

Error Handling in Reactive Systems

Dean Wampler, Ph.D.
Typesafe



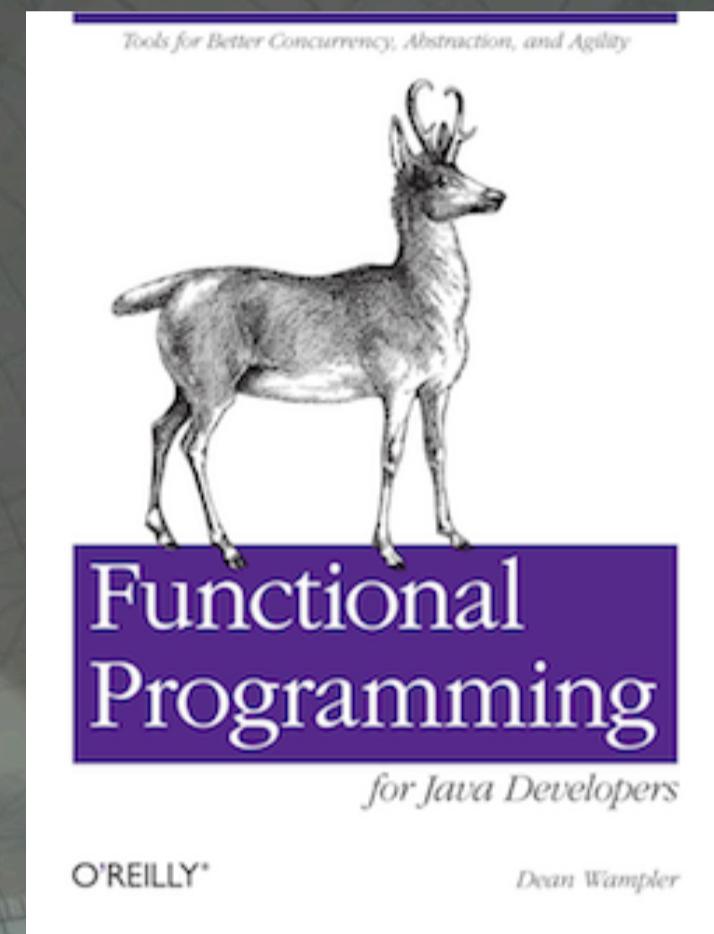
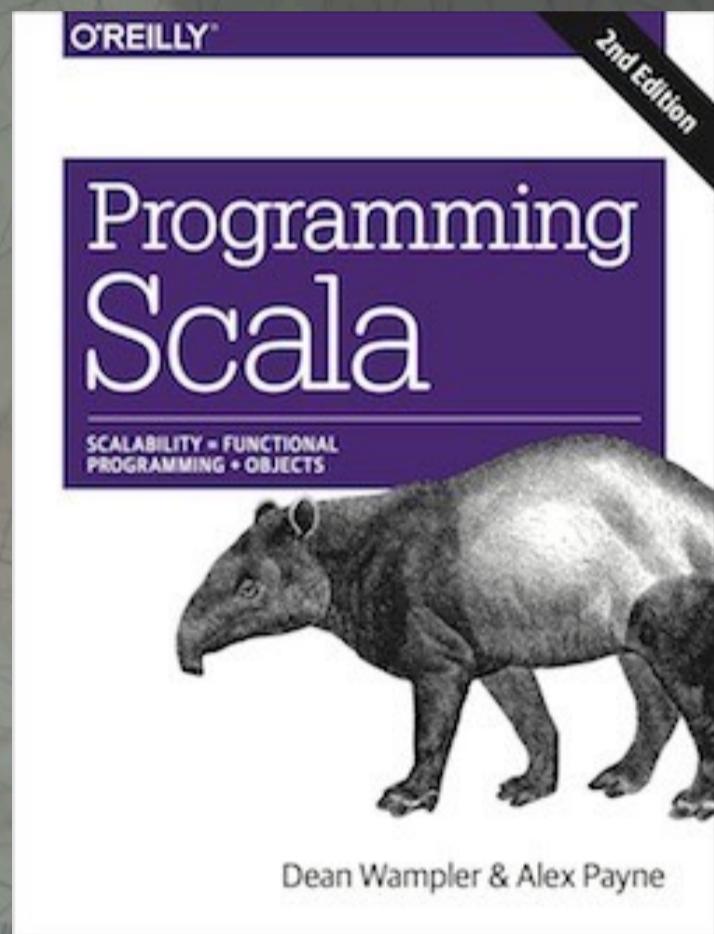
©Typesafe 2014-2015, All Rights Reserved

Wednesday, March 18, 15

Photos from Jantar Mantar (“instrument”, “calculation”), the astronomical observatory built in Jaipur, India, by Sawai Jai Singh, a Rajput King, in the 1720s–30s. He built four others around India. This is the largest and best preserved. All photos are copyright (C) 2000–2015, Dean Wampler. All Rights Reserved.



dean.wampler@typesafe.com
polyglotprogramming.com/talks
@deanwampler

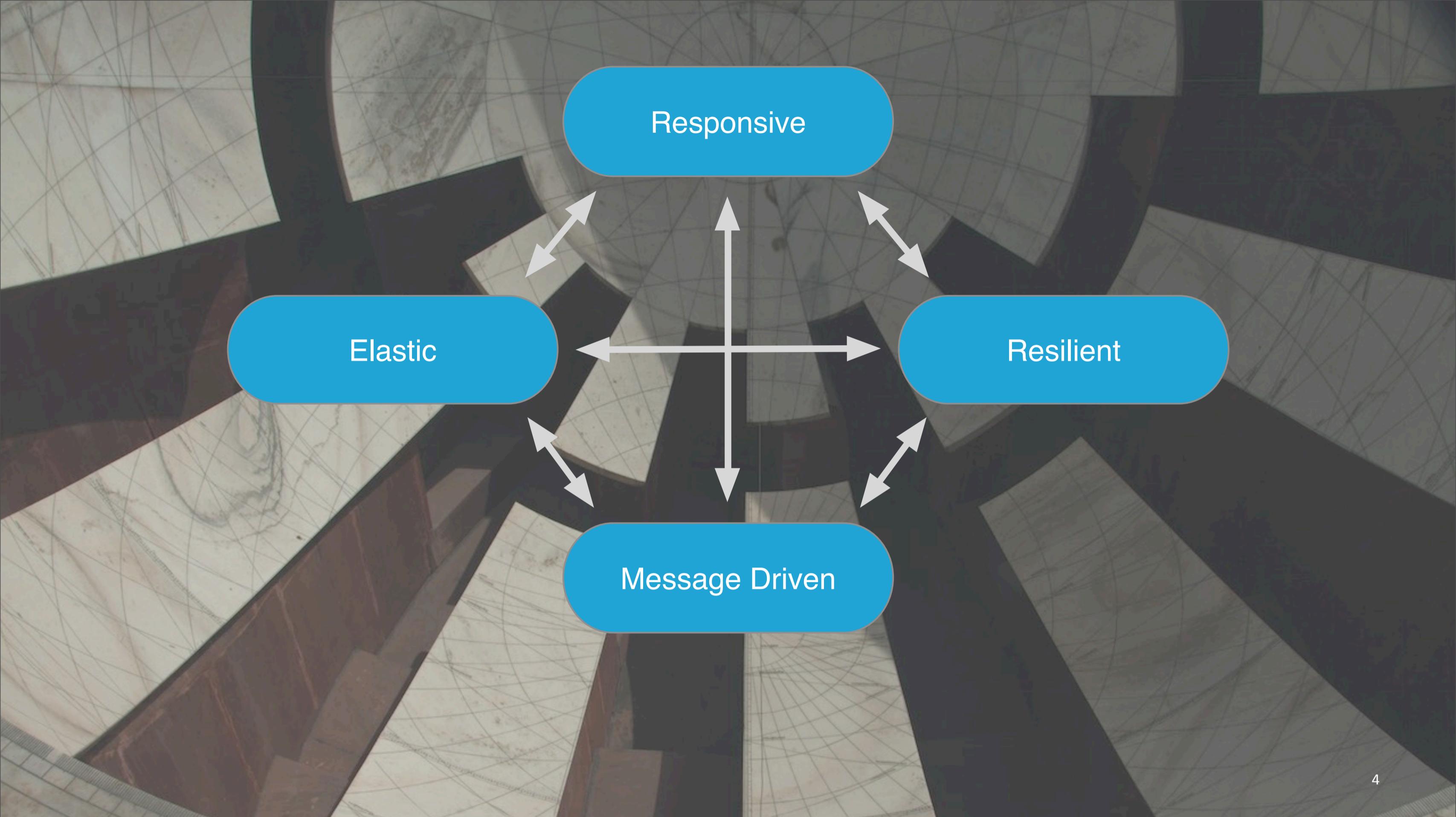


Typesafe Reactive Big Data

typesafe.com/reactive-big-data

Wednesday, March 18, 15

This is my role. We're just getting started, but talk to me if you're interested in what we're doing.

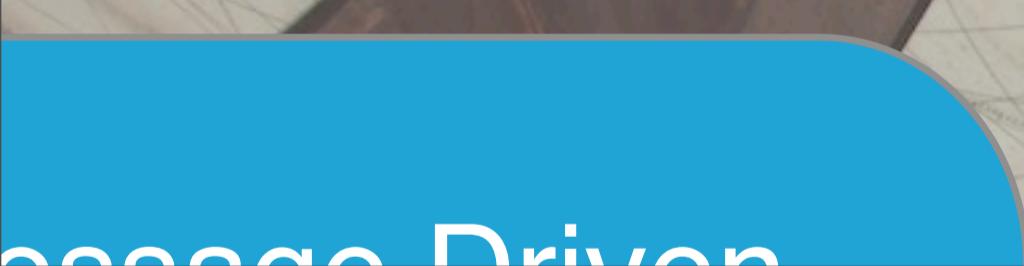




Failures are
first class?



Resilient

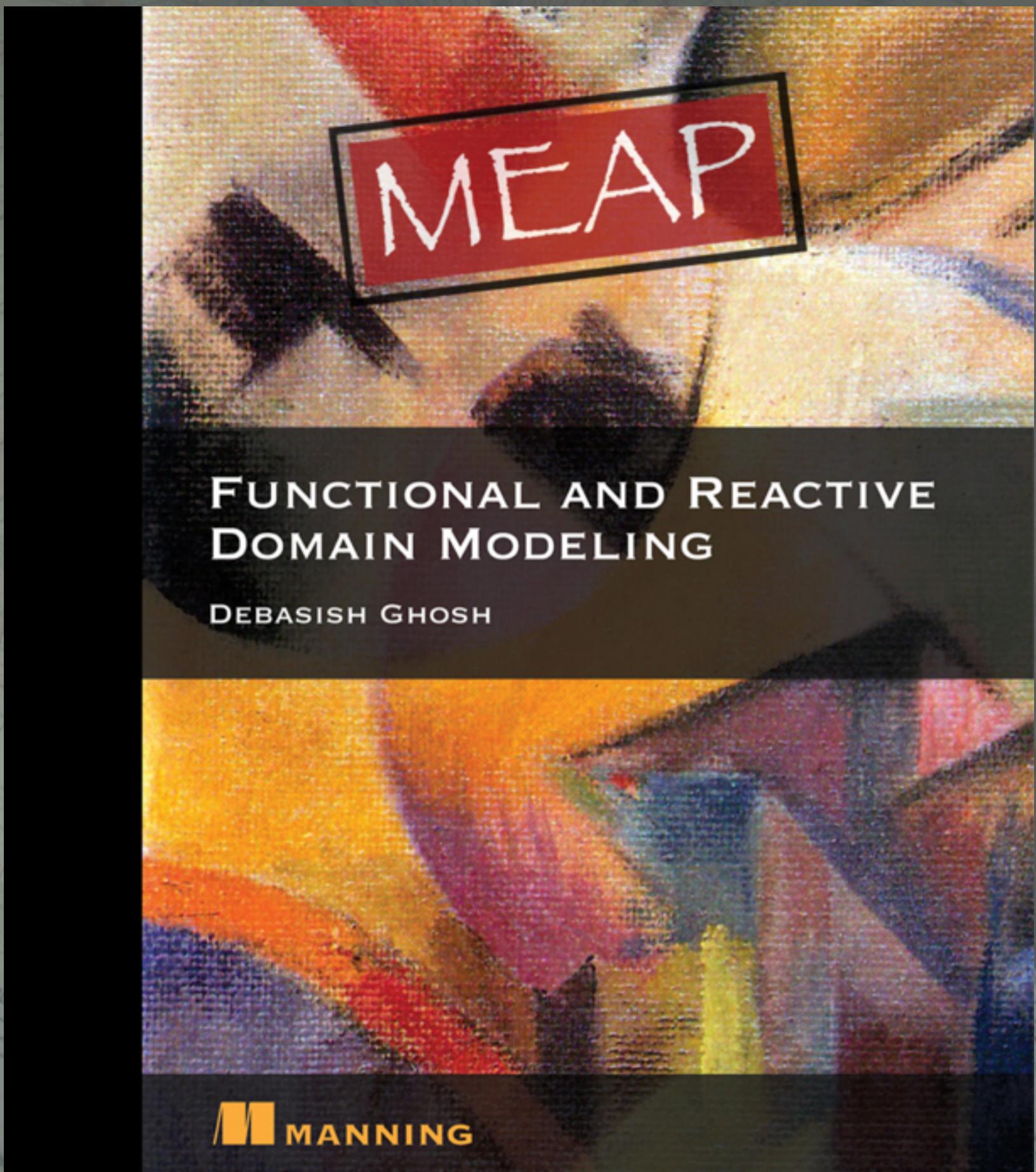


Resilience Driven

5

Wednesday, March 18, 15

Truly resilient systems must make failures first class citizens, in some sense of the word, because they are inevitable when the systems are big enough and run long enough.



6

Wednesday, March 18, 15

I've structured parts of this talk around points made in Debasish's new book, which has lots of interesting practical ideas for combining functional programming and reactive approaches with classic Domain-Driven Design by Eric Evans.

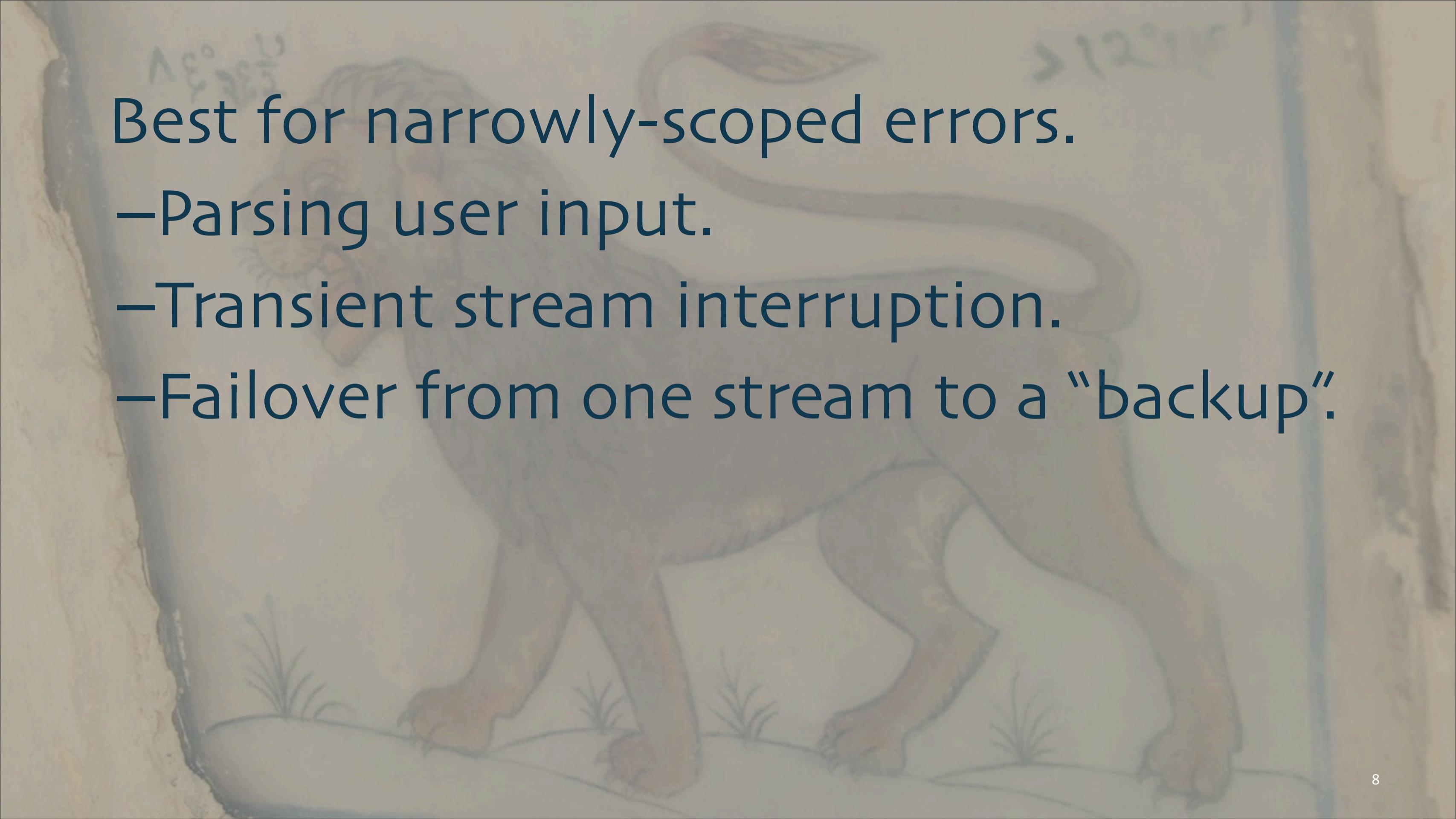


#1 Failure-handling mixed with domain logic.

7

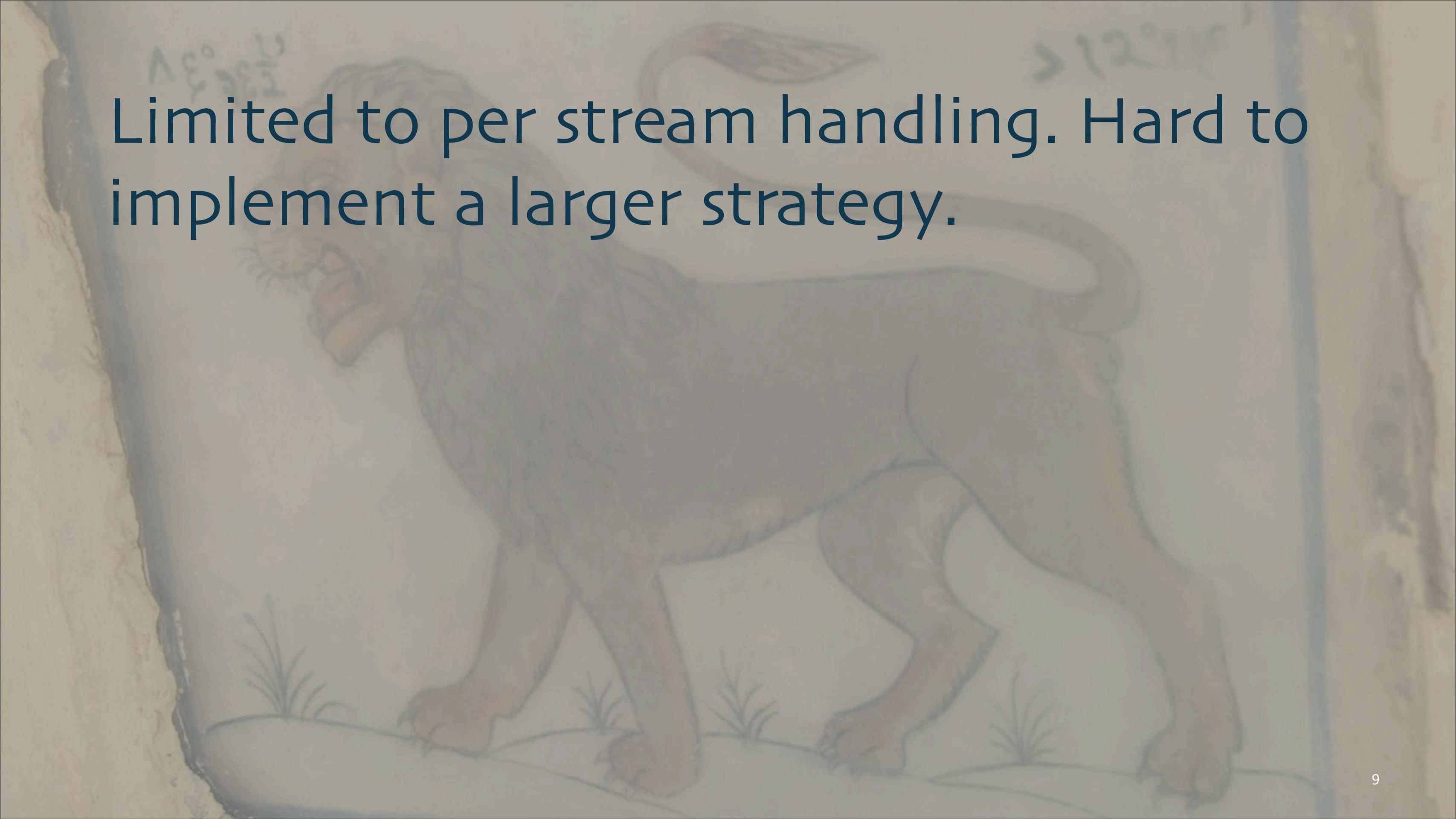
Wednesday, March 18, 15

This is how we've always done it, right?



Best for narrowly-scoped errors.

- Parsing user input.
- Transient stream interruption.
- Failover from one stream to a “backup”.



Limited to per stream handling. Hard to implement a larger strategy.

Communicating Sequential Processes

Message passing
via channels



Wednesday, March 18, 15

See

http://en.wikipedia.org/wiki/Communicating_sequential_processes

<http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html>

<http://blog.drewolson.org/blog/2013/07/04/clojure-core-dot-async-and-go-a-code-comparison/>

and other references in the “bonus” slides at the end of the deck. I also have some slides that describe the core primitives of CSP that I won’t have time to cover.



“Don’t communicate
by sharing memory,
share memory
by communicating”

-- Rob Pike

Wednesday, March 18, 15

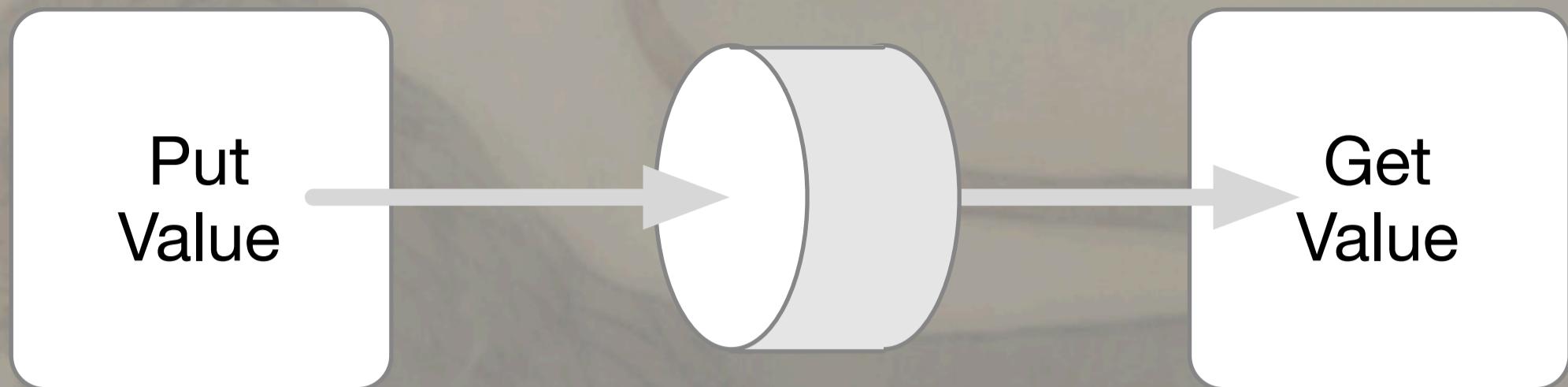
<http://www.youtube.com/watch?v=f6kdp27YZs&feature=youtu.be>

From a talk Pike did at Google I/O 2012.

