverage immutability. Climbing up and down the compute stack really does yield a sense of déjà vu all over again.

1. INTRODUCTION

It wasn't that long ago that computation was expensive, disk storage was expensive, DRAM was expensive, but coordination with latches was cheap. Now, all these have changed using cheap computation (with many-core), cheap commodity disks, and cheap DRAM and SSD, while coordination with latches gets harder because latch latency loses lots of instruction opportunities.

Next, we discuss how the hardware folks have joined the party by leveraging these tricks in SSD and HDD. See Figure 1. Finally, we look at some trade-offs with using immutable data.



If you can't change state you don't need to lock it

The BEAM

Erlang
Prolog Emulator
C Emulator “Jam”
Improved C emulator “Beam”
Native code “Hype”
JIT (work in progress)
Erlang on Xen (super elasticity)
“Beam langages” (Elixir, LFE, …)

Inside the Beam

Create a process
Send a messages
Fast context switch
Small processes
One stack+heap per processes
No shared memory
Only pure copying message passing

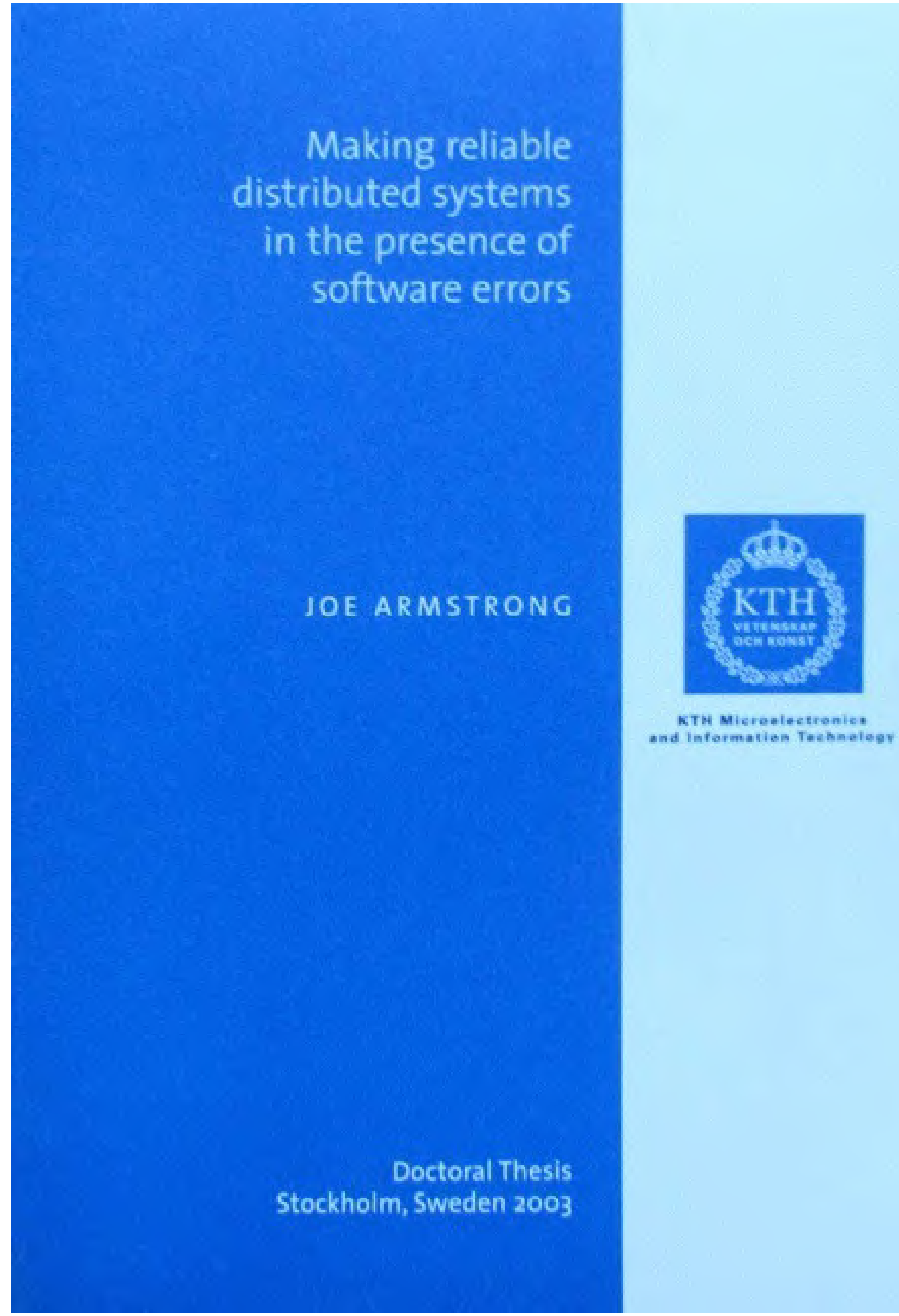
ERICSSON

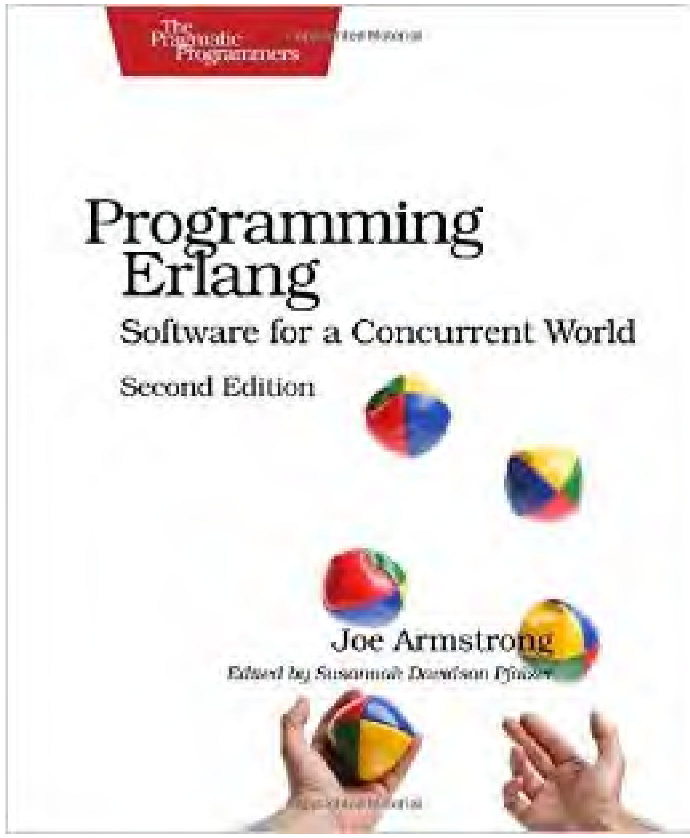
200+ man years of work

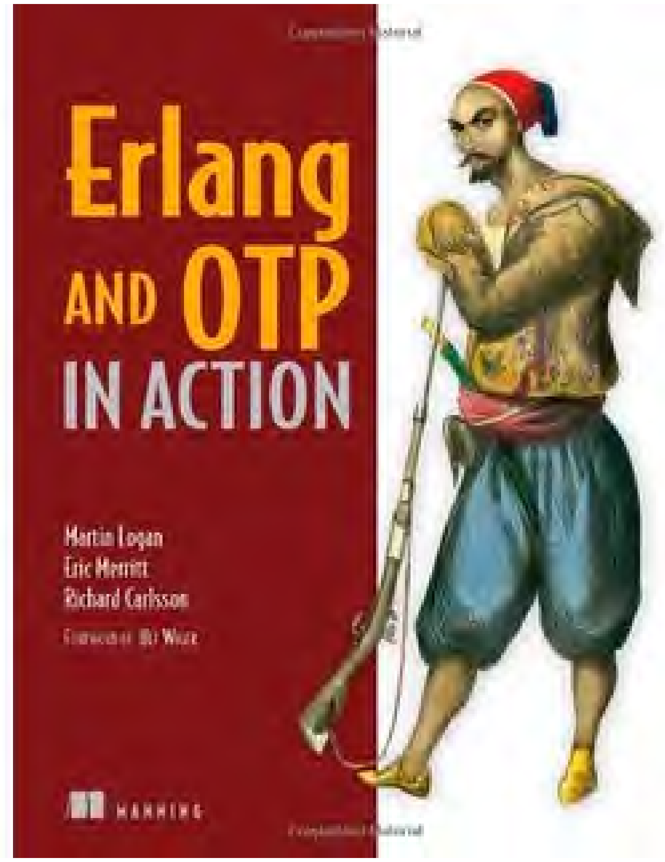
Learning

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

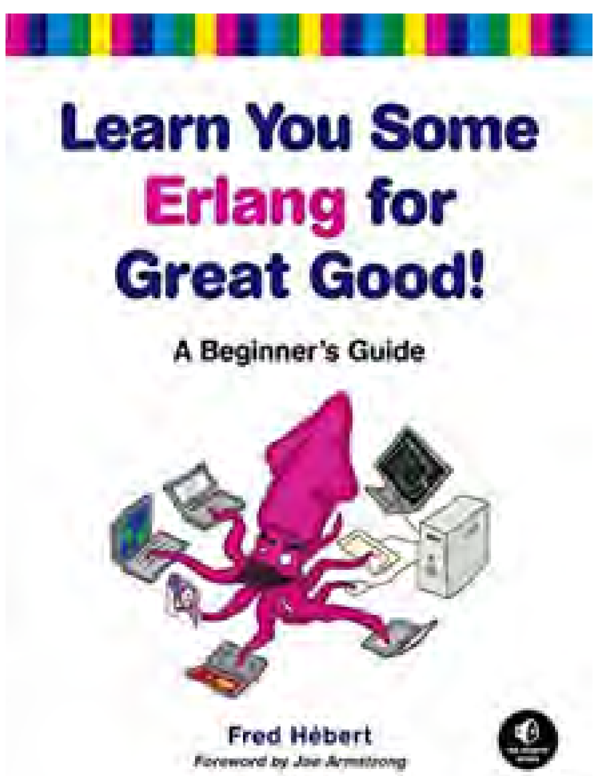
A: Yes



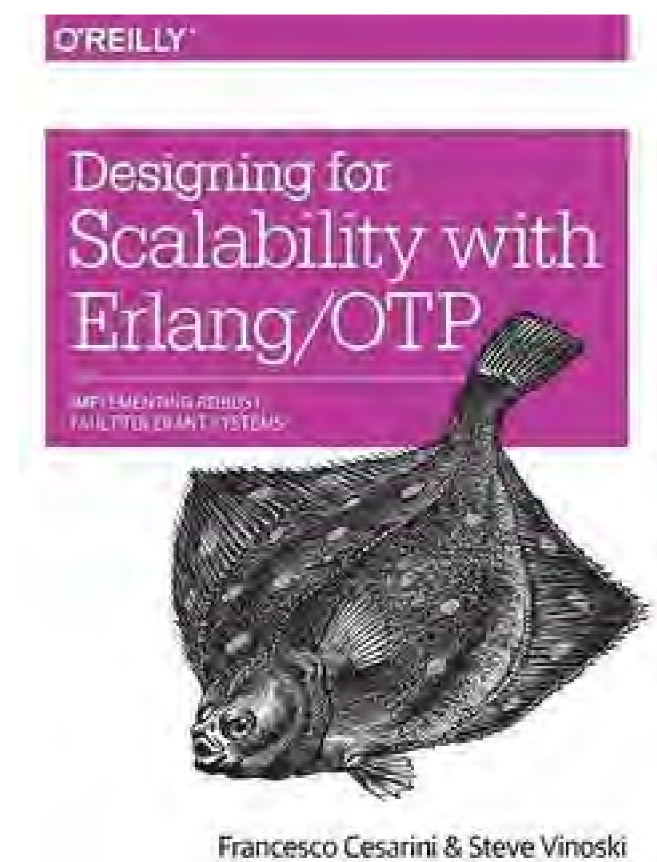

The Pragmatic Programmers
Programming Erlang
Software for a Concurrent World
Second Edition
Joe Armstrong
Edited by Susannah Davidson Pfalzer


Erlang AND OTP IN ACTION
Martin Logan
Eric Merritt
Richard Carlsson
MANNING


A Concurrent Approach to Software Development
Erlang
Programming
O'REILLY
Francesco Cesarini & Simon Thompson