CSP: inspired Go & Clojure's core.async



Wednesday, March 18, 15



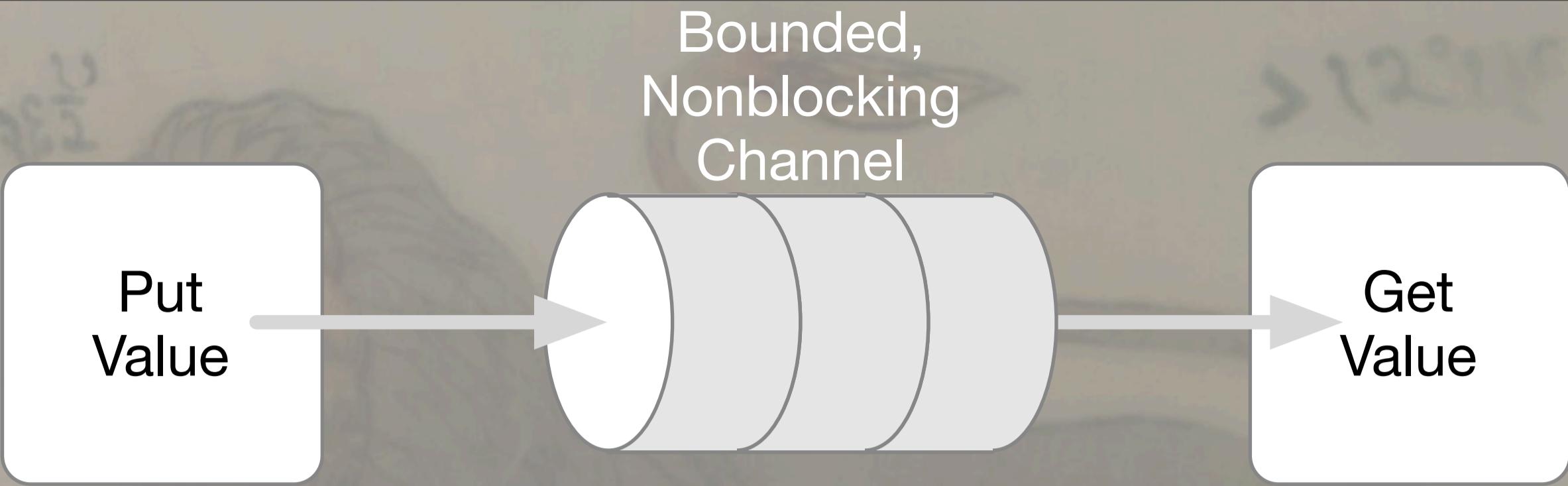
- Block on put if no one to get.
- Channel can be typed.
- Avoid passing mutable state!

Wednesday, March 18, 15

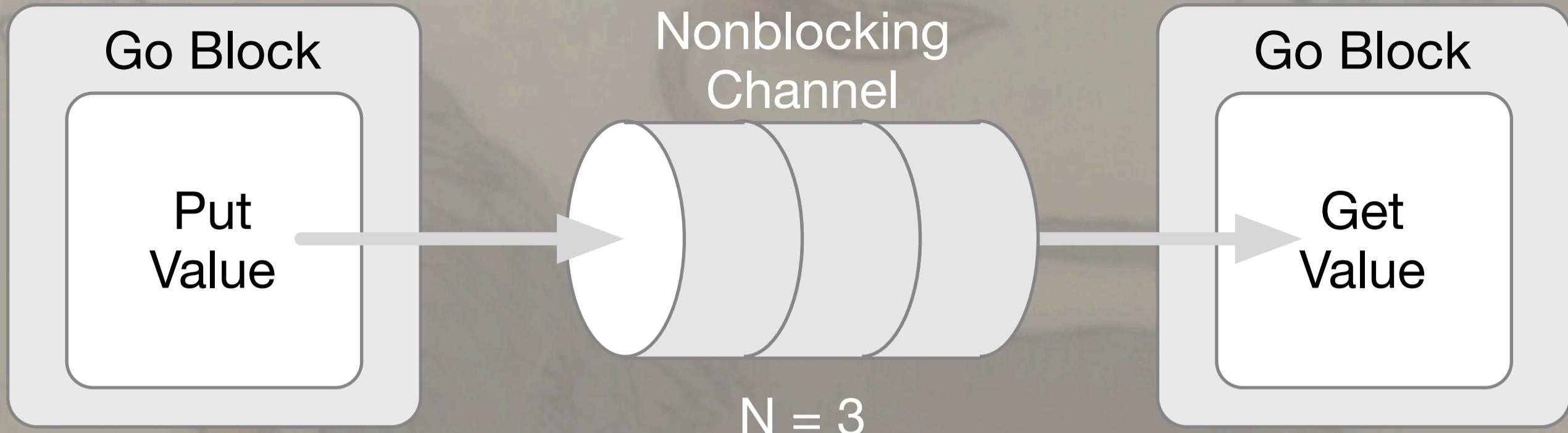
Simplest channel, a blocking, 1-element “connector” used to share values, one at a time between a source and a waiting sync. The put operation blocks if there is no sync waiting on the other end.

The channel can be typed (Go lang).

Doesn’t prevent the usual problems if mutable state is passed over a channel!



- When full:
 - Block on put.
 - Drop newest (i.e., put value).
 - Drop oldest ("sliding" window).



- Core Async: Go Blocks, Threads.
- Go: Go Routines.
- Analogous to futures.

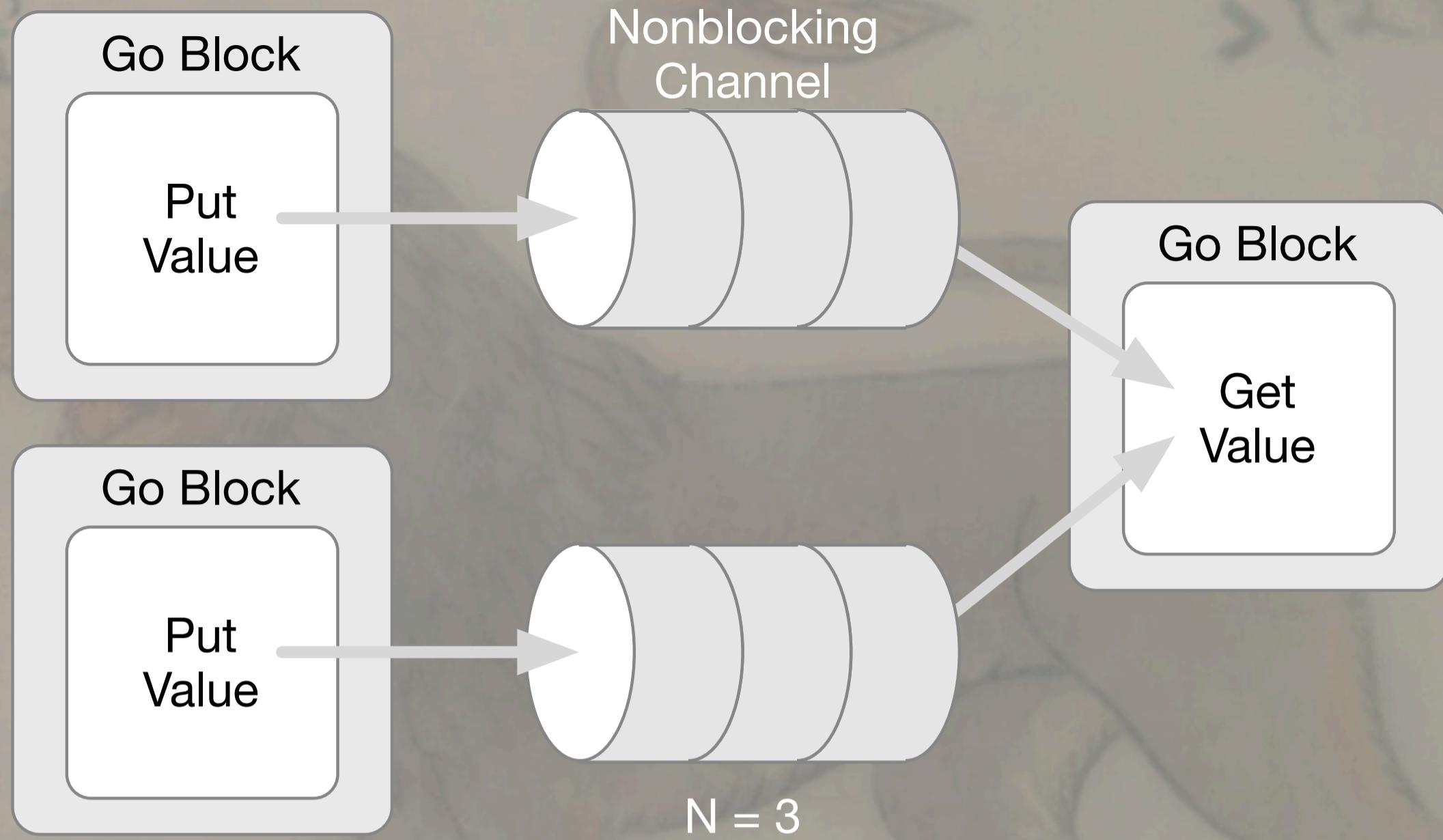
15

Wednesday, March 18, 15

So far, we haven't supported any actual concurrency. I'm using "Go Blocks" here to represent explicit threads in Clojure, when running on the JVM and you're willing to dedicate a thread to the sequence of code, or core async "go blocks", which provide thread-like async behavior, but share real threads. This is the only option for clojure.js, since you only have one thread period.

Similarly for Go, "go blocks" would be "go routines".

In all cases, they are analogous to Java/Scala futures.



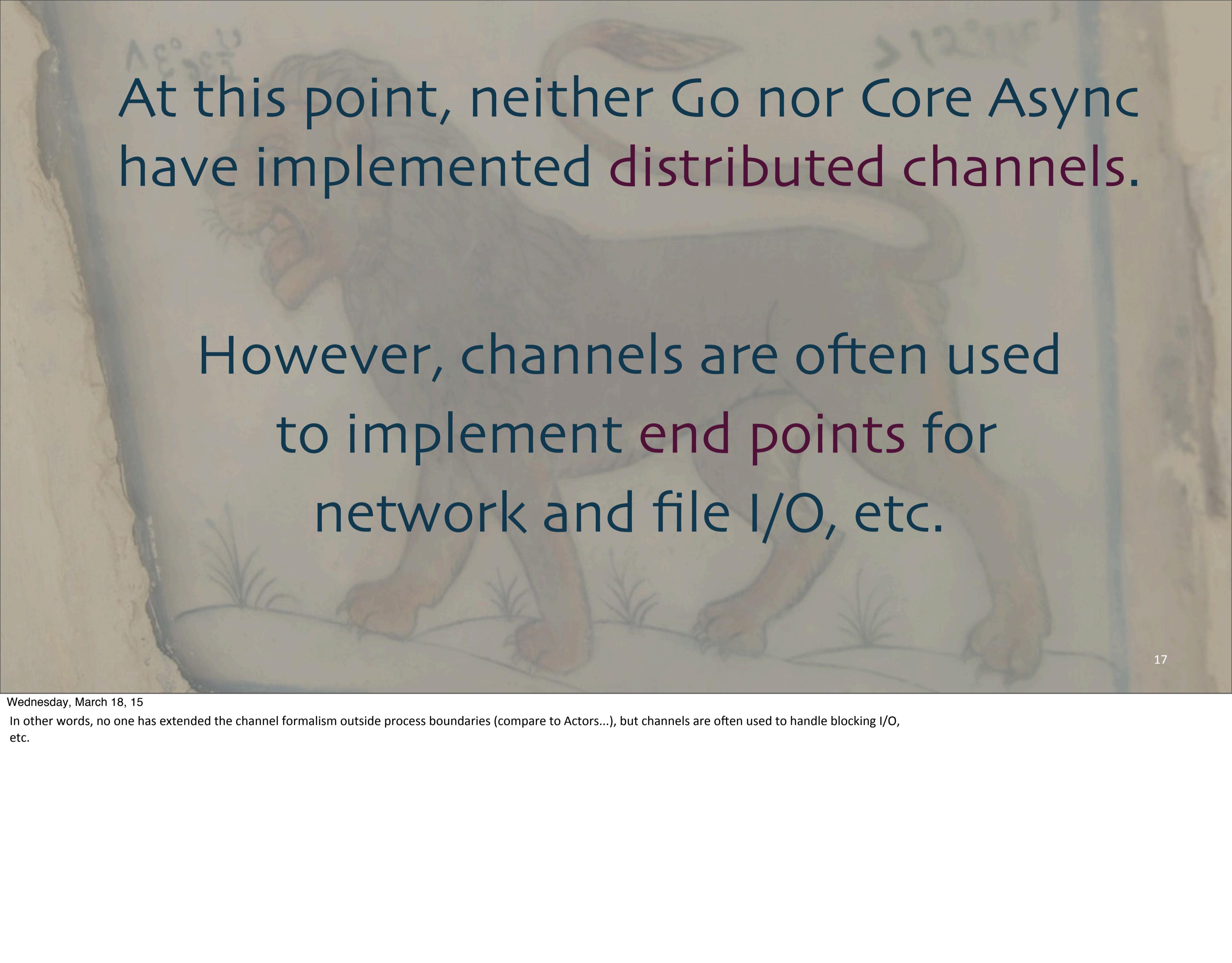
- Blocking or nonblocking.
- Like socket select.

16

Wednesday, March 18, 15

You can "select" on several channels, analogous to socket select. I.e., read from the next channel with a value. In go, there is a "select" construct for this. In core async, there are the "alt!" (blocking) and "alt!!" (nonblocking) functions.

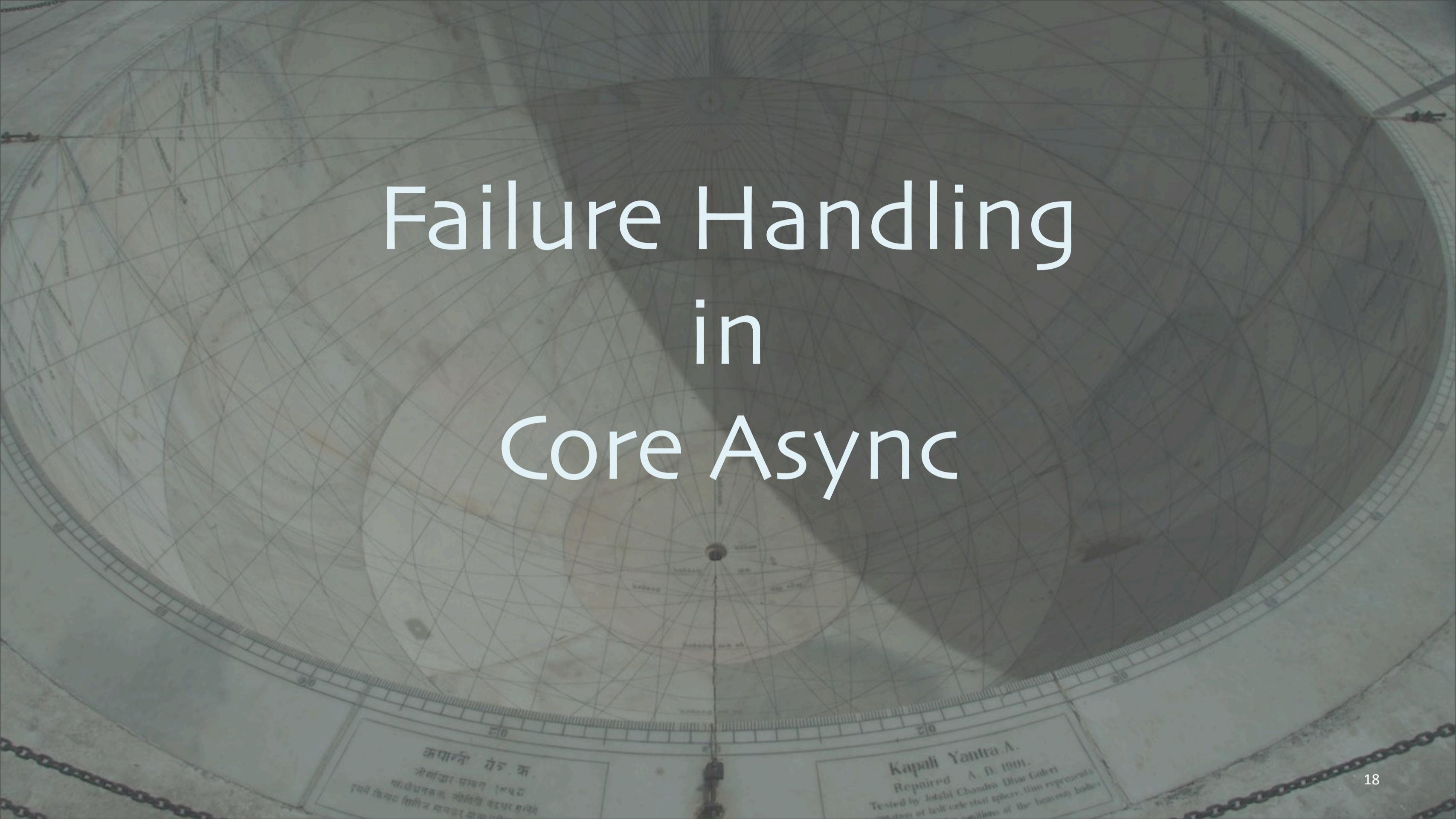
Fan out is also possible.



At this point, neither Go nor Core Async have implemented distributed channels.

However, channels are often used to implement end points for network and file I/O, etc.

Failure Handling in Core Async



Wednesday, March 18, 15

The situation is broadly similar for Go.
Some items here are adapted from a private conversation with Alex Miller (@puredanger).

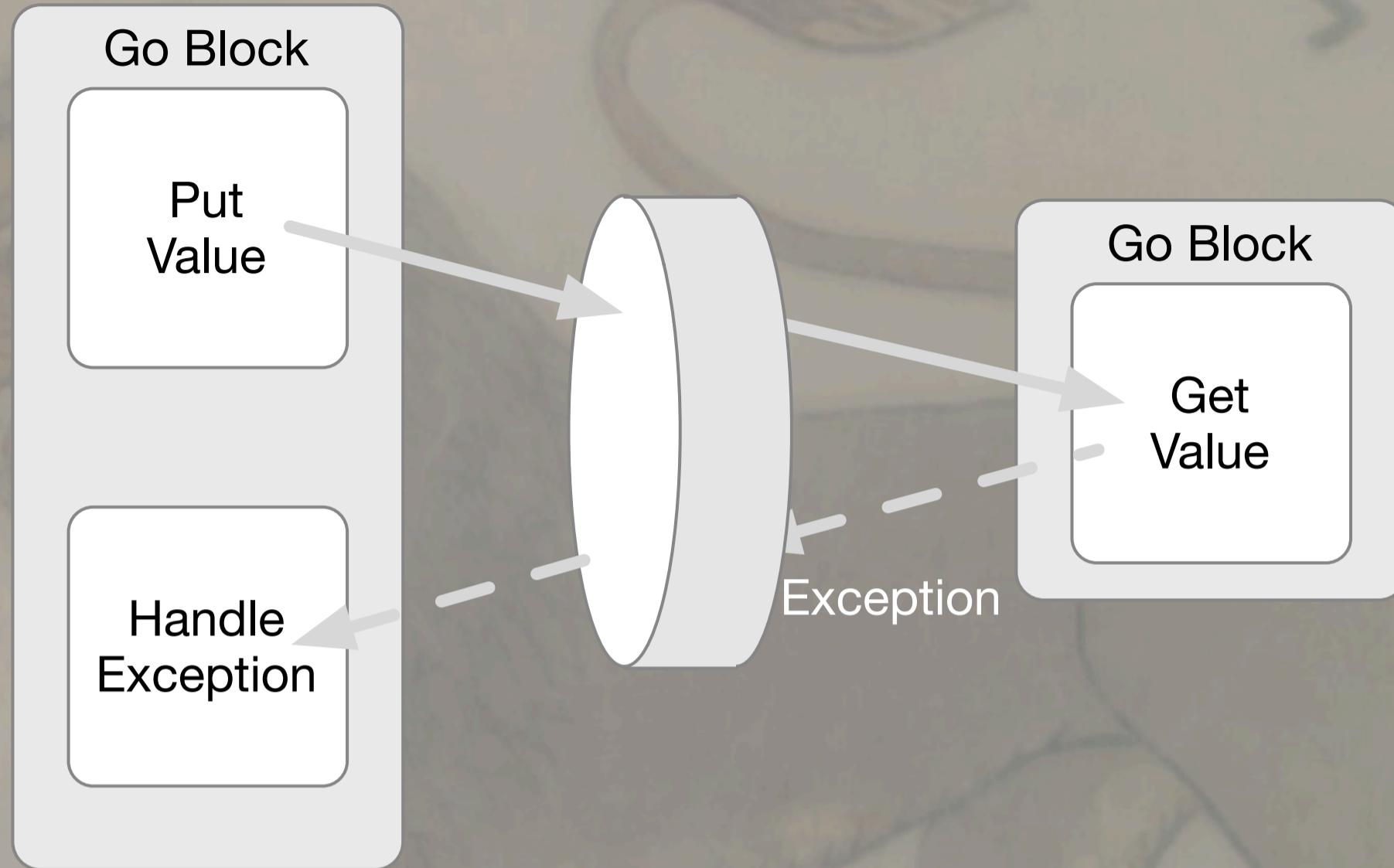
Channel construction takes an optional exception function.

- The exception is passed to the function.
- If it returns non-nil, that value is put on the channel.

Which call stack?

- Processing logic can span several threads!
- A general problem for concurrency implemented using multithreading.

Time



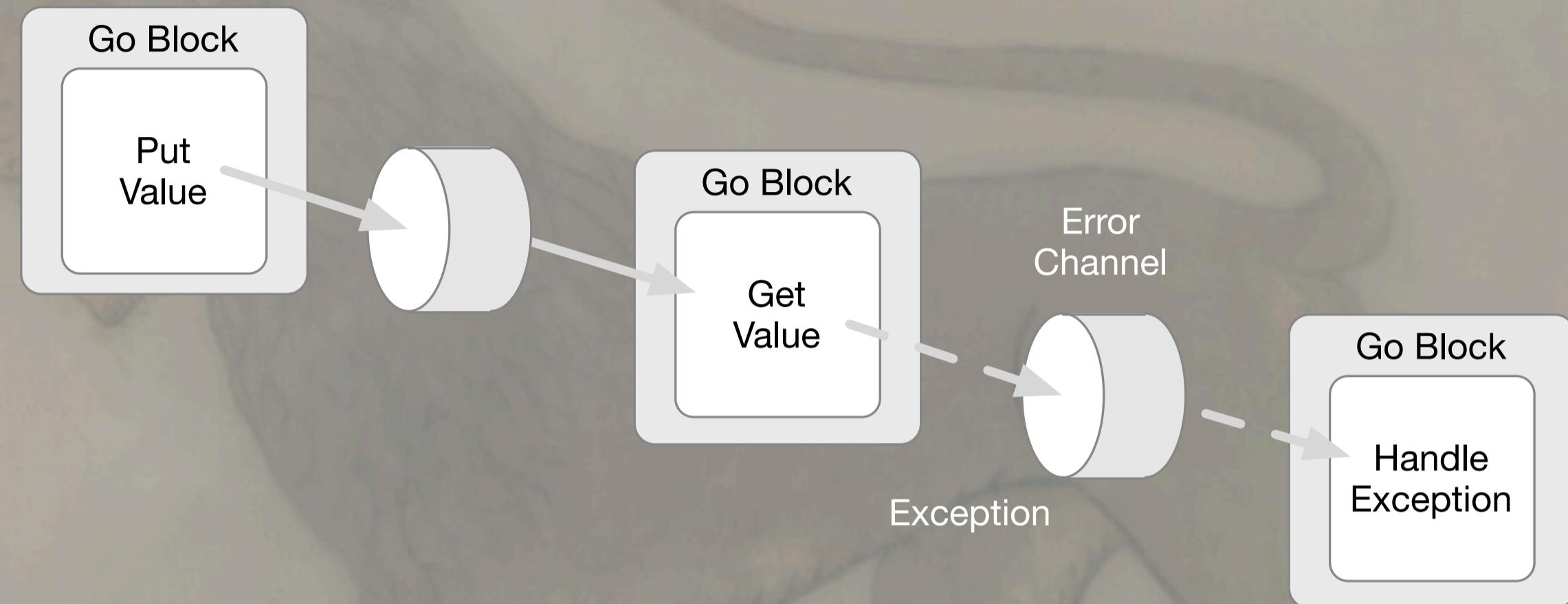
Propagate exceptions back through the channel.

21

Wednesday, March 18, 15

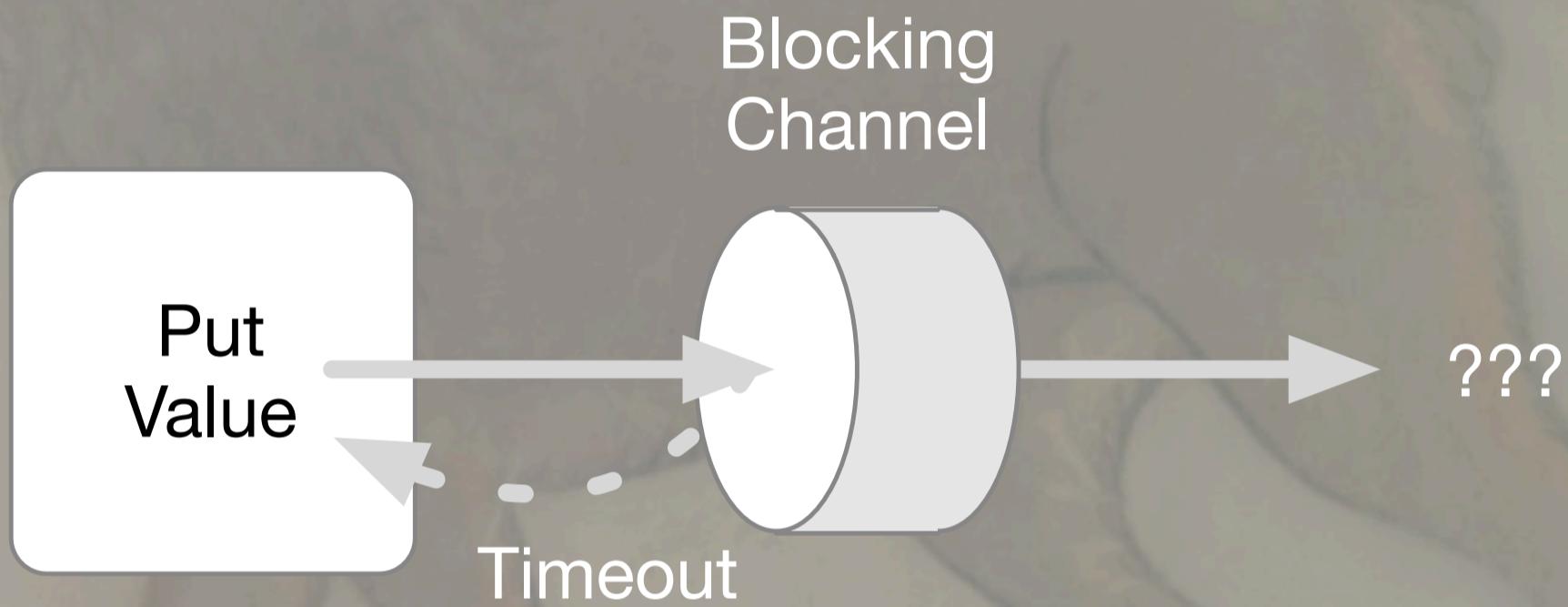
One possible scheme is to push exceptions back through the channel and let the initializing go block decide what to do. It might rethrow the exception.

Time



Propagate exceptions to a special error channel.

Deadlock is possible unless timeouts are used.



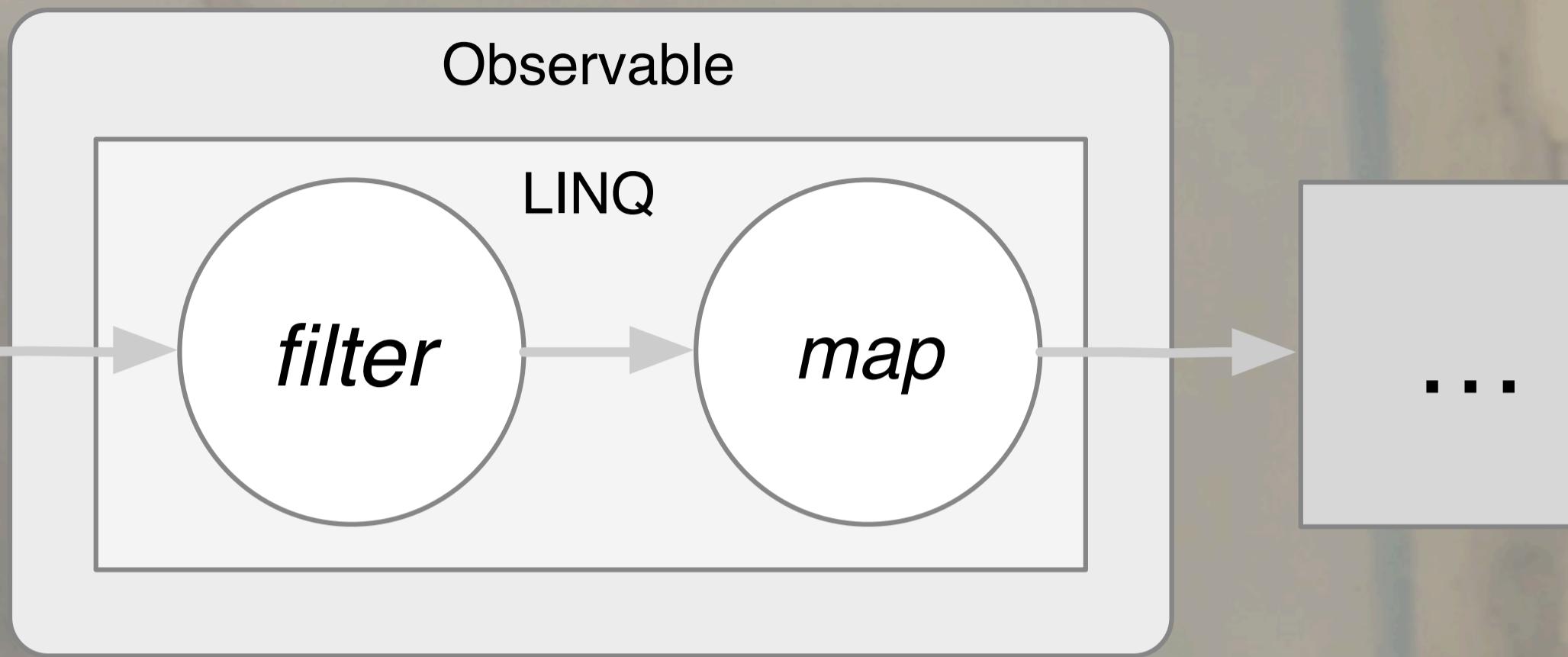
Reactive Extensions



Wednesday, March 18, 15

I'll look at C# examples, but all the Rx language implementations work similarly.

Async
Event Stream



Schedulers



Uses classic patterns
for exception handling,
with extensions.

OnError notification caught with a Catch method.

- Switch to a second stream.

```
val stream = new Subject<MyType>();  
val observer = stream.Catch(otherStream);  
...  
stream.OnNext(item1);  
...  
stream.OnError(new UnhappyException("error"));  
// continue with otherStream.
```

Variant for catching a specific exception, with a function to construct a new stream.

```
val stream = new Subject<MyType>();  
val observer = stream.Catch<MyType, MyException>(  
    ex => /* create new MyType stream */);  
...  
stream.OnNext(item1);  
...  
stream.OnError(new MyException("error"));  
// continue with generated stream.
```

There is also a Finally method.

Analogous to

try {...} finally {...}

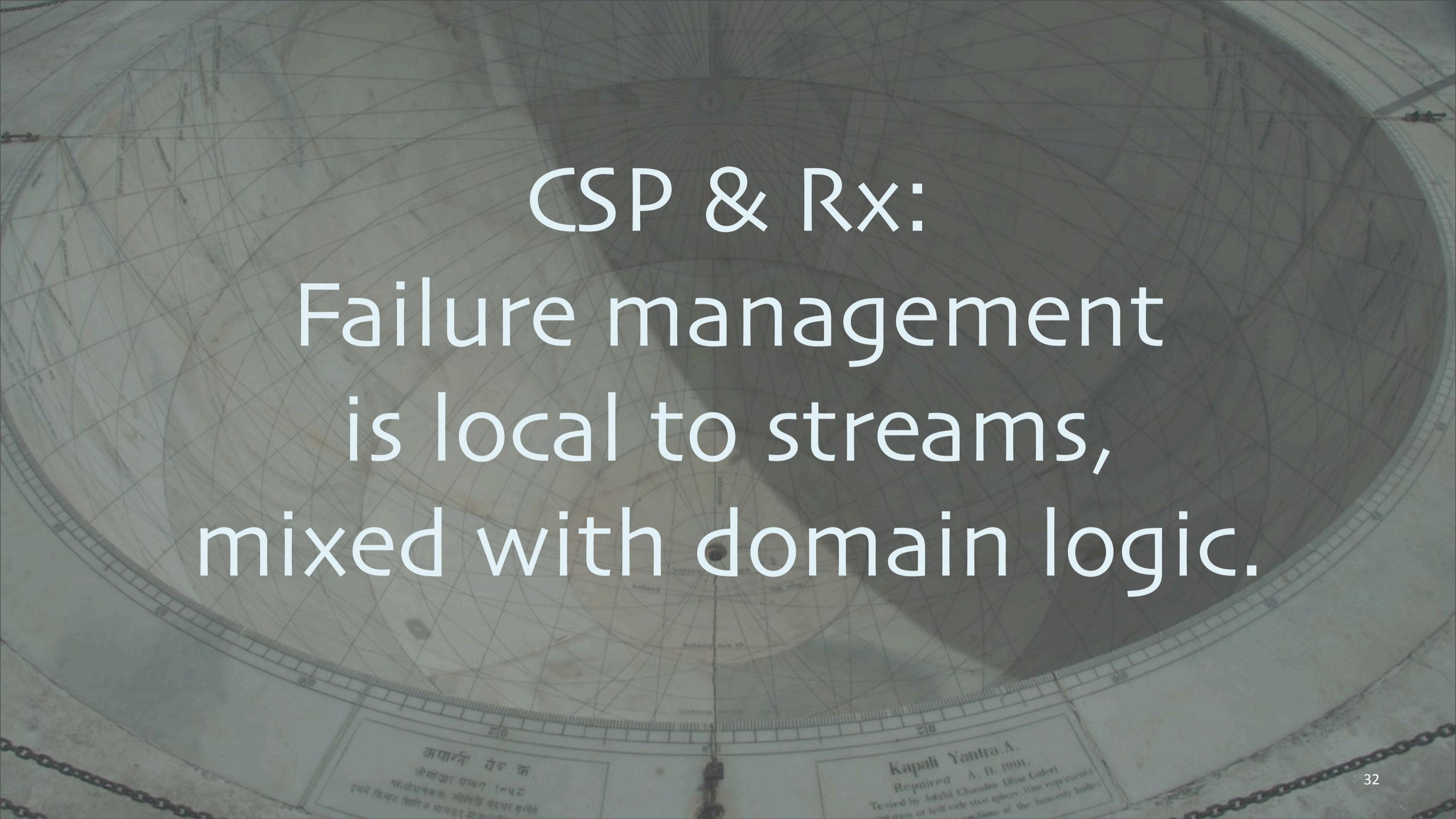
clauses.

OnErrorResumeNext: Swallows exception, continues with alternative stream(s).

```
public static IObservable<TSource> OnErrorResumeNext<TSource>(  
    this IObservable<TSource> first,  
    IObservable<TSource> second) {...}  
  
public static IObservable<TSource> OnErrorResumeNext<TSource>(  
    params IObservable<TSource>[] sources) {...}  
...
```

Retry: Are some exceptions expected, e.g., I/O “hiccups”. Keeps trying. Optional max retries.

```
public static void RetrySample<T>(  
    IObservable<T> source)  
{  
    source.Retry(4) // retry up to 4 times.  
        .Subscribe(t => Console.WriteLine(t));  
    Console.ReadKey();  
}
```



CSP & Rx: Failure management is local to streams, mixed with domain logic.

Wednesday, March 18, 15

What CSP-derived and Rx concurrency systems do, they do well, but we need a larger strategy for reactive resiliency at scale.

Before we consider such strategies, let's discuss another technique.



#2 Prevent common problems.

33

Wednesday, March 18, 15

This is how we've always done it, right?



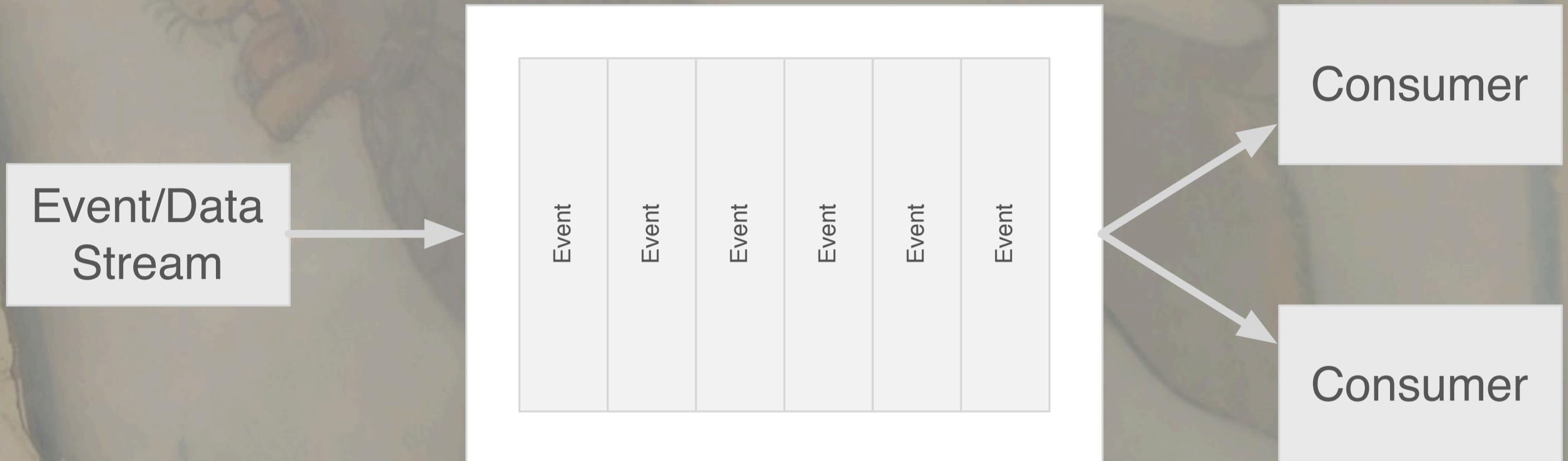
Reactive Streams

Wednesday, March 18, 15

Reactive Streams extend the capabilities of CSP channels and Rx by addressing flow control concerns.

Reactive Streams

Bounded or Unbounded Queue?



Streams (data flows) are a natural model for many distributed problems, i.e., one-way CSP channels at scale.

35

Wednesday, March 18, 15

You want a queue in the middle of producer and consumer to buffer events and enable asynchrony, but should that queue be bounded or unbounded? If unbounded, eventually, it will grow to exhaust memory. If bounded, what should happen when it's full? Should the producer just drop messages, block, crash...?

Reactive Streams

Bounded Queue



<http://www.reactive-streams.org/>

36

Wednesday, March 18, 15

The key element of reactive streams (over any others...) is the notion of back pressure, where the producer and consumer coordinate on the rate of event delivery.

Back pressure:

- No OoM errors (unbounded queue).
- No arbitrary dropped events or blocking (bounded).
- You decide when and where to drop events or do something else.
 - Enables strategic flow control.

Back pressure:

- Is it push or pull?
- Both - push most of the time, pull when flow control between producer & consumer is necessary.

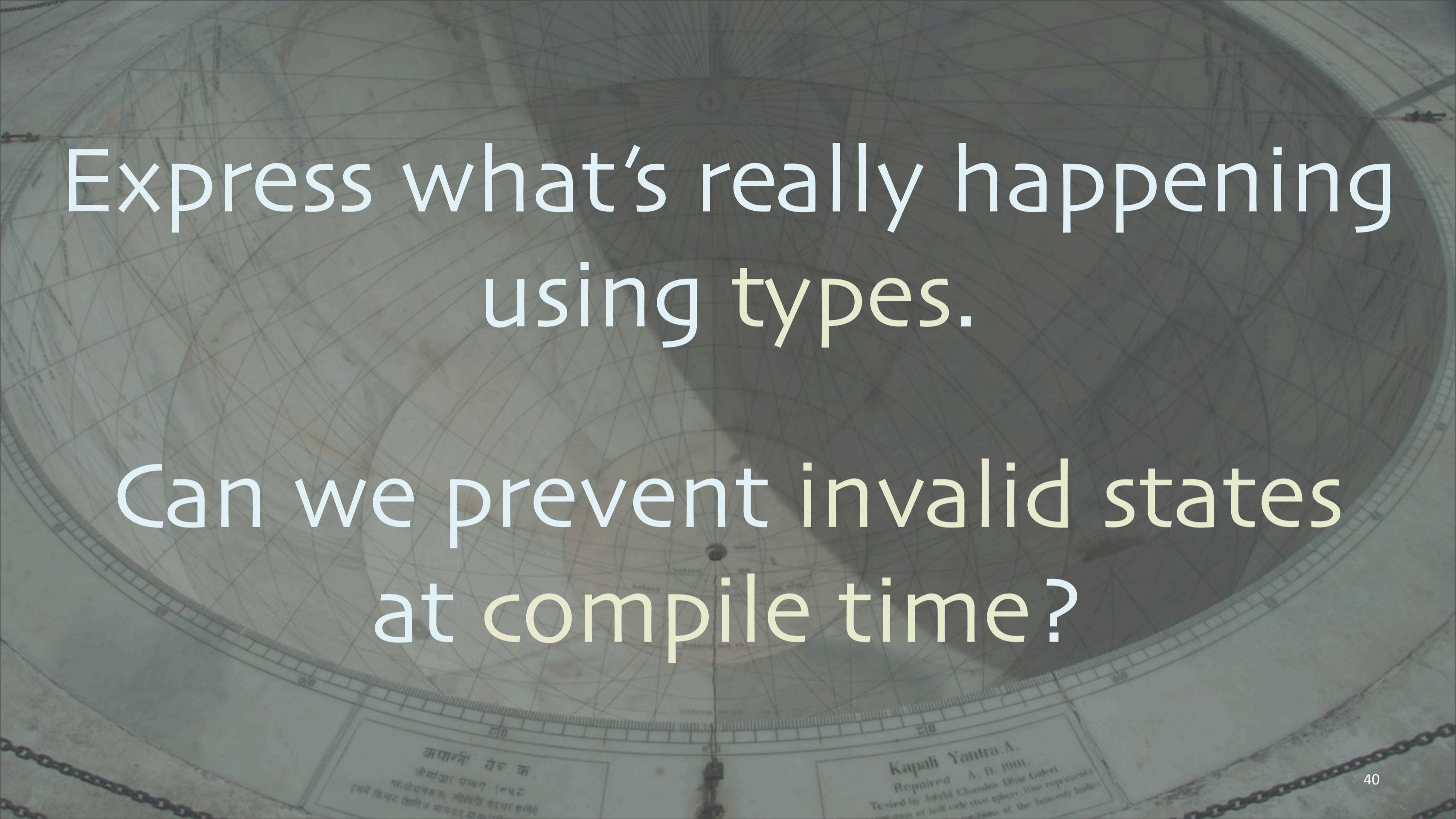


#3 Leverage types to prevent errors.

39

Wednesday, March 18, 15

This is how we've always done it, right?



Express what's really happening
using types.

Can we prevent invalid states
at compile time?

Wednesday, March 18, 15

First, let's at least be honest with the reader about what's actually happening in blocks of code.

When code raises exceptions:

```
case class Order(  
  id: Long, cost: Money, items: Seq[(Int,SKU)])  
  
object Order {  
  def parse(string: String): Try[Order] = Try {  
    val array = string.split("\t")  
    if (bad(array)) throw new ParseError(string)  
    new Order(...)  
  }  
  private def bad(array: Array[String]): Boolean = {...}  
}
```

41

Wednesday, March 18, 15

Idiomatic Scala for “defensive” parsing of incoming data as strings. Wrap the parsing and construction logic in a Try {...}. Note the capital T; this will construct a Try instance, either a subclass Success, if everything works, or a Failure, if an exception is thrown.

See the github repo for this presentation for a complete example: <https://github.com/deanwampler/Presentations>

Latency? Use Futures

- Or equivalents, like go blocks.

```
case class Account(  
  id: Long, orderIds: Seq[Long])  
...  
  
def getAccount(id: Long): Future[Account] =  
  Future { /* Web service, DB query, etc... */ }  
  
def getOrders(ids: Seq[Long]): Future[Seq[Order]] =  
  Future { /* Web service, DB query, etc... */ }  
...
```

Latency? Use Futures

- Or equivalents, like go blocks.

```
...  
def ordersForAccount(accountId: Long): Future[Seq[Order]] =  
  for {  
    account <- getAccount(accountId)  
    orders  <- getOrders(account.orderIds)  
  } yield orders.toVector
```

Wednesday, March 18, 15

Futures can be sequenced “monadically”, so our code has a nice synchronous feel to it, but we can exploit async. execution. “yield” specifies what’s returned, which will actually be wrapped in another Future by the for comprehension. We convert orders to a Vector (a kind of Seq), which is a very efficient data structure in Scala.

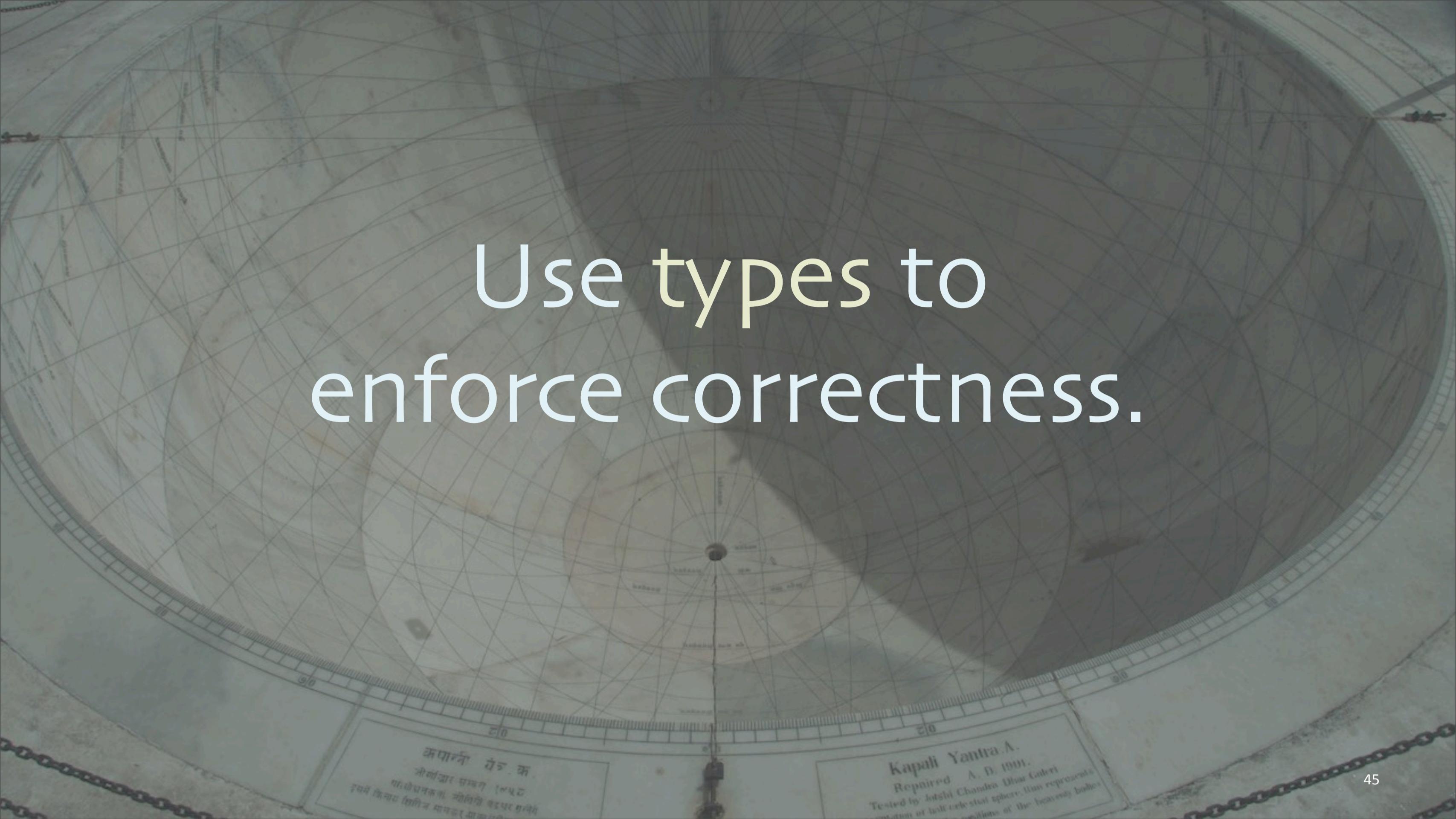
See the github repo for this presentation for a complete example: <https://github.com/deanwampler/Presentations>

Latency? Use Futures

- Or equivalents, like go blocks.

```
val accountId = ...  
val ordersFuture = ordersForAccount(accountId)
```

```
ordersFuture.onSuccess {  
    case orders =>  
        println(s"#$accountId: $orders")  
}  
ordersFuture.onFailure {  
    case exception => println(s"#$accountId: " +  
        "Failed to process orders: $exception")  
}
```



Use types to
enforce correctness.

Wednesday, March 18, 15

First, let's at least be honest with the reader about what's actually happening in blocks of code.



Functional Reactive Programming

Wednesday, March 18, 15

On the subject of type safety, let's briefly discuss FRP. It was invented in the Haskell community, where there's a strong commitment to type safety as a tool for correctness.