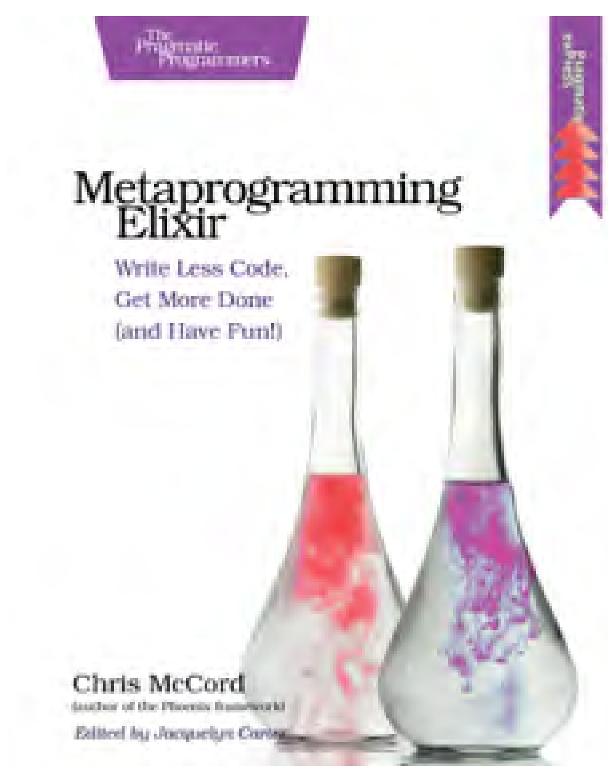

Learn You Some Erlang for Great Good!
A Beginner's Guide
Fred Hébert
Foreword by Joe Armstrong


FRED HÉBERT
STUFF GOES BAD: ERLANG IN ANGER


O'REILLY
Designing for Scalability with Erlang/OTP
Francesco Cesarini & Steve Vinoski


The Pragmatic Programmers
Programming Elixir
Functional
|> Concurrent
|> Pragmatic
|> Fun
Dave Thomas
Foreword by
José Valim,
Creator of Elixir
edited by Lynn Beighley


Elixir
IN ACTION
Saša Jurić
MANNING


The Pragmatic Programmers
Metaprogramming Elixir
Write Less Code,
Get More Done
(and Have Fun!)
Chris McCord
(author of the Phoenix framework)
Edited by Jacquelyn Carter


The Pragmatic Programmers
Your Elixir Source
Programming Phoenix
Productive |> Reliable |> Fast
TM
Chris McCord,
Bruce Tate,
and José Valim
edited by Jacquelyn Carter

Thank you

have a fun conference