Represent evolving state by time-varying values.

```
Reactor.flow { reactor =>
  val path = new Path(
    (reactor.await(mouseDown)).position)
  reactor.loopUntil(mouseUp) {
    val m = reactor.awaitNext(mouseMove)
    path.lineTo(m.position)
    draw(path)
  }
  path.close()
  draw(path)
}
```

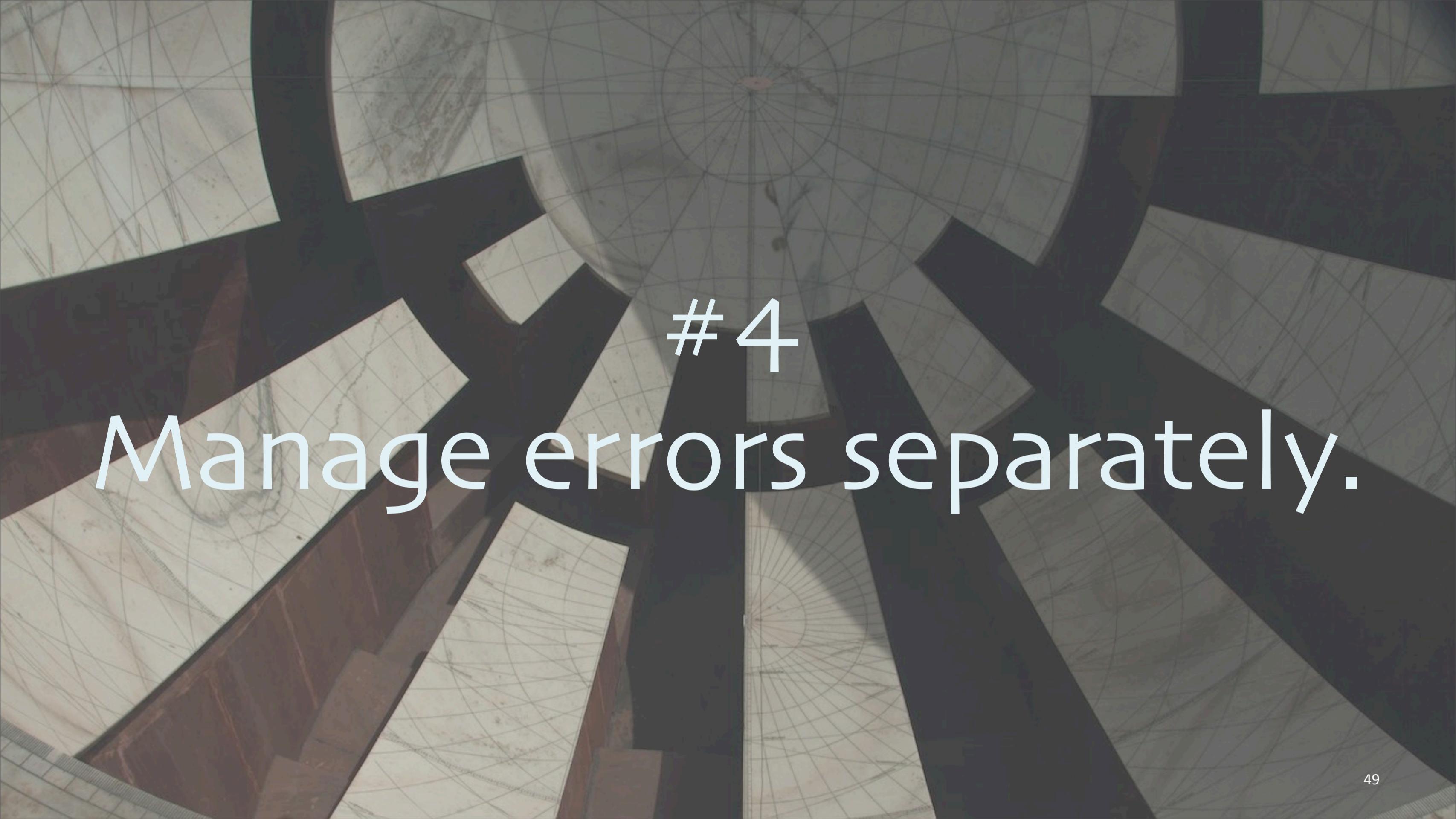
From [Deprecating the Observer Pattern with Scala.React.](#)

Can you declaratively prevent errors?

Sculthorpe and Nilsson, Safe functional reactive programming through dependent types

Wednesday, March 18, 15

True to its Haskell routes, FRP tries to use the type system to explicitly
<http://dl.acm.org/citation.cfm?doid=1596550.1596558>



#4 Manage errors separately.

49

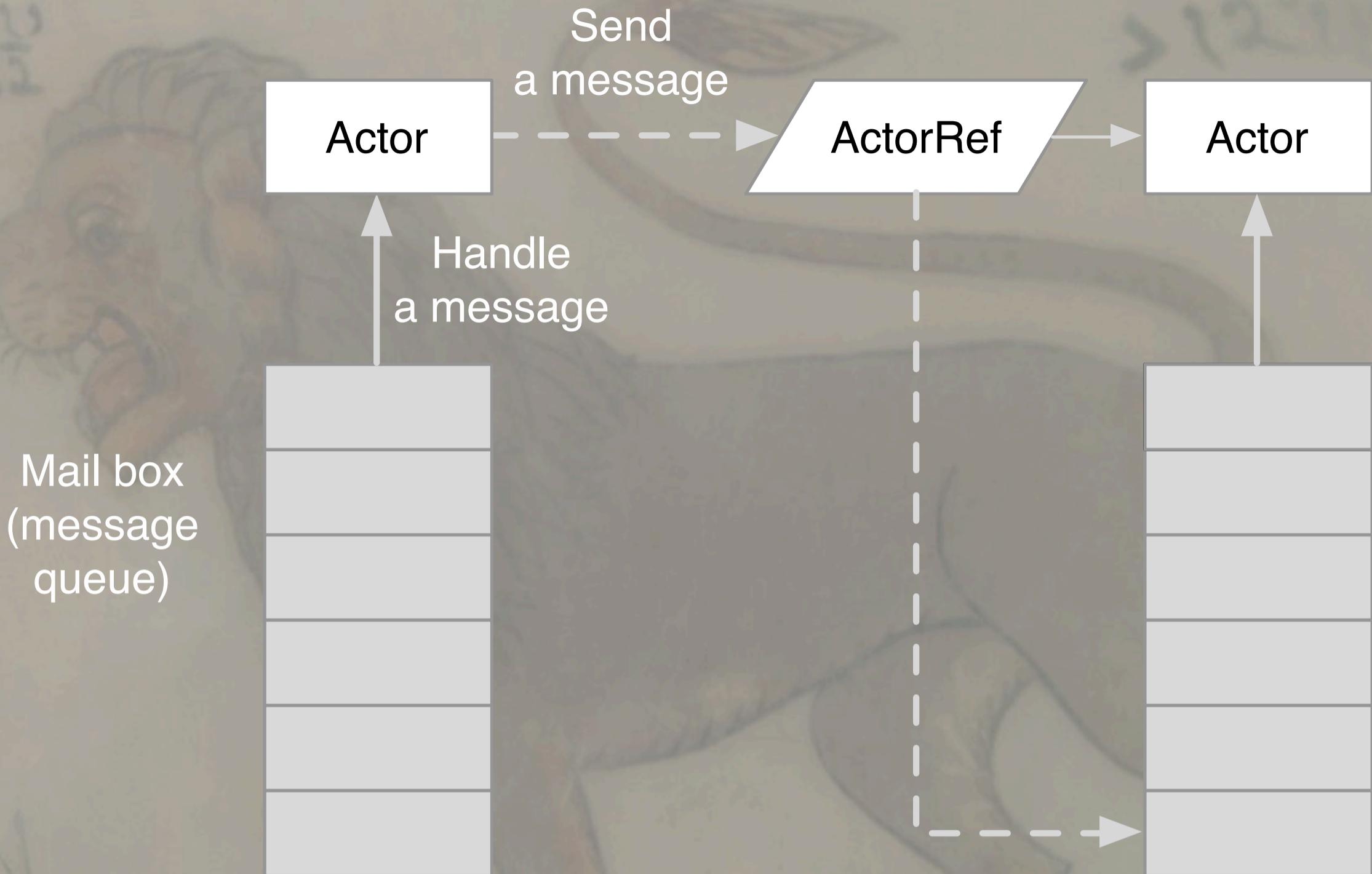
Wednesday, March 18, 15

This is how we've always done it, right?

Actor Model



Wednesday, March 18, 15

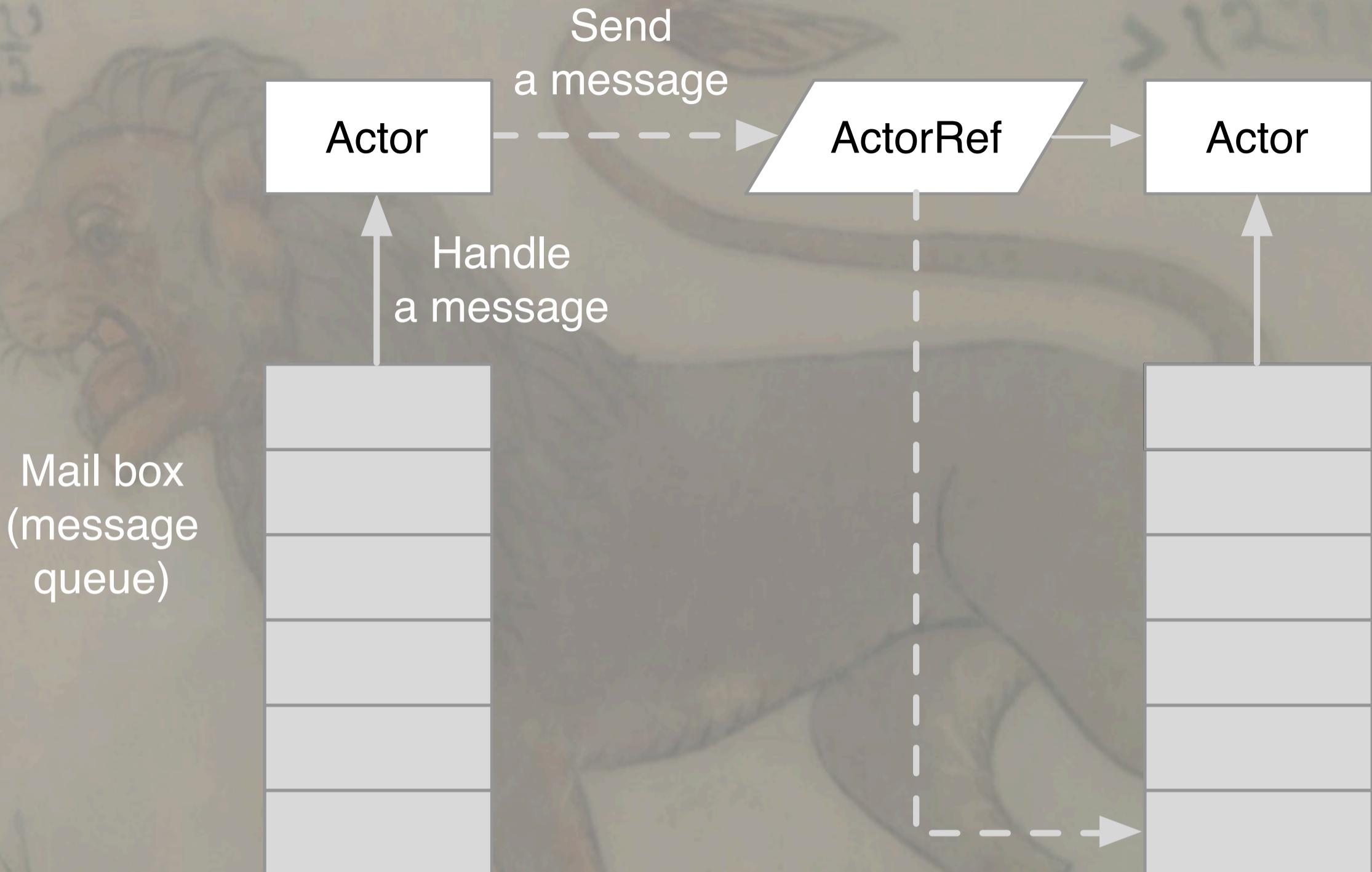


(Akka example - akka.io)

51

Wednesday, March 18, 15

This is how they look in Akka, where there is a layer of indirection, the ActorRef, between actors. This helps with the drawback that actors know each other's identities, but mostly it's there to make the system more resilient, where a failed actor can be restarted while keeping the same ActorRef that other actors hold on to.



Superficially similar to channels.

52

Wednesday, March 18, 15

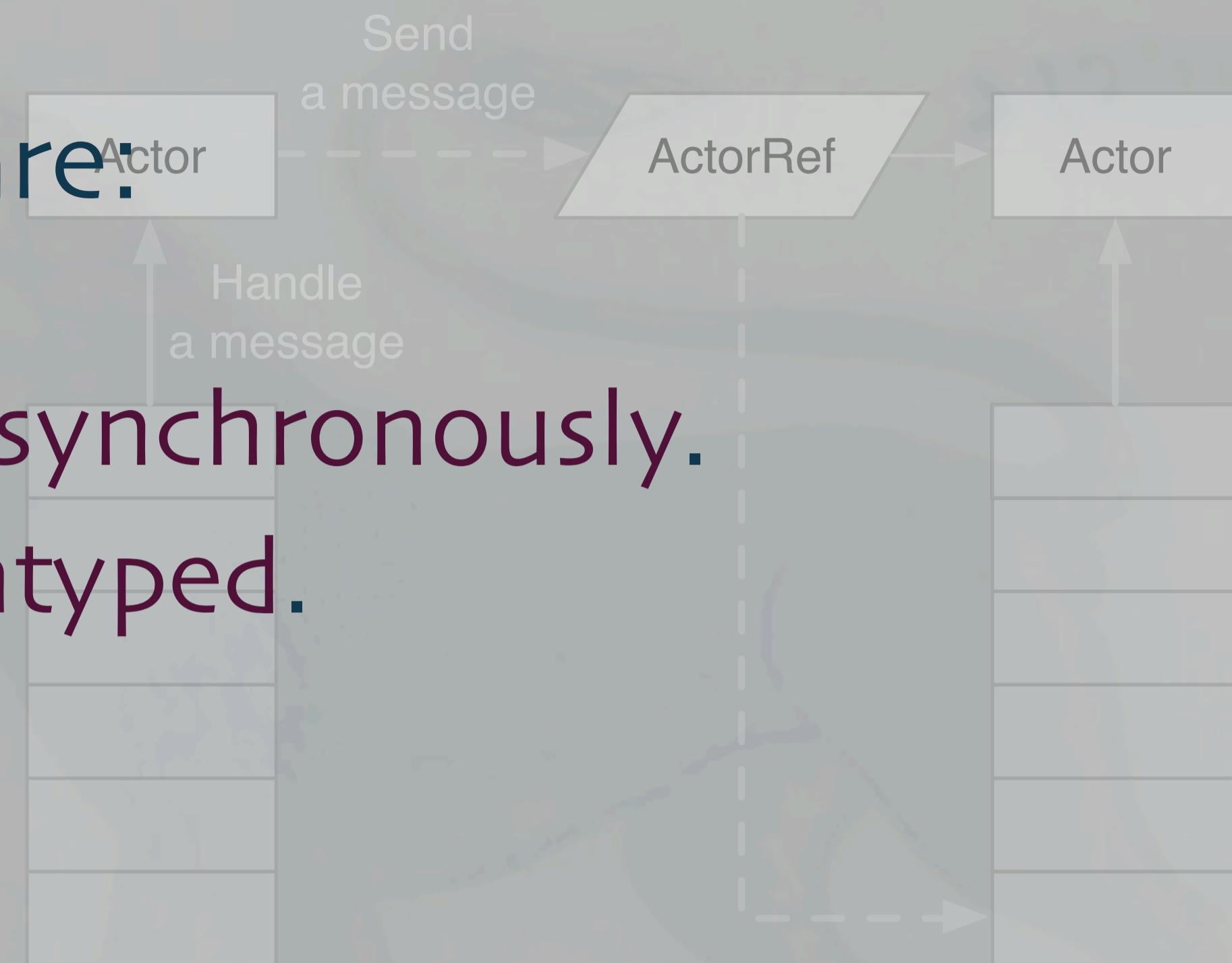
This is how they look in Akka, where there is a layer of indirection, the ActorRef, between actors. This helps with the drawback that actors know each other's identities, but mostly it's there to make the system more resilient, where a failed actor can be restarted while keeping the same ActorRef that other actors hold on to.

In response to a message, an Actor can:

- Send $0-n$ msgs to other actors.
- Create $0-n$ new actors.
- Change its behavior for responding to the next message.

Messages are:

- Handled asynchronously.
- Usually untyped.



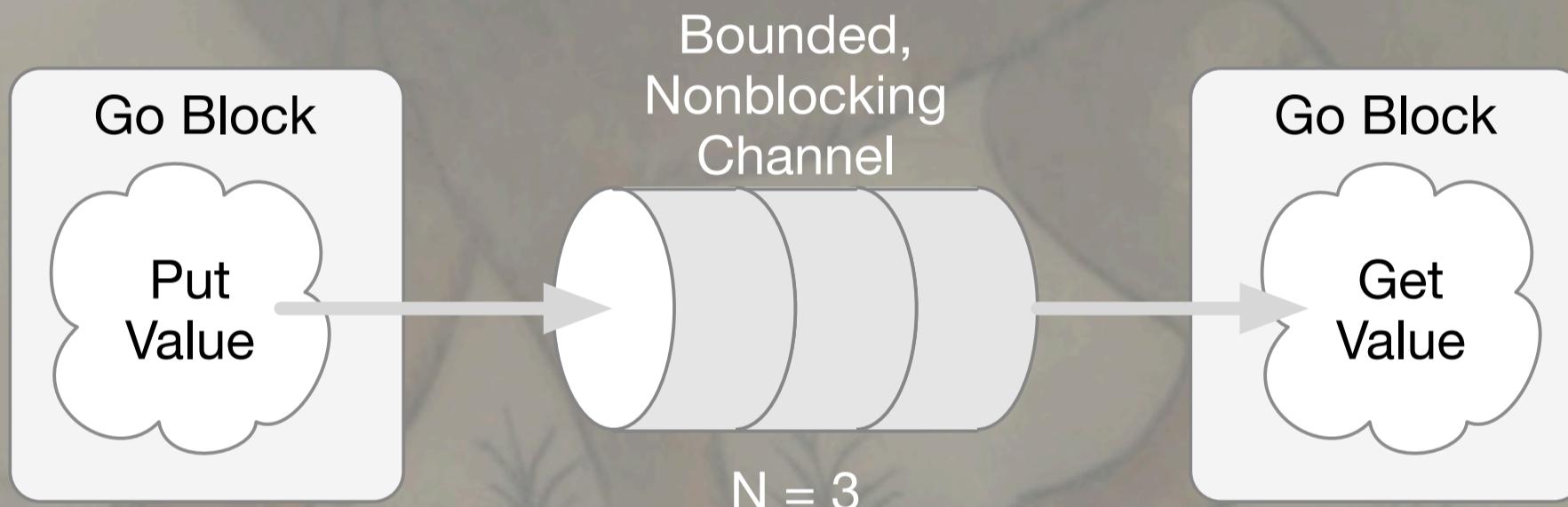
CSP and Actors are dual



Wednesday, March 18, 15

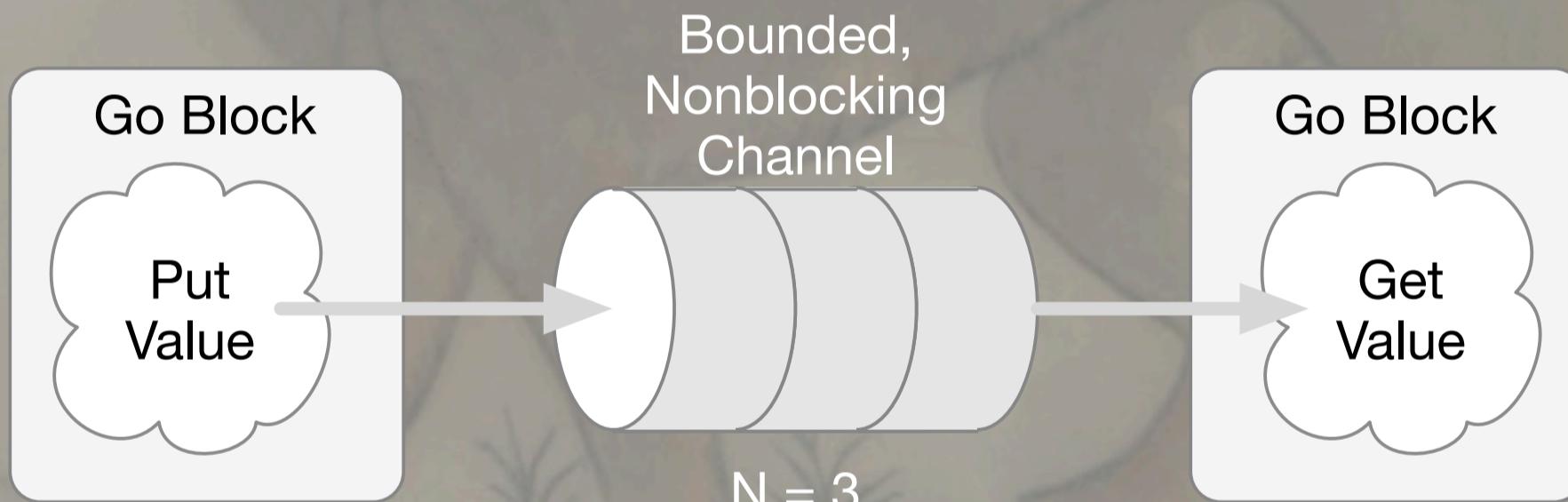
CSP Processes are anonymous

... while actors have identities.



CSP messaging is synchronous

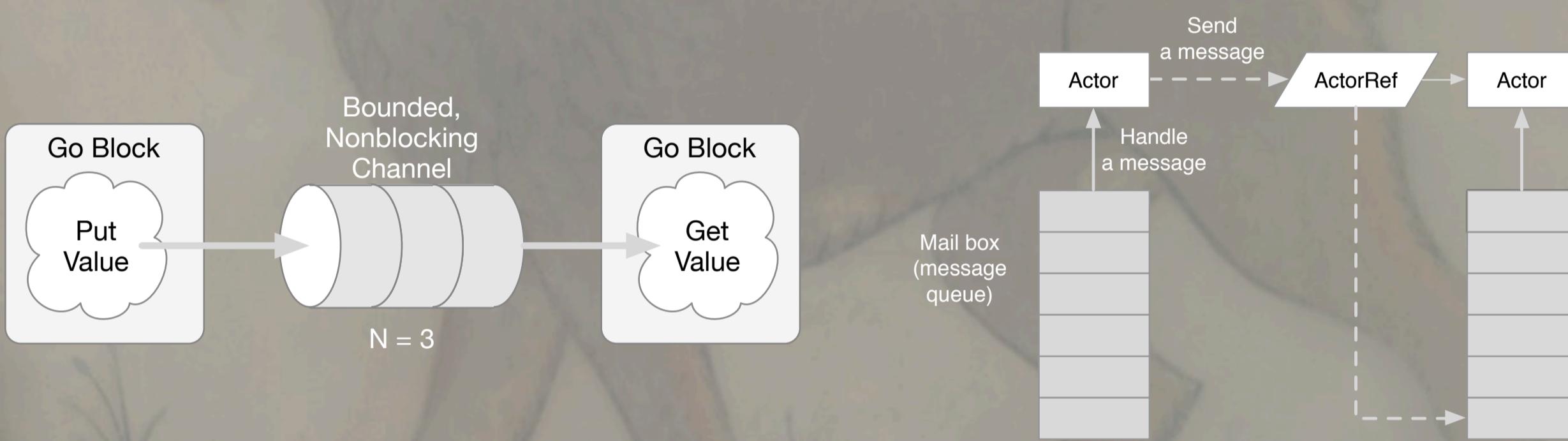
A sender and receiver must rendezvous,
while actor messaging is asynchronous.





... but CSP and Actors
can implement
each other

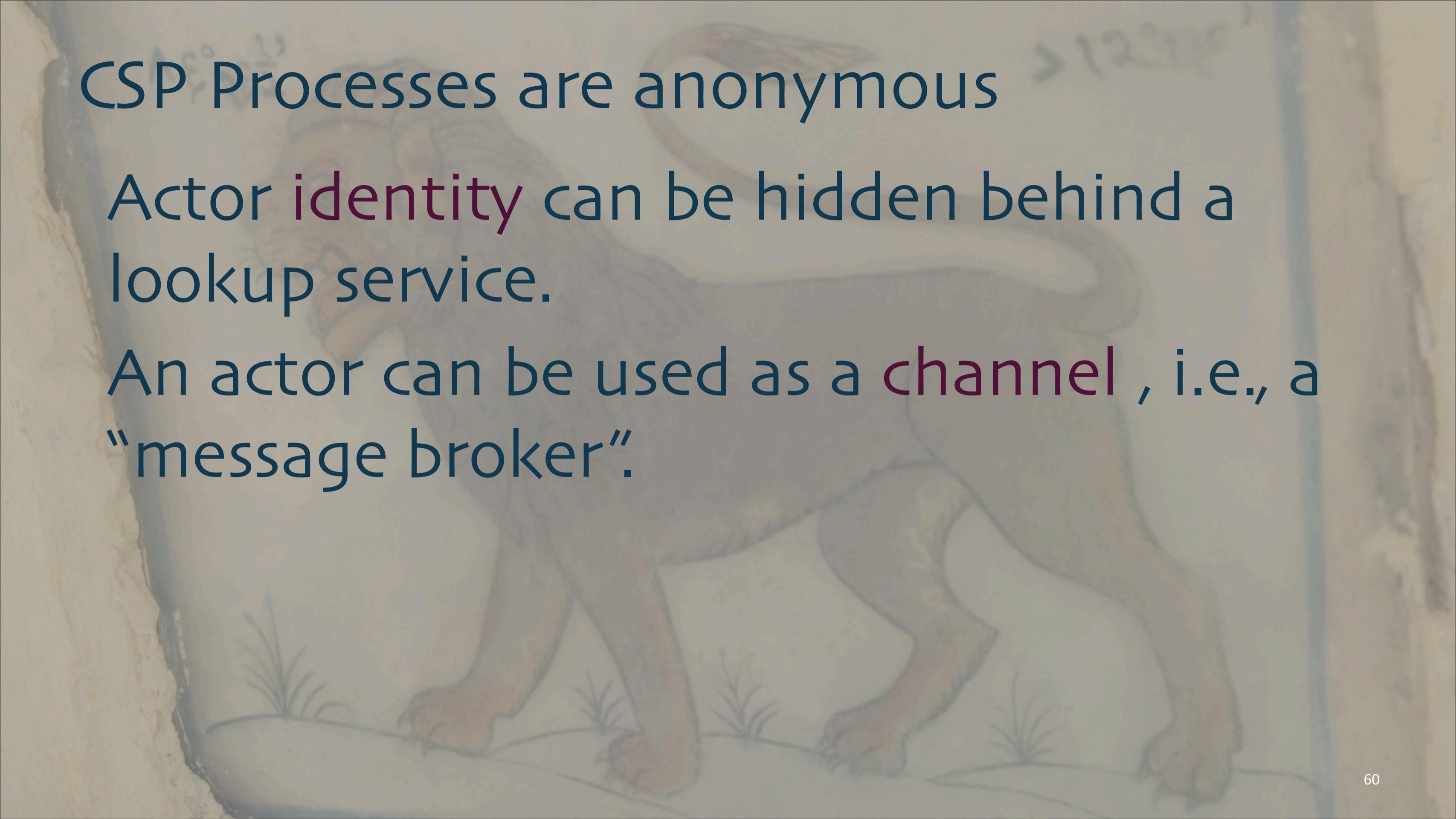
An actor mailbox looks a lot like a channel.



59

Wednesday, March 18, 15

In actors, the receiver doesn't even need to be ready to receive messages yet.



CSP Processes are anonymous

Actor identity can be hidden behind a lookup service.

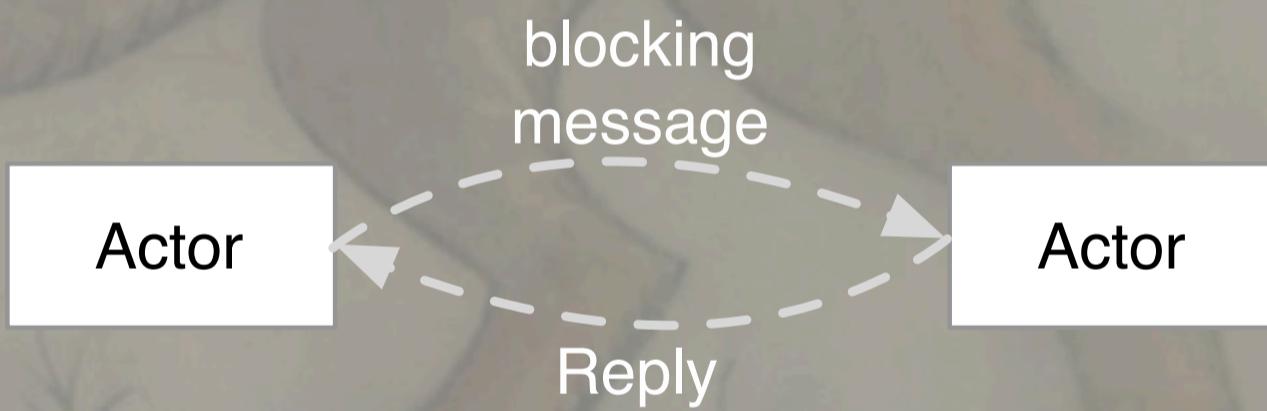
An actor can be used as a channel , i.e., a “message broker”.

CSP Processes are anonymous

Conversely, a reference to the channel is often shared between a sender and receiver.

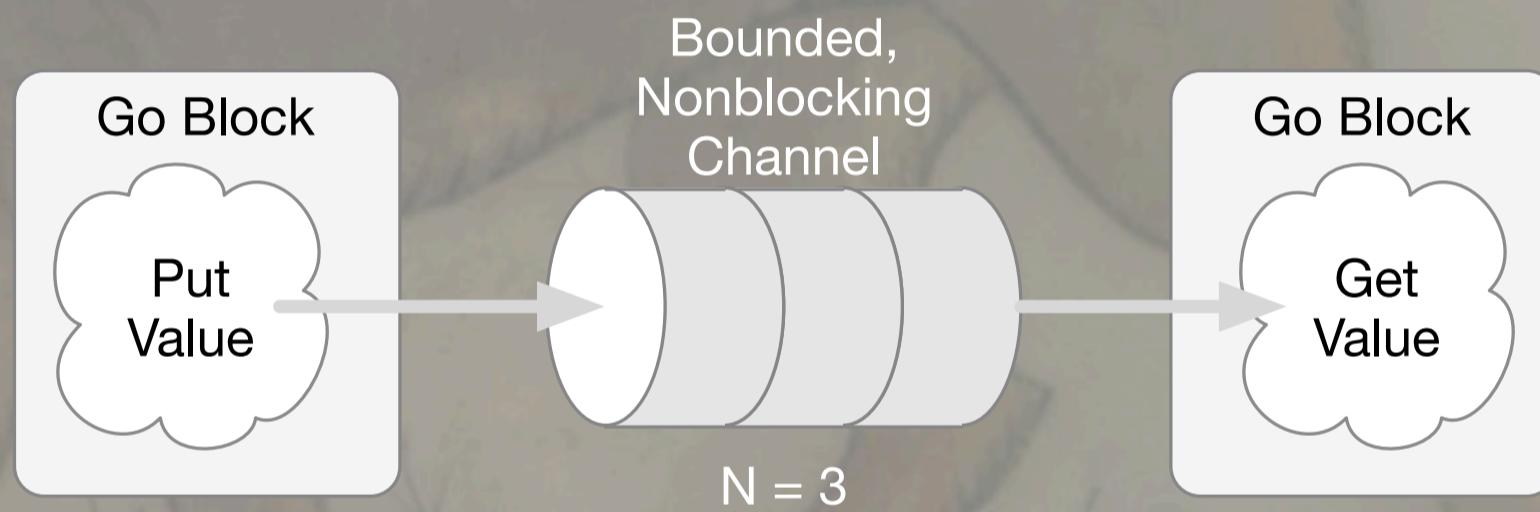
CSP messaging is synchronous

Actor messaging can be synchronous if the sender uses a blocking message send that waits for a response.



CSP messaging is synchronous

Buffered channels behave
asynchronously.



Erlang and Akka

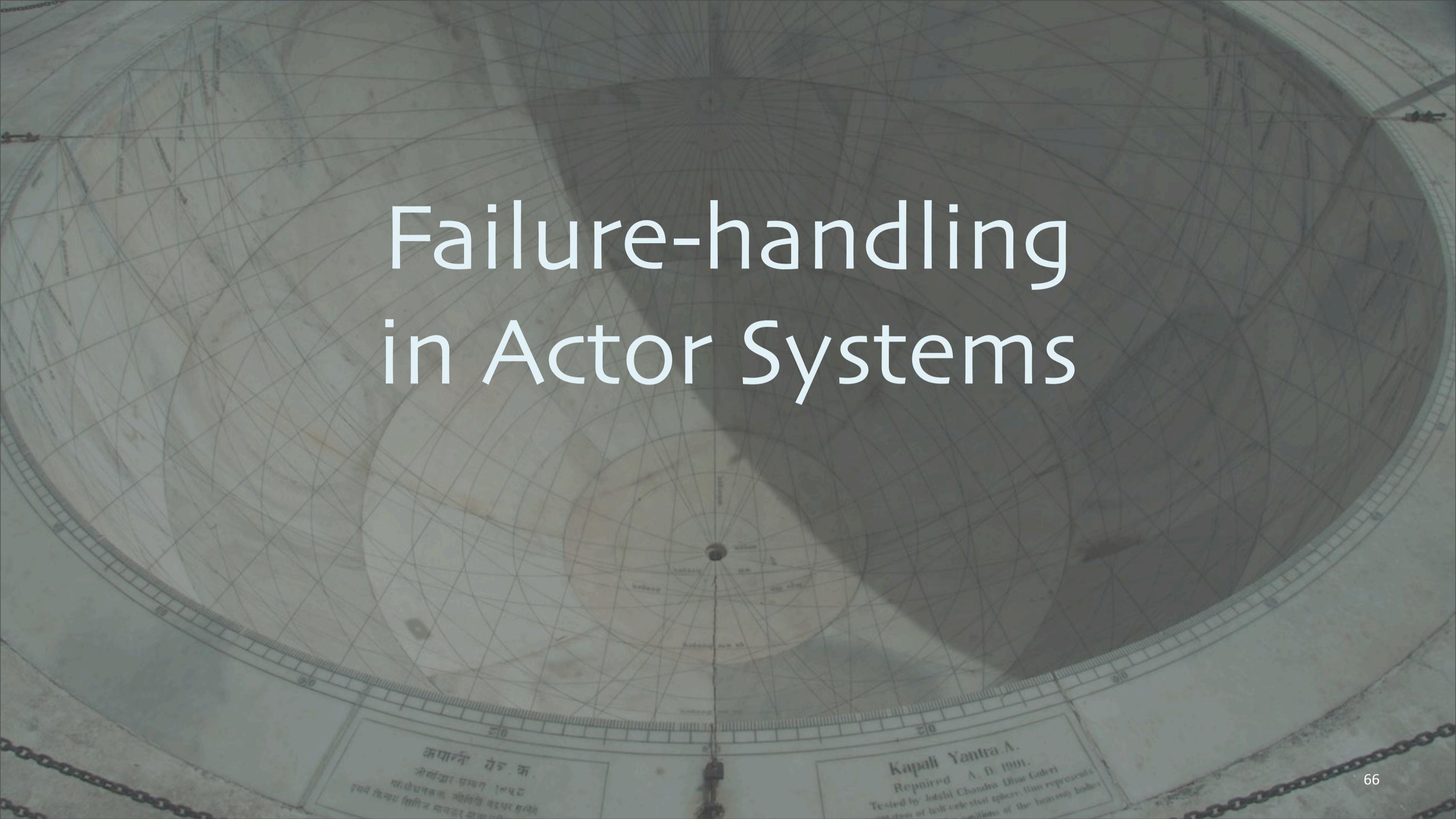


Wednesday, March 18, 15

Distributed Actors

- Generalize actor identities to URLs.
- But distribution adds a number of failure modes...

Failure-handling in Actor Systems



Wednesday, March 18, 15

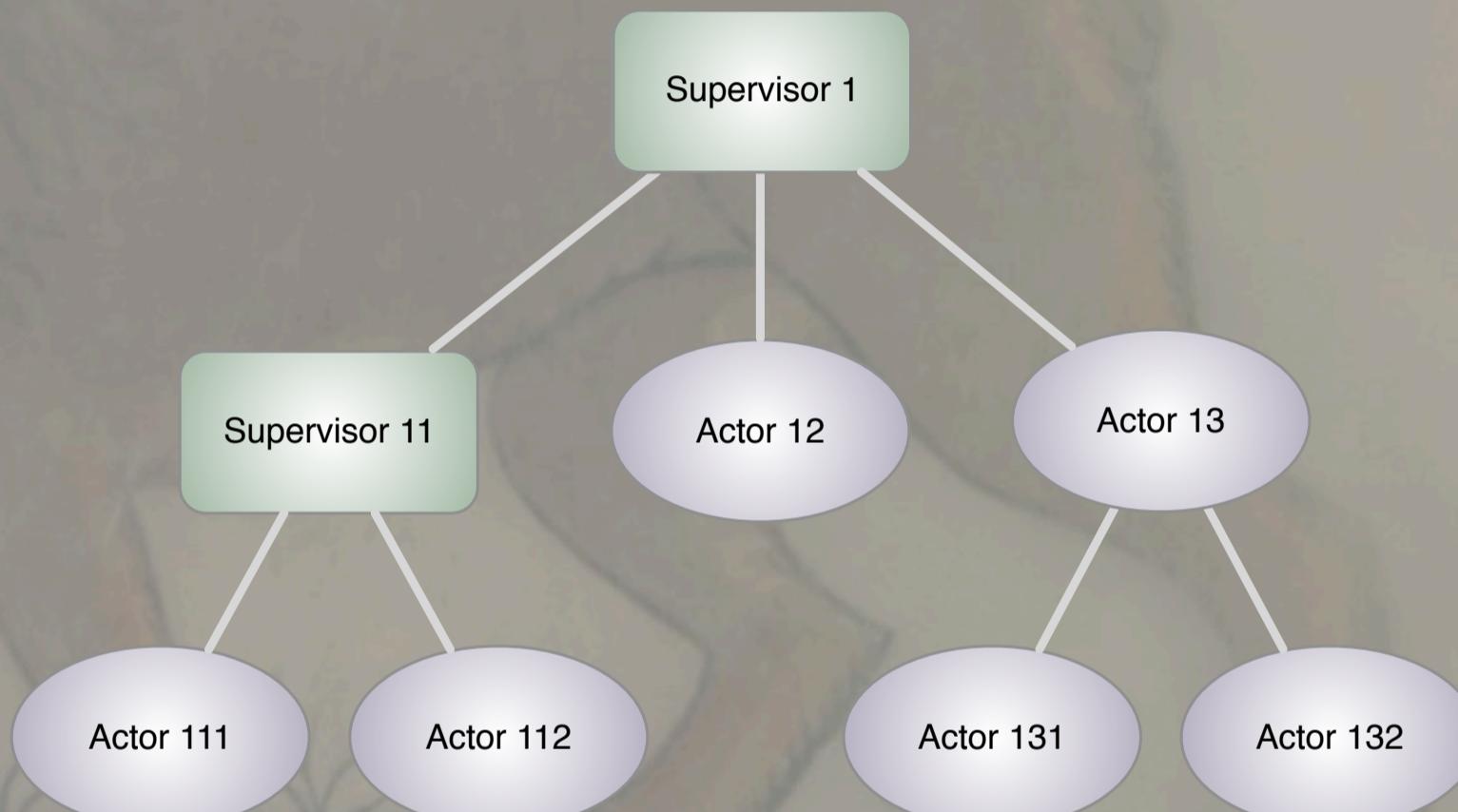
Let it Crash!



Wednesday, March 18, 15

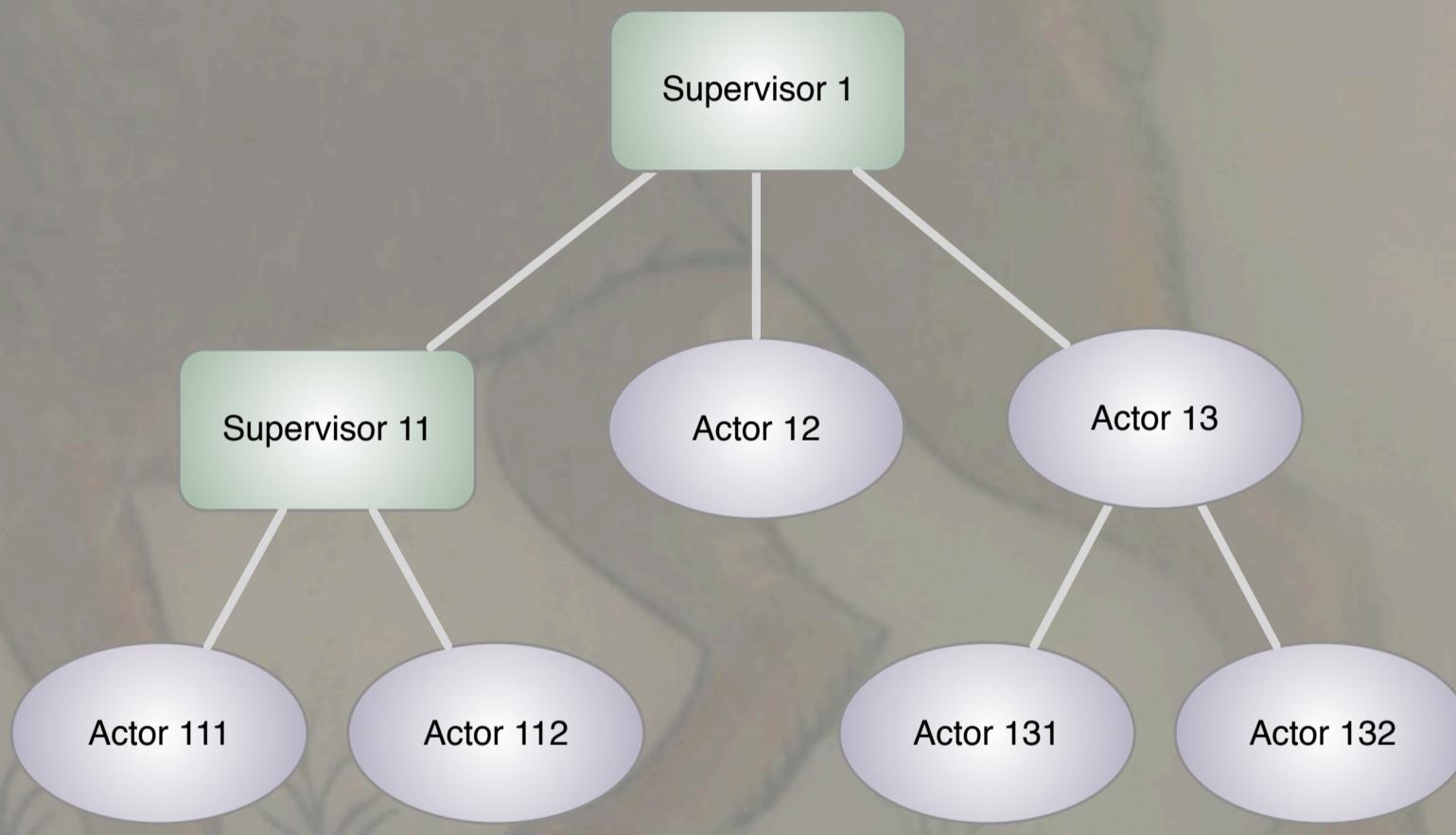
Erlang introduced supervisors

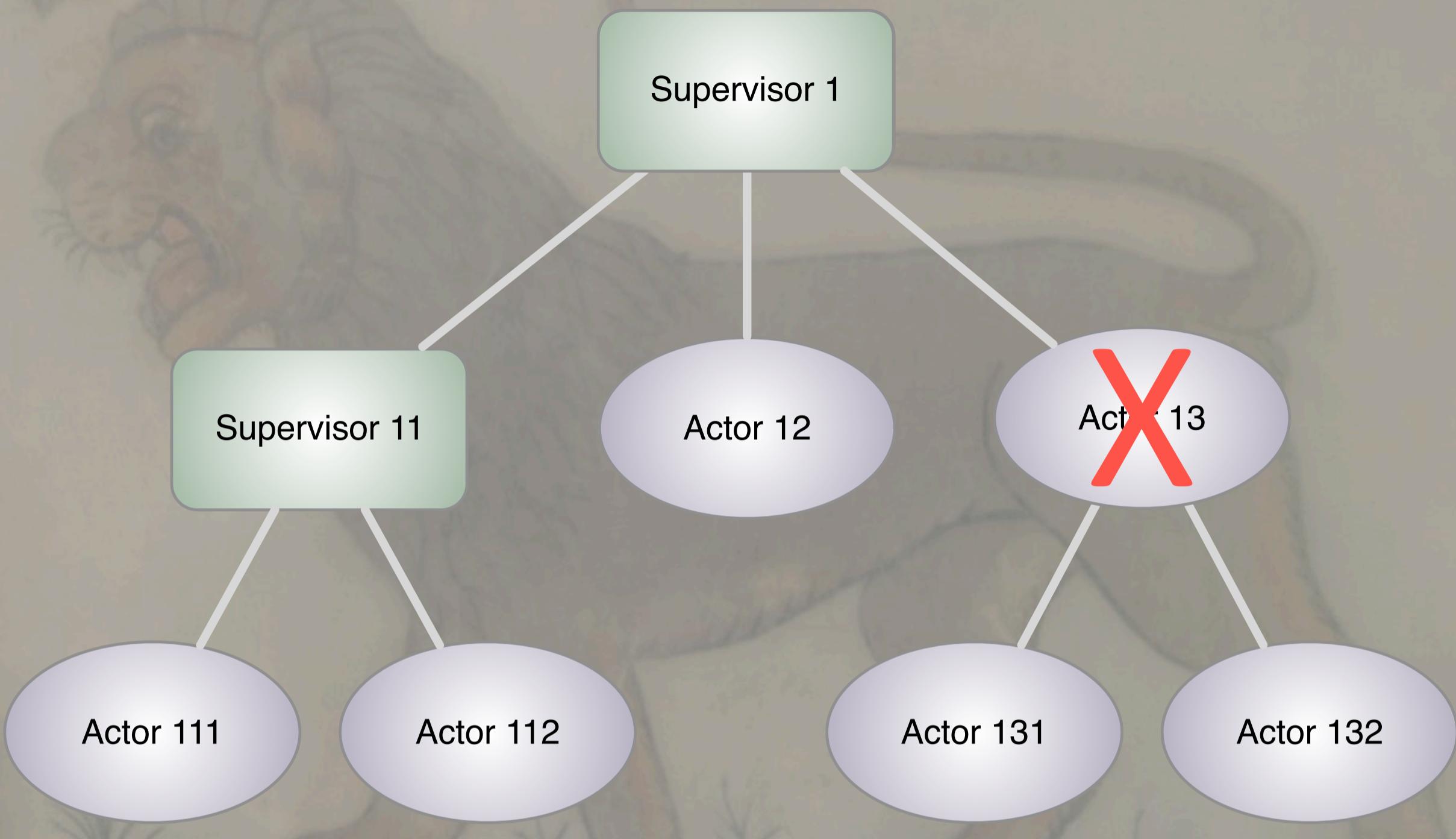
A hierarchy of actors that manage each “worker” actor’s lifecycle.

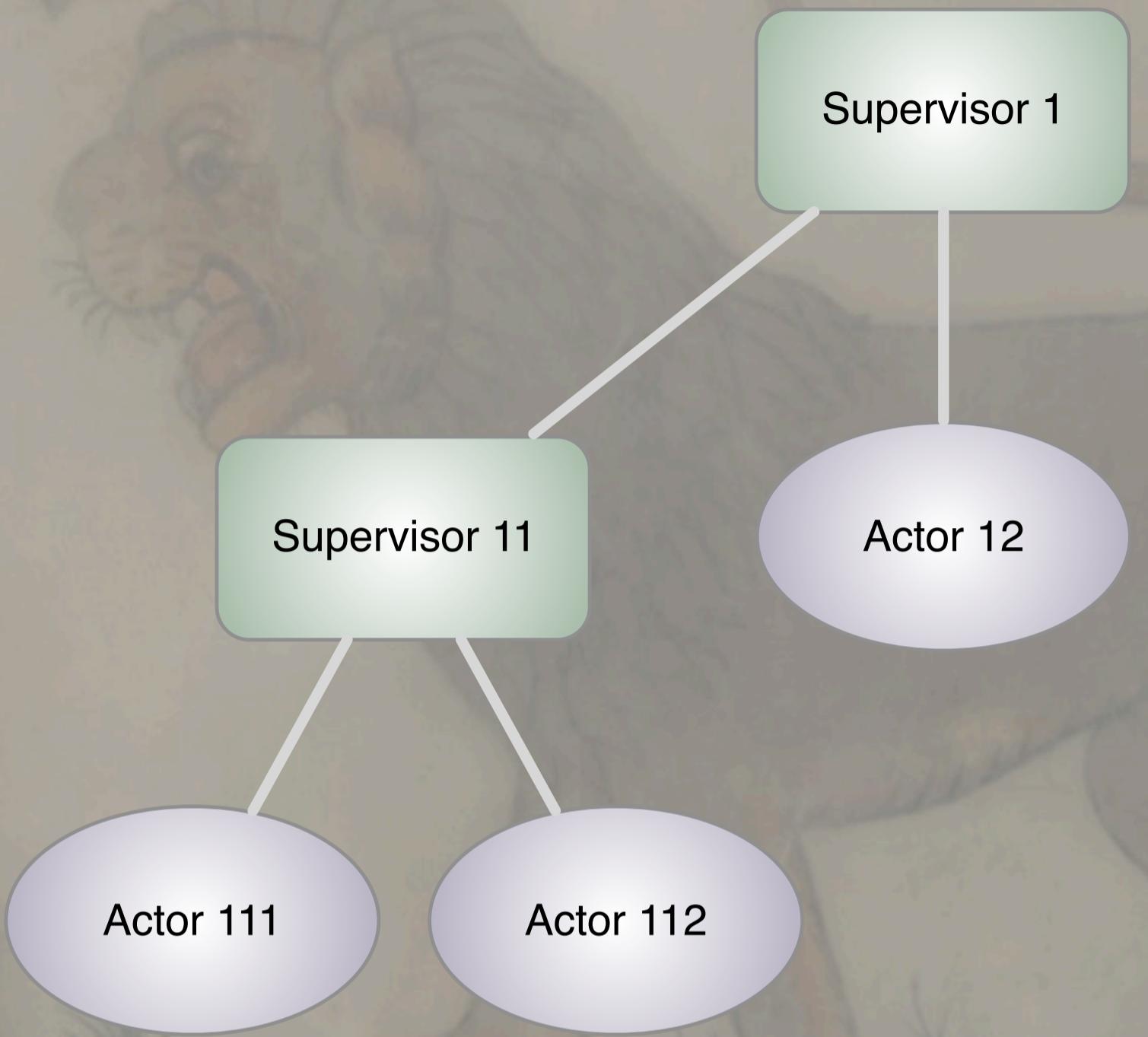


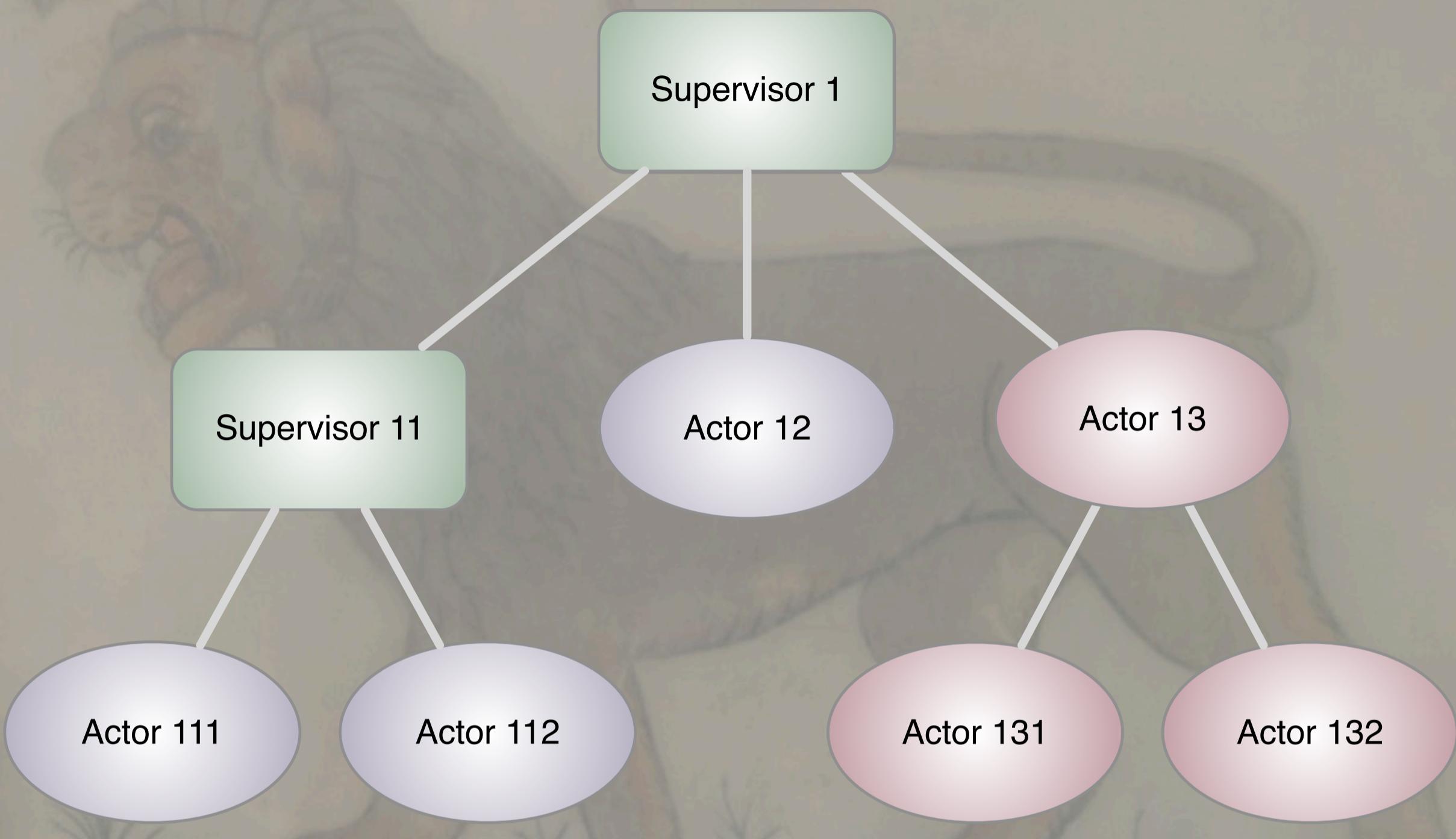
Erlang introduced supervisors

Generalizes nicely to distributed actor systems.





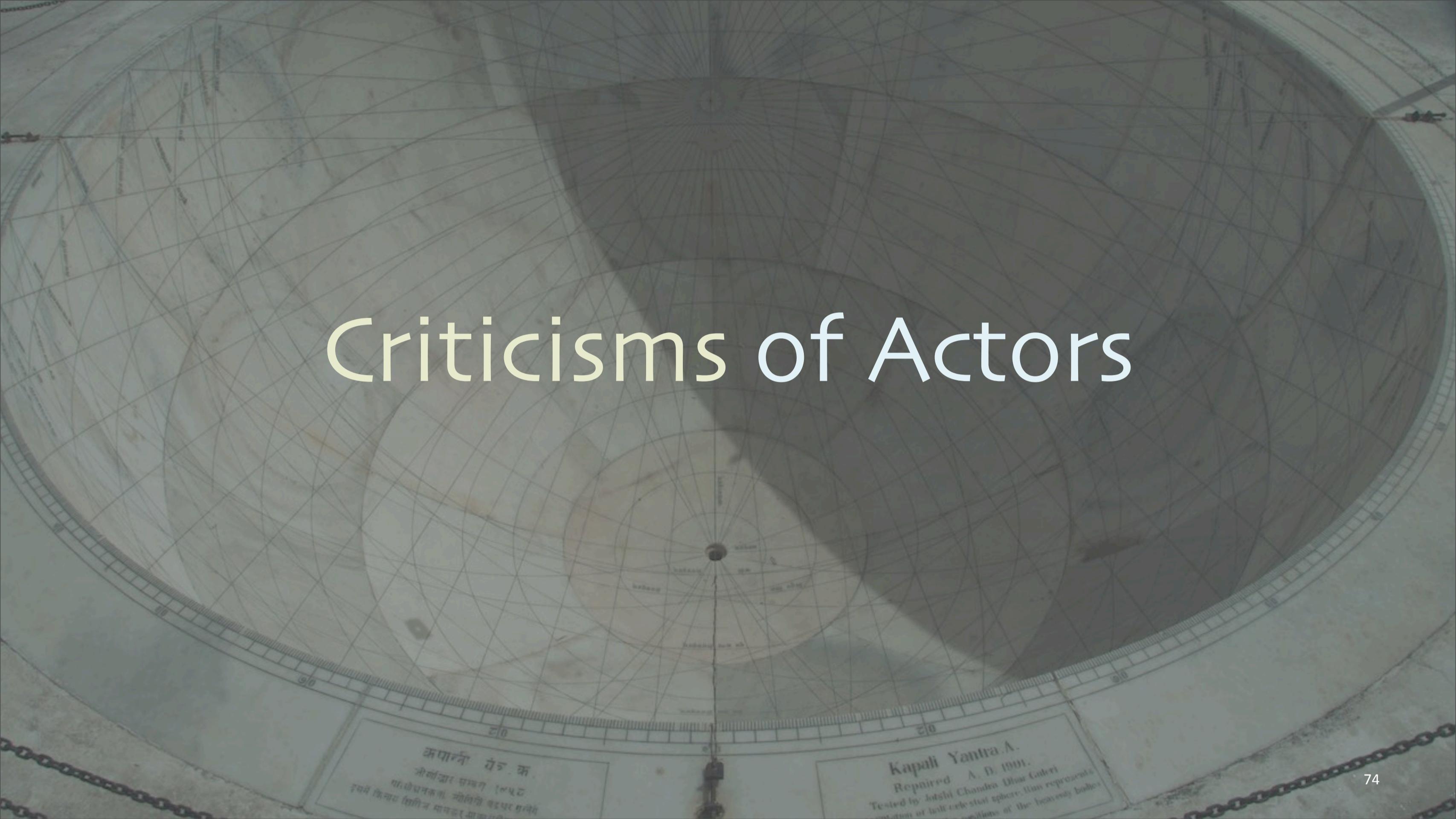




Advantages

- Enables **strategic** error handling across module boundaries.
- Separates **normal** and error logic.
- Failure handling is **configurable** and **pluggable**.

Criticisms of Actors



Wednesday, March 18, 15

Rich Hickey

[Actors] still couple the producer with the consumer. Yes, one can emulate or implement certain kinds of queues with actors, but since any actor mechanism already incorporates a queue, it seems evident that queues are more primitive. ... and channels are oriented towards the flow aspects of a system.

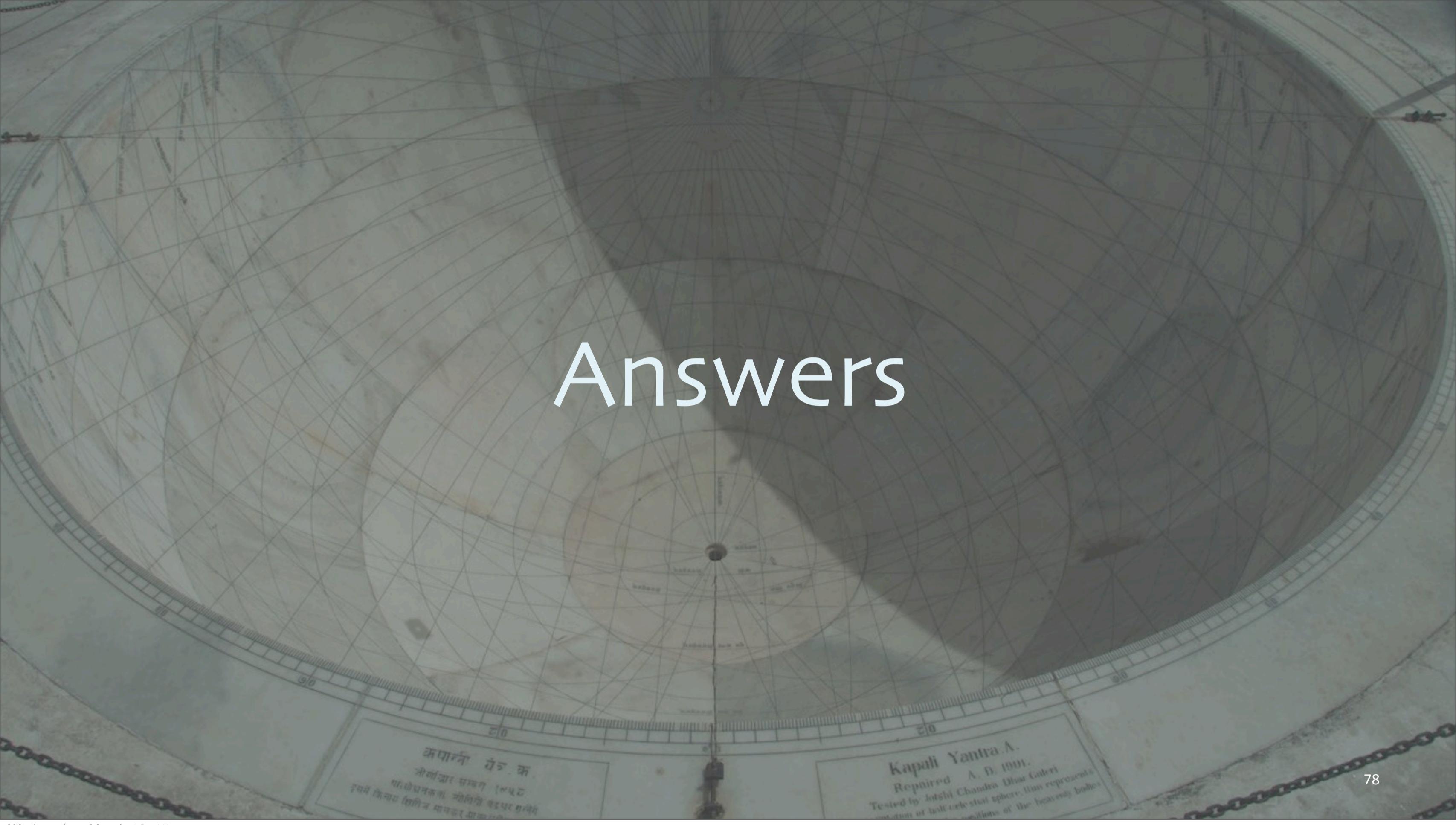
Other Criticisms

- Unbounded queues (mailboxes).
- Internal mutating state (hidden in function closures).
- Must send message to deref state. What if the mailbox is backed up?
- Couples a queue, mutating state, and a process.
- Effectively “asynchronous OOP”.

I'll add...

- Most actor systems are untyped.
- Typed channels add that extra bit of type safety.

Answers



Wednesday, March 18, 15

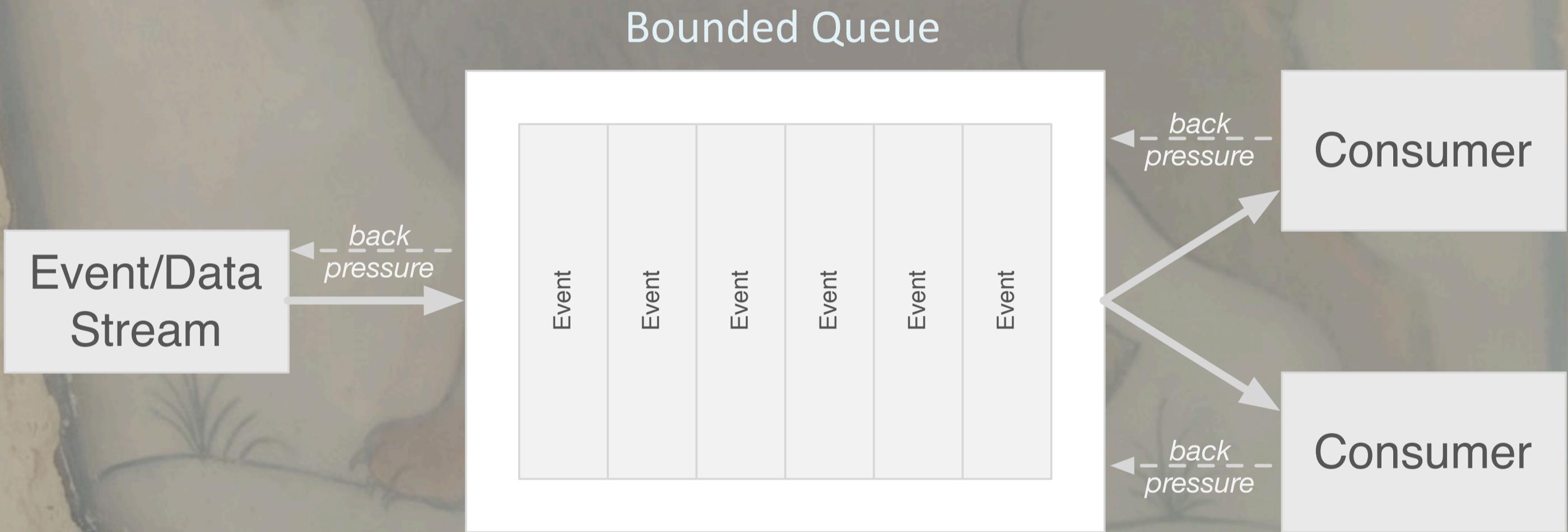
The fact that Actors and CSP can be used to implement each other suggests that the criticisms are less than meets the eye...

Unbounded queues

- Bounded queues are available in production-ready Actor implementations.
- Reactive Streams with back pressure enable strategic management of flow.
 - Can be implemented with Actors.

Reactive Streams

Akka streams provide a higher-level abstraction on top of Actors with better type safety (effectively, typed channels) and operational semantics.



Internal mutating state

- Actually an advantage.
- Encapsulation of mutating state within an Actor is a systematic approach to large-scale, reliable management of state evolution.
- “Asynchronous OOP” is a fine strategy when it fits your problem.

Must send message to get state

- Also an advantage.
- Protocol for coordinating and separating reads and writes.
 - But you could also have an actor send the new state as a response message to the sender or broadcast to “listeners”.

Couples a queue, mutable state, and a process

- Production systems provide as much decoupling as you need.

Actors are untyped

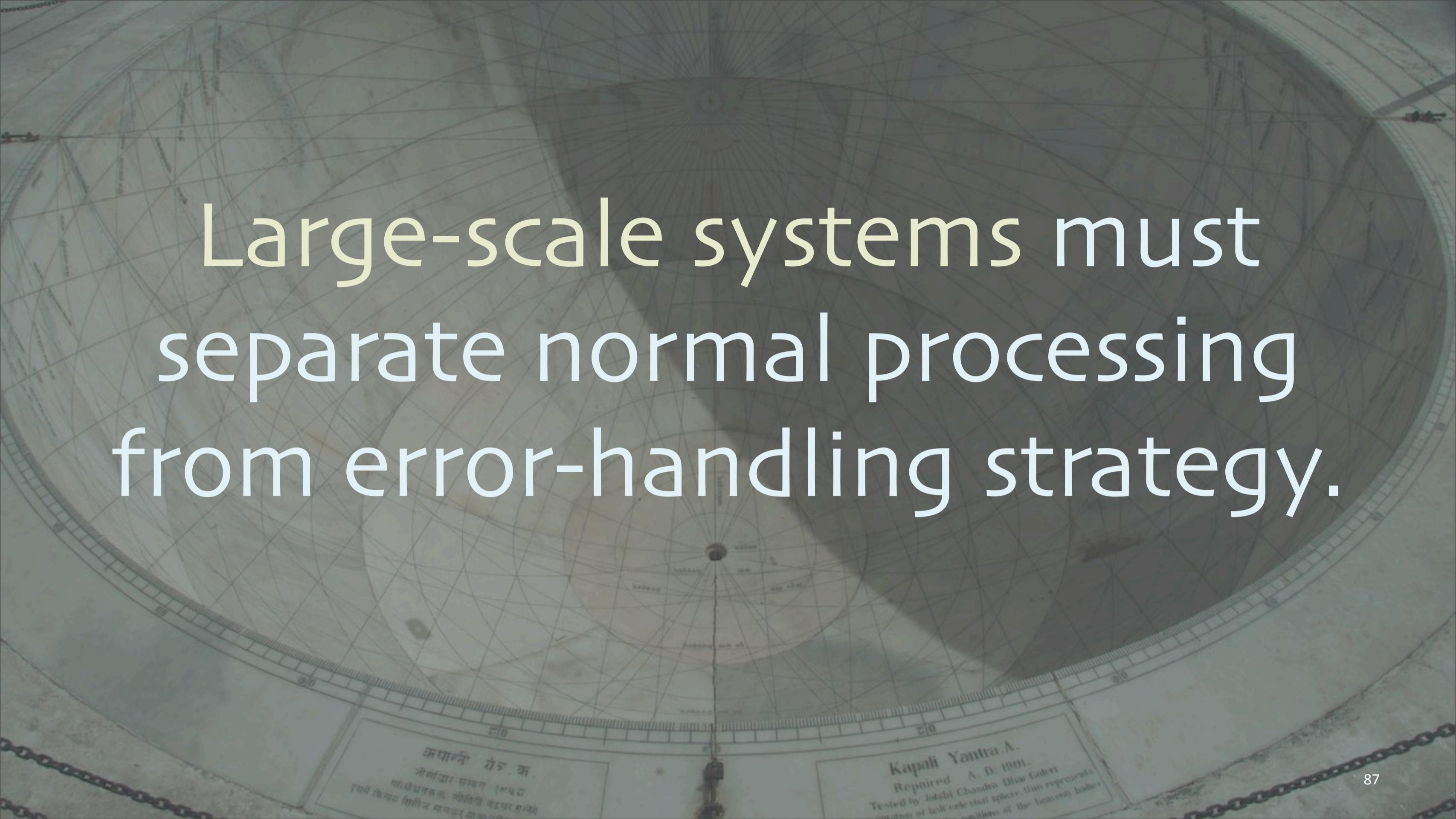
- While typed actor experiments continue, I think of actors as analogs of OS processes:
 - Clear abstraction boundaries.
 - Must be paranoid about the data you're ingesting.

Actors are untyped

- ... but actually, Akka is adding typed ActorRefs.



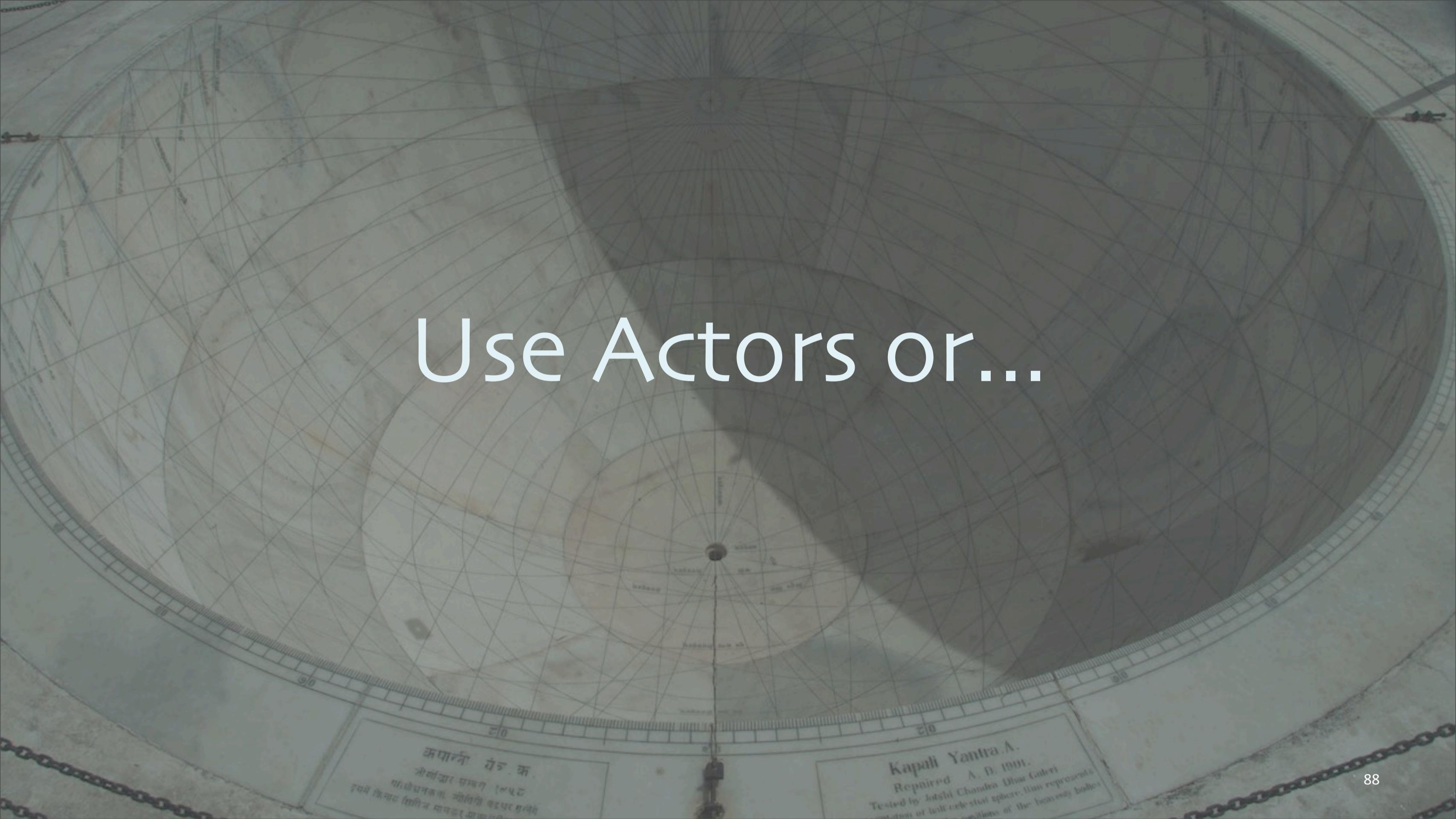
How should
we handle
failures?



Large-scale systems must separate normal processing from error-handling strategy.

Wednesday, March 18, 15

Not all concurrency problems require something as sweeping as an actor system with supervisors, but at a certain scale, you'll need some sort of separation between your recovery strategy and the normal processing logic.



Use Actors or...

कपान्त यंत्र के
स्त्रीलघु यंत्र समग्र १८५८
प्राचीन यंत्र का विवरण देखने के लिए इस पर्याप्त
क्षमता के लिए यह यंत्र उपयोगी है।

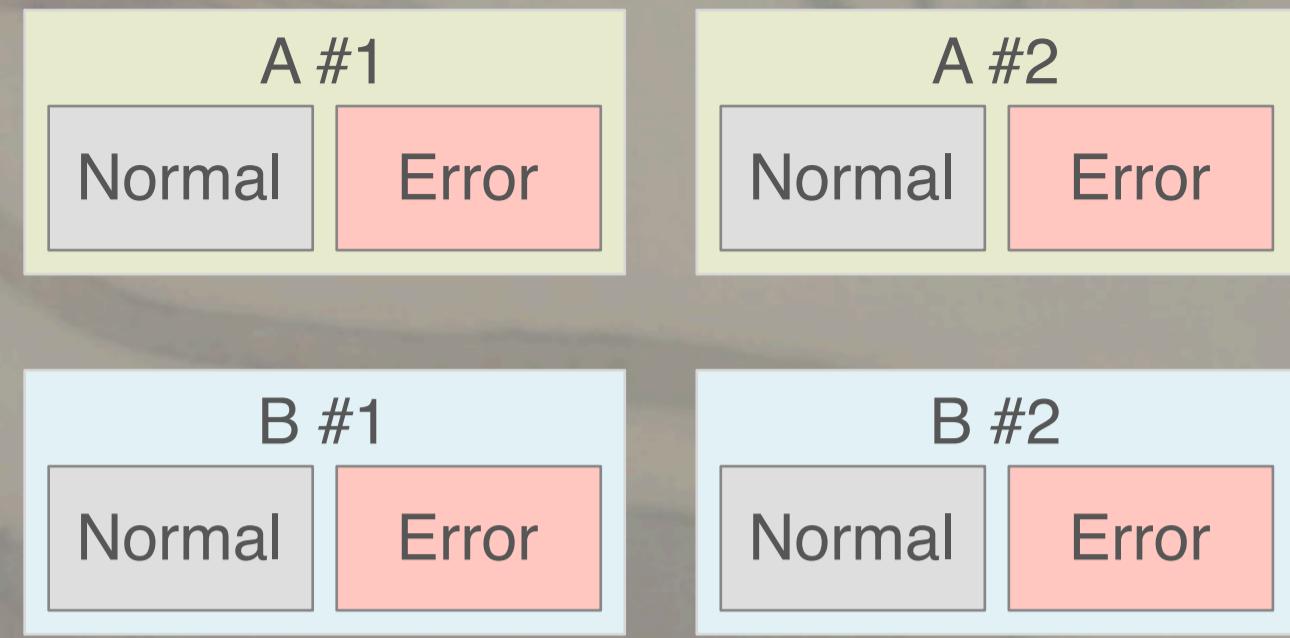
Kapali Yantra A.
Repaired A. D. 1901.
Tested by Jitshi Chandra Dhar Guleri
This is a diagram of the celestial sphere. It represents
the positions of the heavenly bodies.



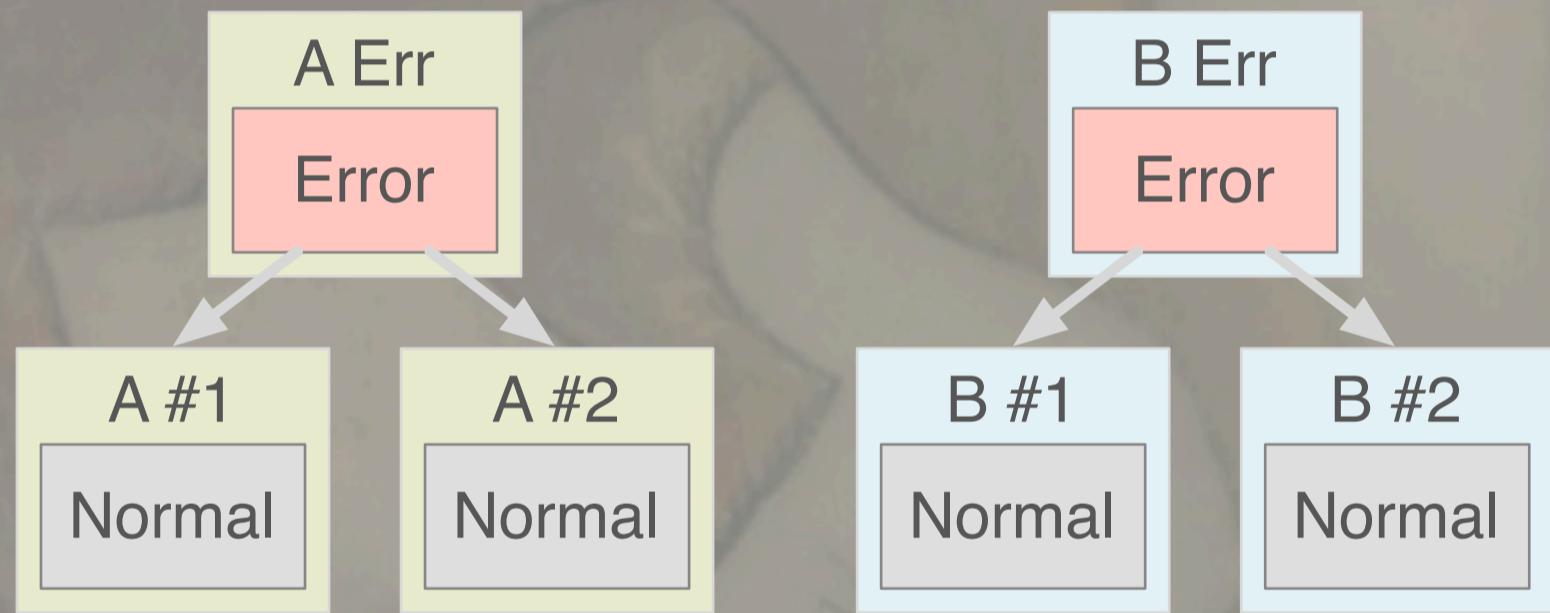
<https://github.com/Netflix/Hystrix>

- Better separation of concerns.
 - Failure handling delegated to a separate component or service.
- Strategy for failure handling can be pluggable.
- Better scalability...

- Better scalability:



VS.



Conclusions



Wednesday, March 18, 15

Actors

- Untyped interfaces.
- More OOP than FP.
- Overhead higher than function calls.

Actors

- Actually quite low level:
 - Analog of OS processes.
 - Reactive Streams is a functional, higher-level abstraction that can be built on actors.

Actors

- + Industry proven scalability and resiliency.
- + Native asynchrony.
- + Distribution is a natural extension.

Best-in-class strategy for failure handling.

CSP, Rx, etc.

- Limited failure handling facilities.
- Distributed channels?

CSP, Rx, etc.

+ Emphasize data flows.

+ Typed channels.

Optimal replacement for multithreaded
(intra-process) programming.



Typesafe

<http://typesafe.com/reactive-big-data>
dean.wampler@typesafe.com
polyglotprogramming.com/talks

©Typesafe 2014-2015, All Rights Reserved

Wednesday, March 18, 15

Photos from Jantar Mantar (“instrument”, “calculation”), the astronomical observatory built in Jaipur, India, by Sawai Jai Singh, a Rajput King, in the 1720s-30s. He built four others around India. This is the largest and best preserved. All photos are copyright (C) 2012-2015, Dean Wampler. All Rights Reserved.

Bonus Slides

Communicating Sequential Processes

Message passing
via channels



Wednesday, March 18, 15

See

http://en.wikipedia.org/wiki/Communicating_sequential_processes

<http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html>

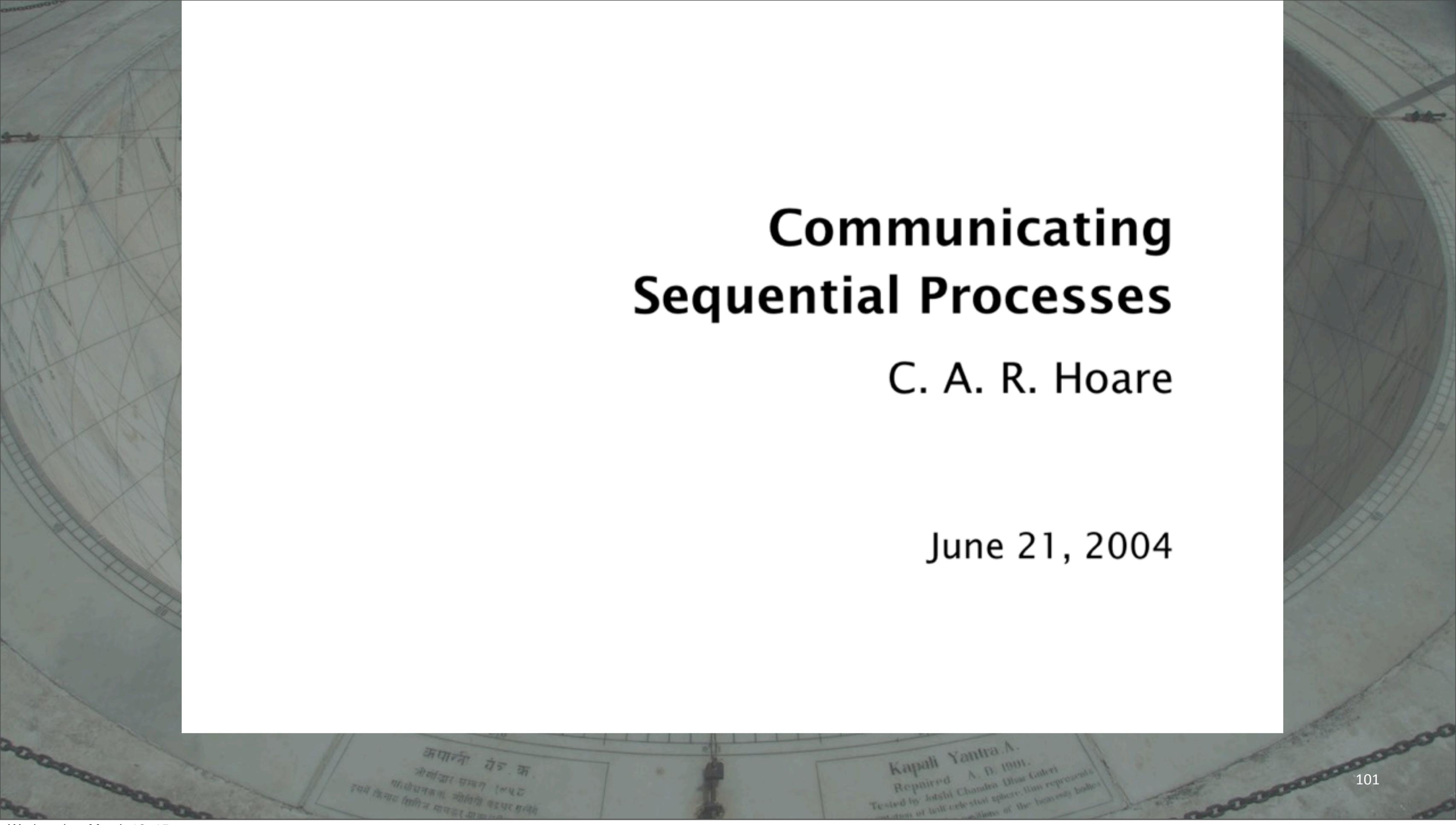
<http://blog.drewolson.org/blog/2013/07/04/clojure-core-dot-async-and-go-a-code-comparison/>

and other references in the “bonus” slides at the end of the deck. I also have some slides that describe the core primitives of CSP that I won’t have time to cover.

Communicating Sequential Processes

C. A. R. Hoare

June 21, 2004



Wednesday, March 18, 15

Hoare's book on CSP, originally published in '85 after CSP had been significantly evolved from the initial programming language he defined in the 70's to a theoretical model with a well-defined calculus by the mid 80's (with the help of other people, too). The book itself has been subsequently refined. The PDF is available for free.

The Theory and Practice of Concurrency

A.W. Roscoe

Published 1997, revised to 2000 and lightly revised to 2005.

The original version is in print in April 2005 with Prentice-Hall (Pearson).
This version is made available for personal reference only. This version is
copyright (©) Pearson and Bill Roscoe.

Modern treatment of CSP. Roscoe helped transform the original CSP language into its more rigorous, process algebra form, which was influenced by Milner's Calculus of Communicating Systems work. This book's PDF is available free. This treatment is perhaps more accessible than Hoare's book.

CSP Operators



Wednesday, March 18, 15

Prefix

$$a \rightarrow P$$

A process communicates event a to its environment. Afterwards the process behaves like P .

Deterministic Choice

$$a \rightarrow P \quad \square \quad b \rightarrow Q$$

A process communicates event a or b to its environment. Afterwards the process behaves like P or Q , respectively.

Nondeterministic Choice

$$a \rightarrow P \sqcap b \rightarrow Q$$

The process doesn't get to choose which is communicated, a or b .

Interleaving

$P \parallel\!\!\!|| Q$

Completely independent processes. The events seen by them are interleaved in time.

Interface Parallel

$$P \mid[\{a\}] \mid Q$$

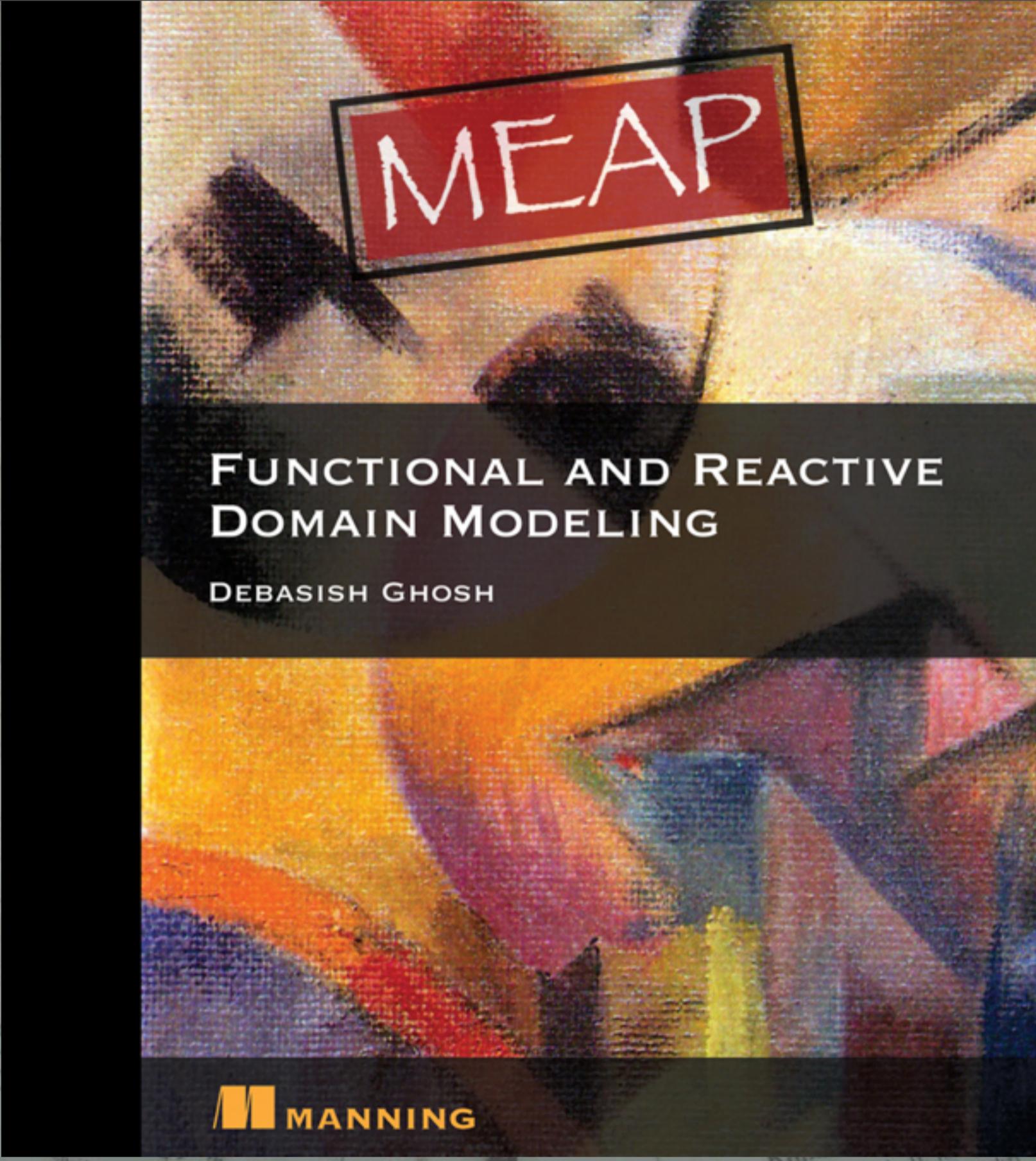
Represents synchronization on event a between P and Q .

Hiding

$$a \rightarrow P \setminus \{a\}$$

A form of abstraction, by making some events unobservable. P hides events a .

References



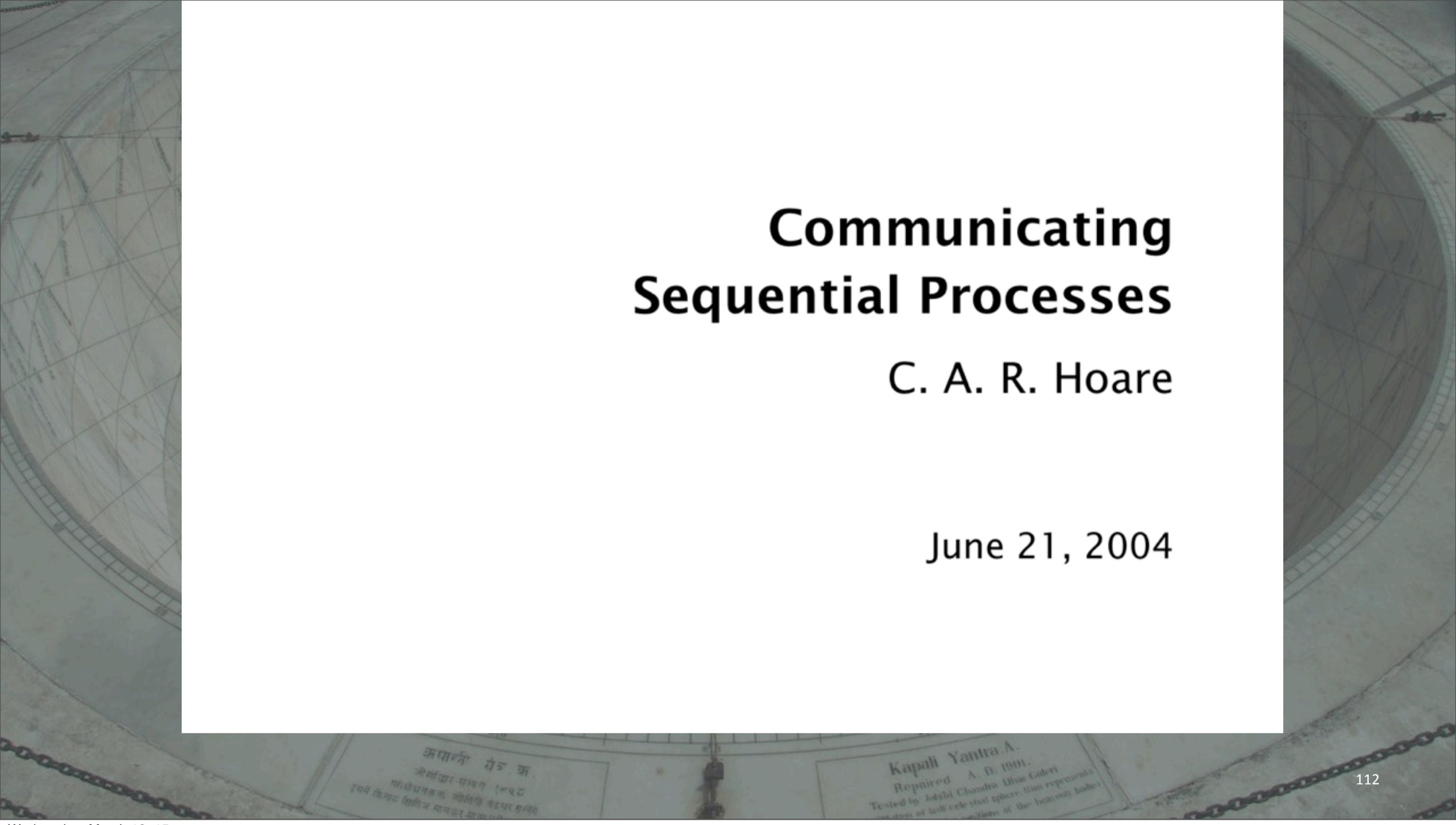
Wednesday, March 18, 15

Lots of interesting practical ideas for combining functional programming and reactive approaches to class Domain-Driven Design by Eric Evans.

Communicating Sequential Processes

C. A. R. Hoare

June 21, 2004



Wednesday, March 18, 15

Hoare's book on CSP, originally published in '85 after CSP had been significantly evolved from a programming language to a theoretical model with a well-defined calculus. The book itself has been subsequently refined. The PDF is available for free.

The Theory and Practice of Concurrency

A.W. Roscoe

Published 1997, revised to 2000 and lightly revised to 2005.

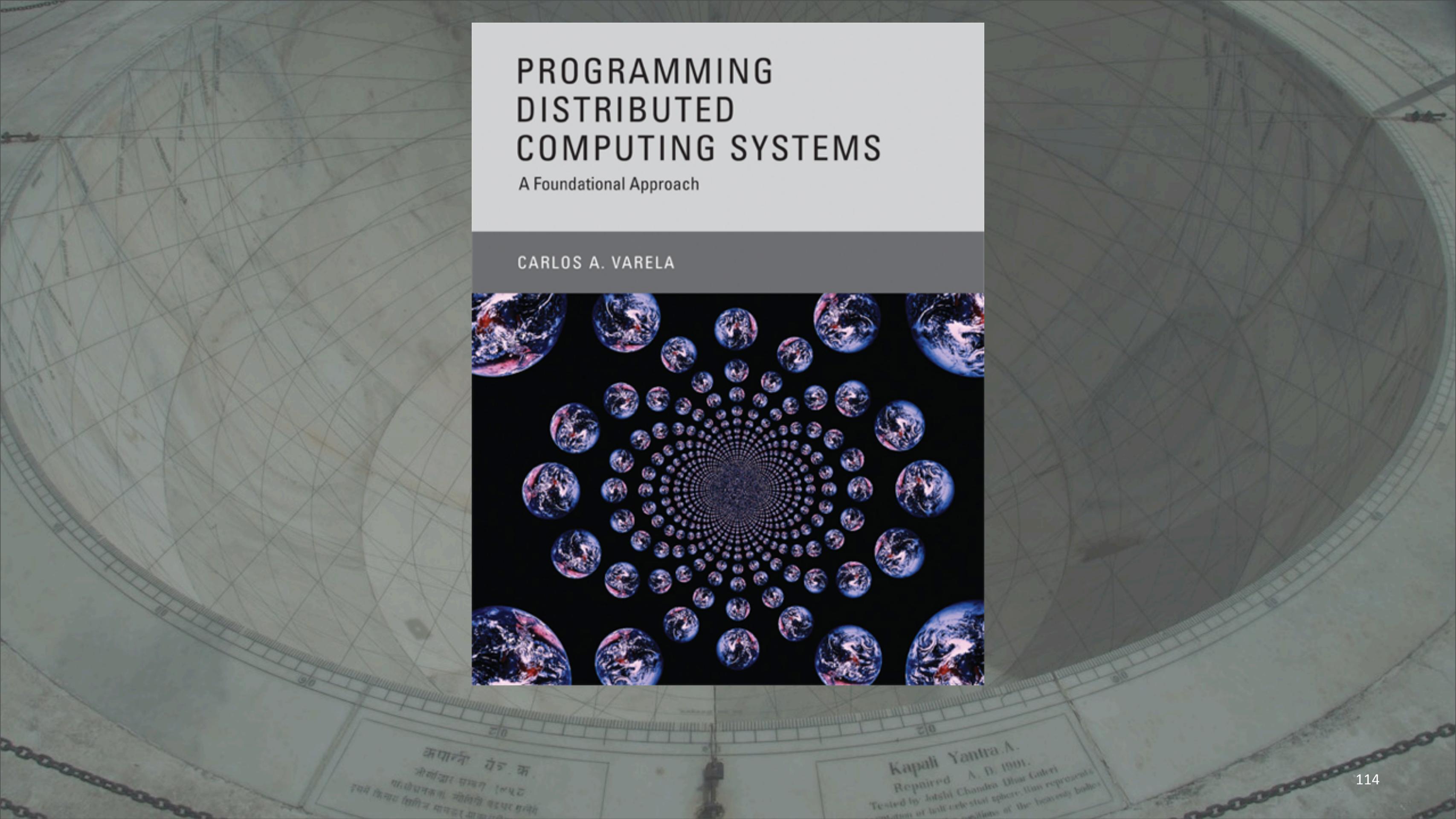
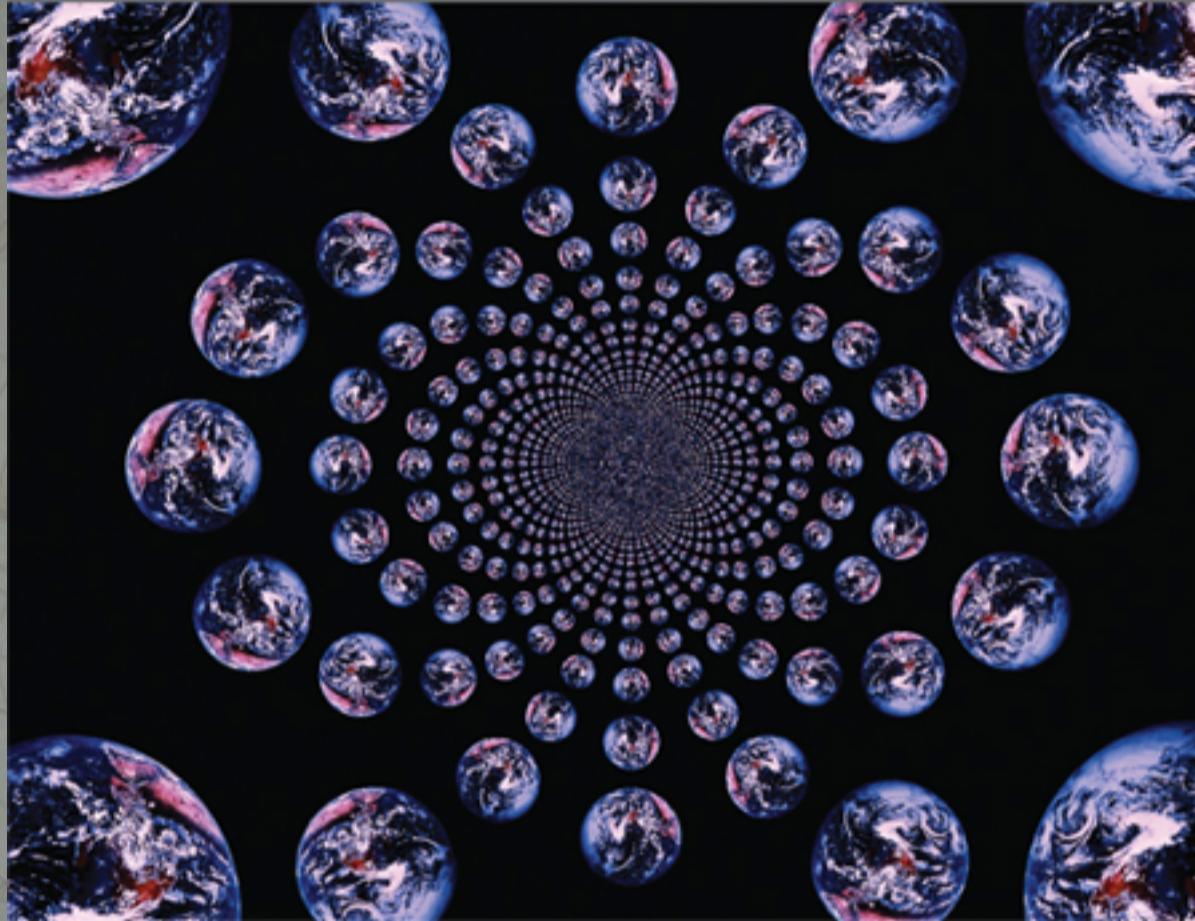
The original version is in print in April 2005 with Prentice-Hall (Pearson).
This version is made available for personal reference only. This version is
copyright (©) Pearson and Bill Roscoe.

Modern treatment of CSP. Roscoe helped transform the original CSP language into its more rigorous, process algebra form, which was influenced by Milner's Calculus of Communicating Systems work. This book's PDF is available free. The treatment is more accessible than Hoare's book.

PROGRAMMING DISTRIBUTED COMPUTING SYSTEMS

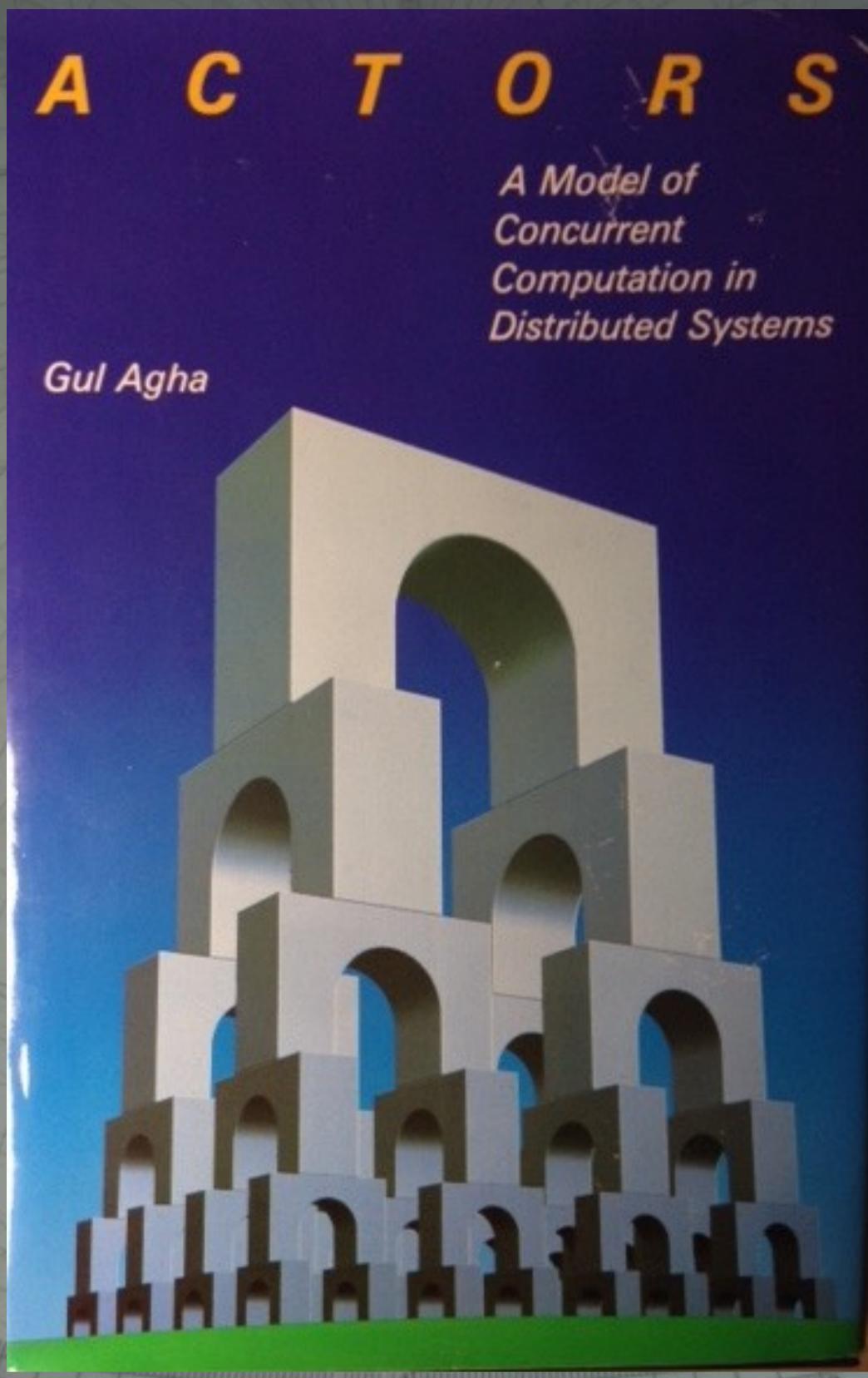
A Foundational Approach

CARLOS A. VARELA



Wednesday, March 18, 15

A survey of theoretical models of distributed computing, starting with a summary of lambda calculus, then discussing the pi, join, and ambient calculi. Also discusses the actor model. The treatment is somewhat dry and could use more discussion of real-world implementations of these ideas, such as the Actor model in Erlang and Akka.



Wednesday, March 18, 15

Gul Agha was a grad student at MIT during the 80s and worked on the actor model with Hewitt and others. This book is based on his dissertation.
It doesn't discuss error handling, actor supervision, etc. as these concepts .

His thesis, <http://dspace.mit.edu/handle/1721.1/6952>, the basis for his book,<http://mitpress.mit.edu/books/actors>

See also Paper for a survey course with Rajesh Karmani, <http://www.cs.ucla.edu/~palsberg/course/cs239/papers/karmani-agha.pdf>

Michel Raynal

Distributed Algorithms for Message-Passing Systems

 Springer

DISTRIBUTED ALGORITHMS

AN INTUITIVE APPROACH

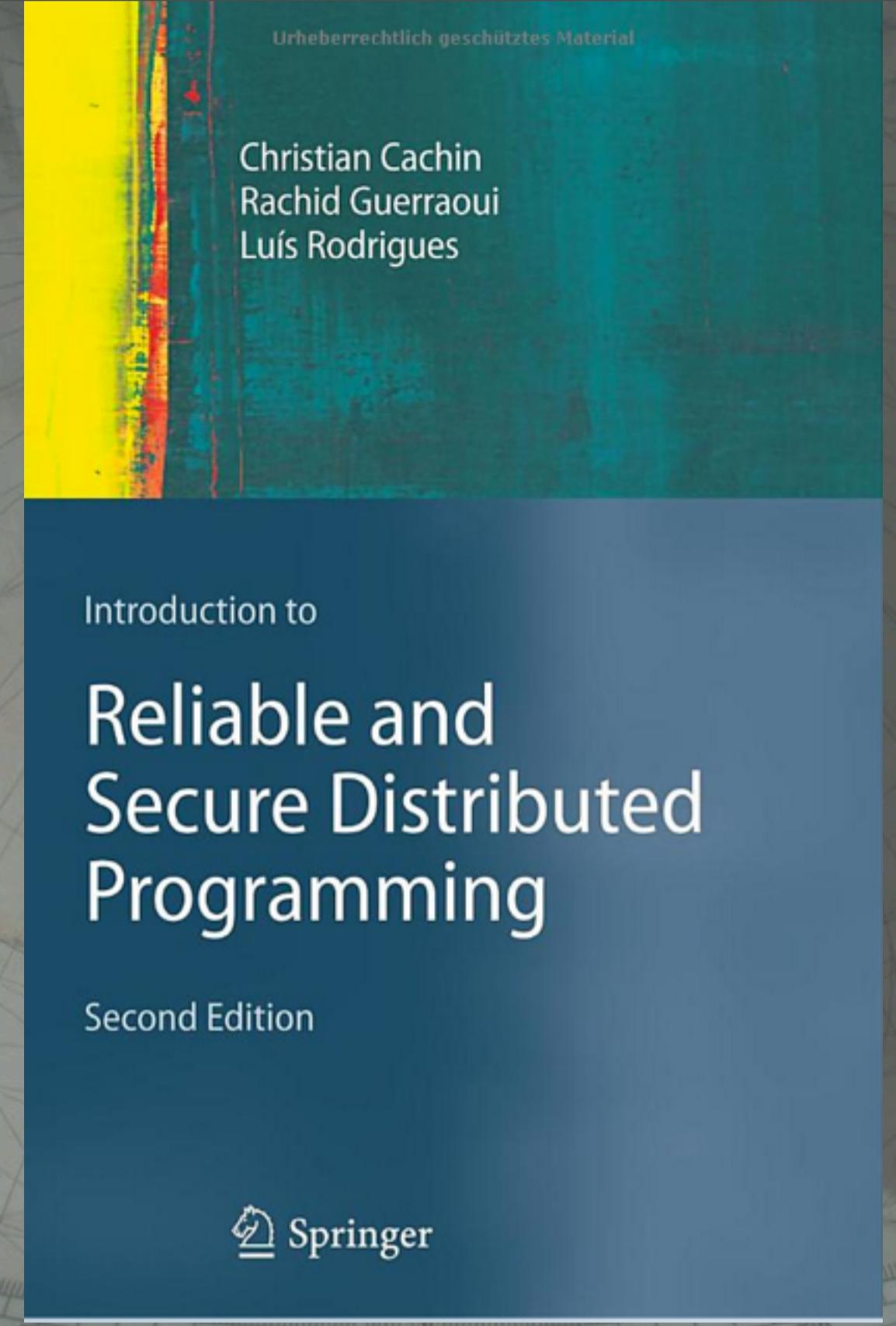


WAN FOKKINK

Copyrighted material

Wednesday, March 18, 15

A less comprehensive and formal, but more intuitive approach to fundamental algorithms.



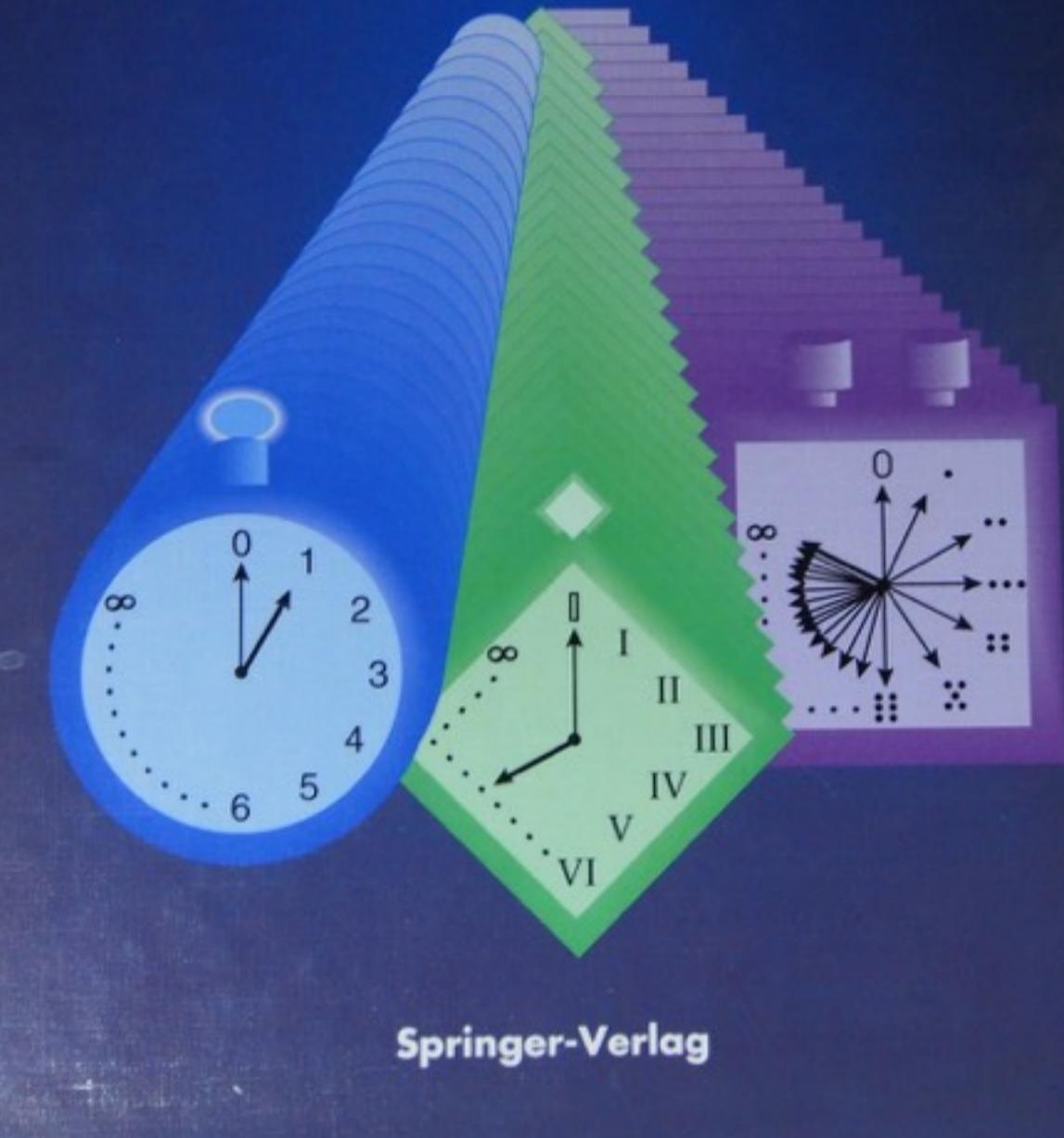
Wednesday, March 18, 15

Comprehensive and somewhat formal like Raynal's book, but more focused on modeling common failures in real systems.

Zohar Manna
Amir Pnueli

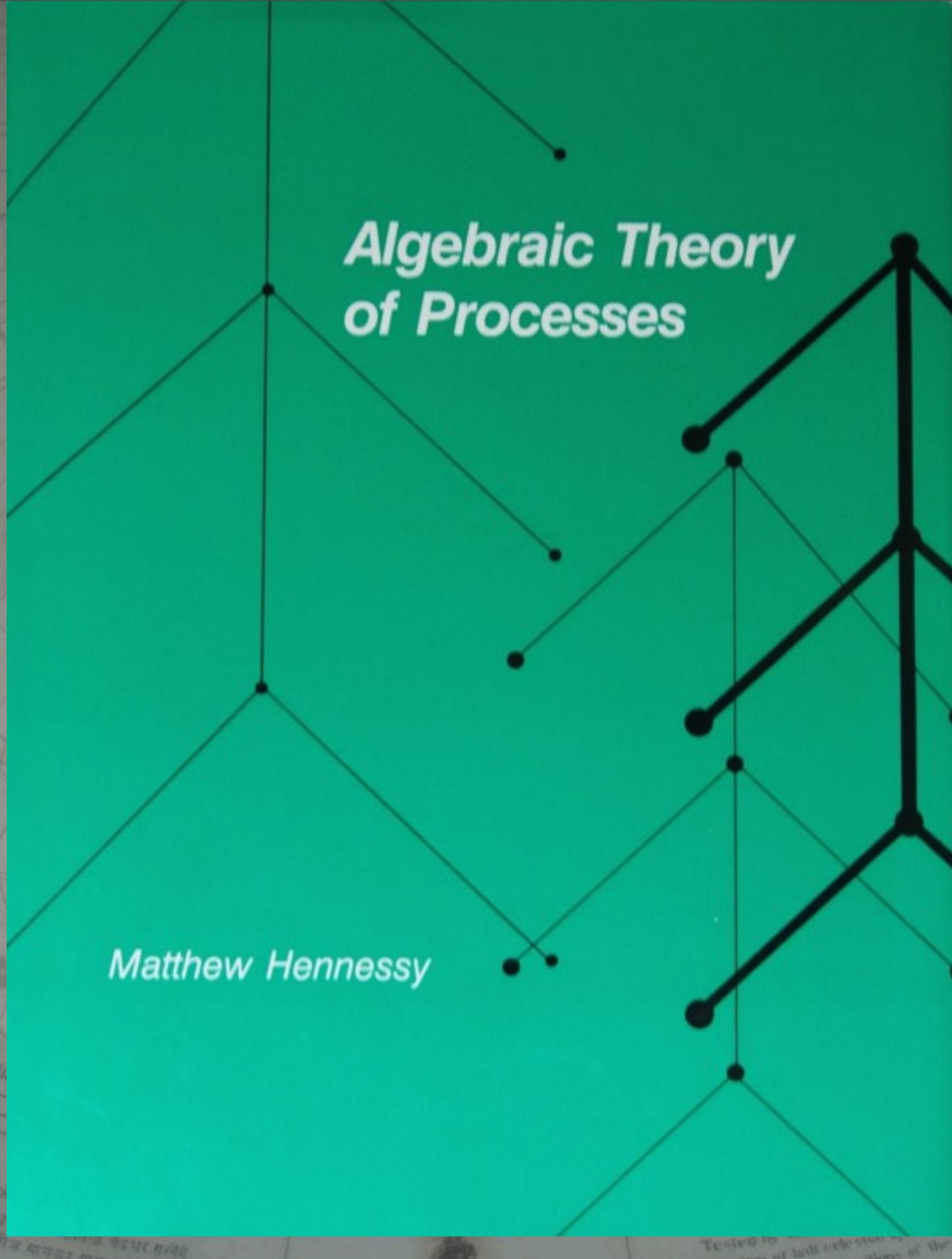
The Temporal Logic of Reactive and Concurrent Systems

•Specification•



Wednesday, March 18, 15

1992: Yes, “Reactive” isn’t new ;) This book is lays out a theoretical model for specifying and proving “reactive” concurrent systems based on temporal logic. While its goal is to prevent logic errors, It doesn’t discuss handling failures from environmental or other external causes in great depth.



Wednesday, March 18, 15

1988: Another treatment of concurrency using algebra. It's not based on CSP, but it has similar constructs.

DISTRIBUTED COMPUTING through COMBINATORIAL TOPOLOGY



Copyrighted Material

Maurice Herlihy
Dmitry Kozlov
Sergio Rajsbaum

Wednesday, March 18, 15

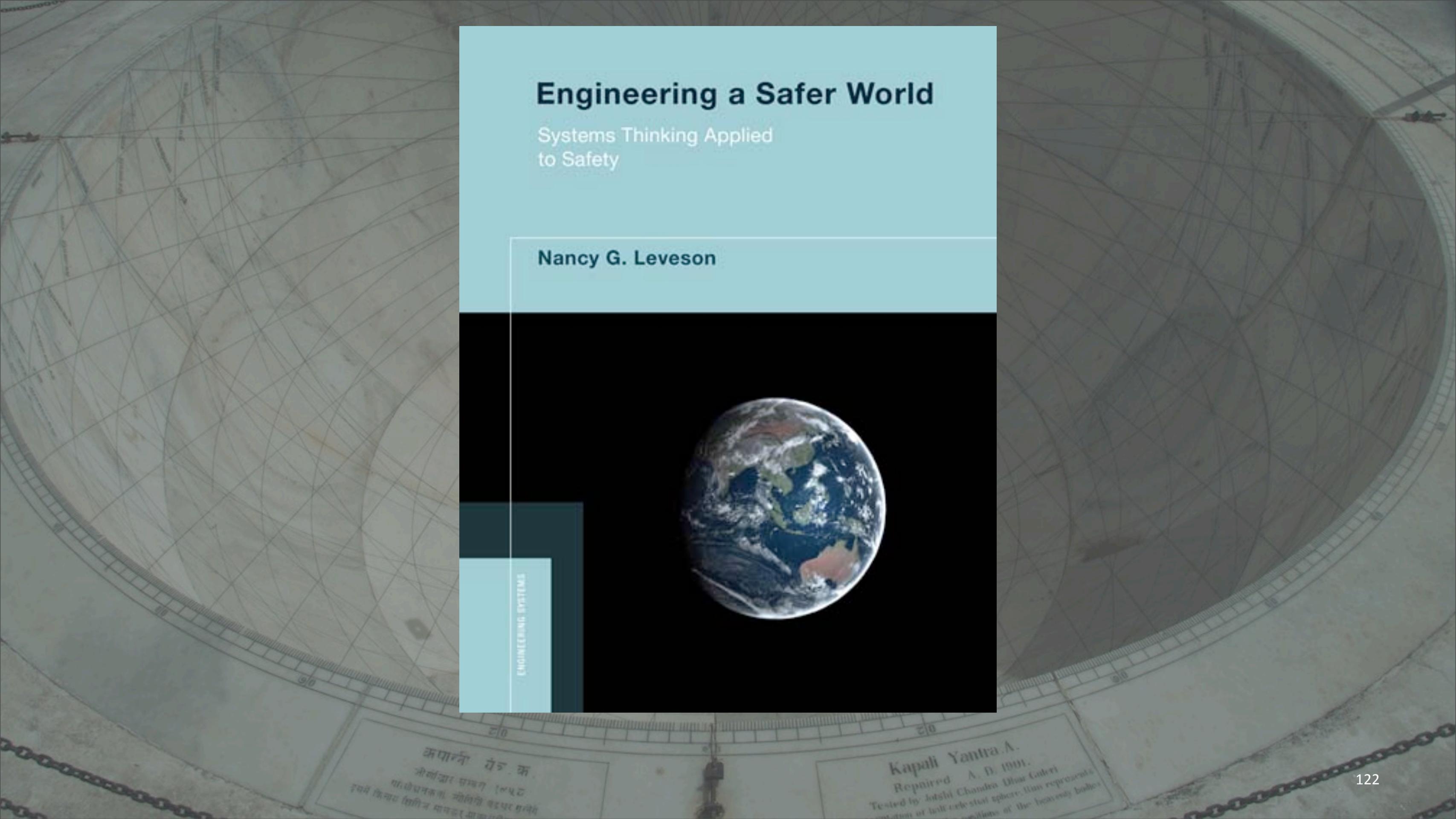
A recent text that applies combinatorics (counting things) and topology (properties of geometric shapes) to the analysis of distributed systems. Aims to be pragmatic for real-world scenarios, like networks and other physical systems where failures are practical concerns.

Engineering a Safer World

Systems Thinking Applied
to Safety

Nancy G. Leveson

ENGINEERING SYSTEMS



Wednesday, March 18, 15

<http://mitpress.mit.edu/books/engineering-safer-world>

Farther afield, this book discusses safety concerns from a systems engineering perspective.

Others

- Rob Pike: Go Concurrency Patterns
 - <http://www.youtube.com/watch?v=f6kdp27YZs&feature=youtu.be>
- Comparison of Clojure Core Async and Go
 - <http://blog.drewolson.org/blog/2013/07/04/clojure-core-dot-async-and-go-a-code-comparison/>