

Error Handling in Reactive Systems

Dean Wampler, Ph.D.
Typesafe



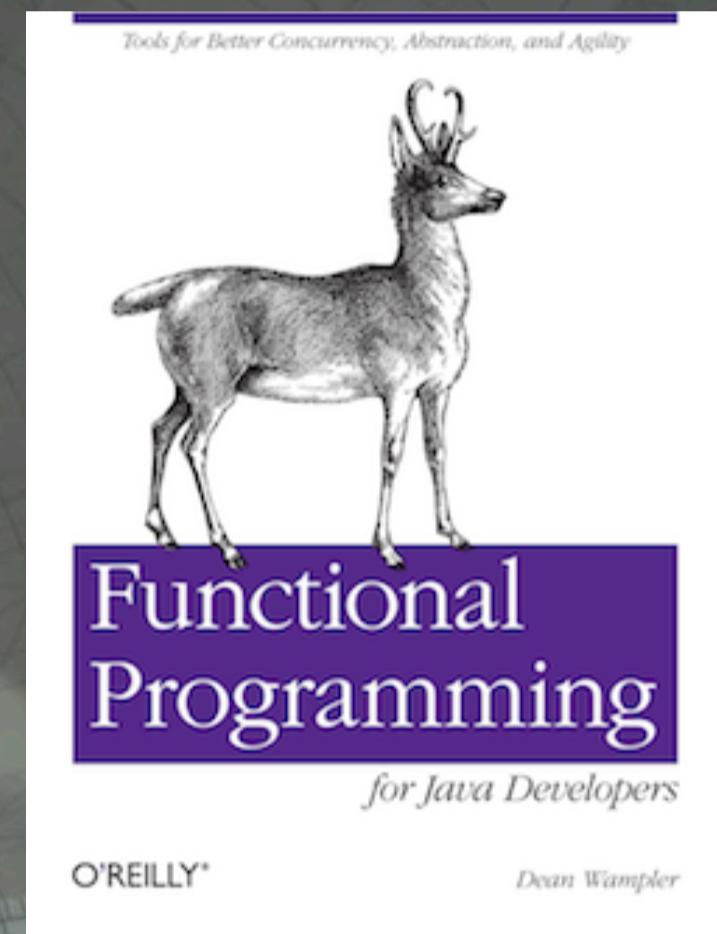
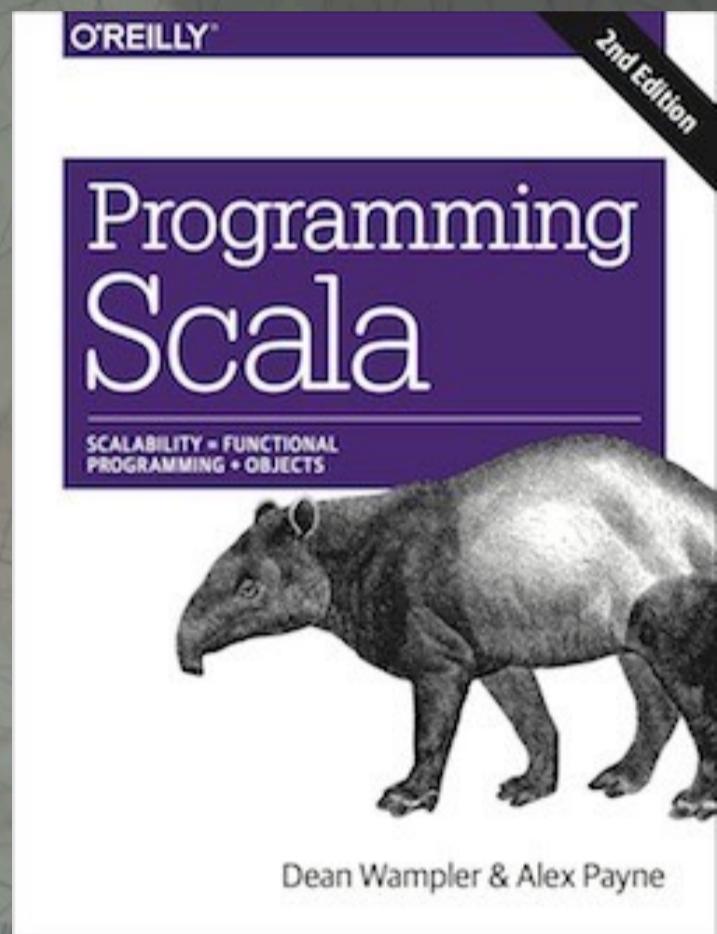
©Typesafe 2014-2015, All Rights Reserved

1

Wednesday, February 25, 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) 2012–2014, Dean Wampler. All Rights Reserved.

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



©Typesafe 2014-2015, All Rights Reserved

2

Typesafe Reactive Big Data

typesafe.com/reactive-big-data

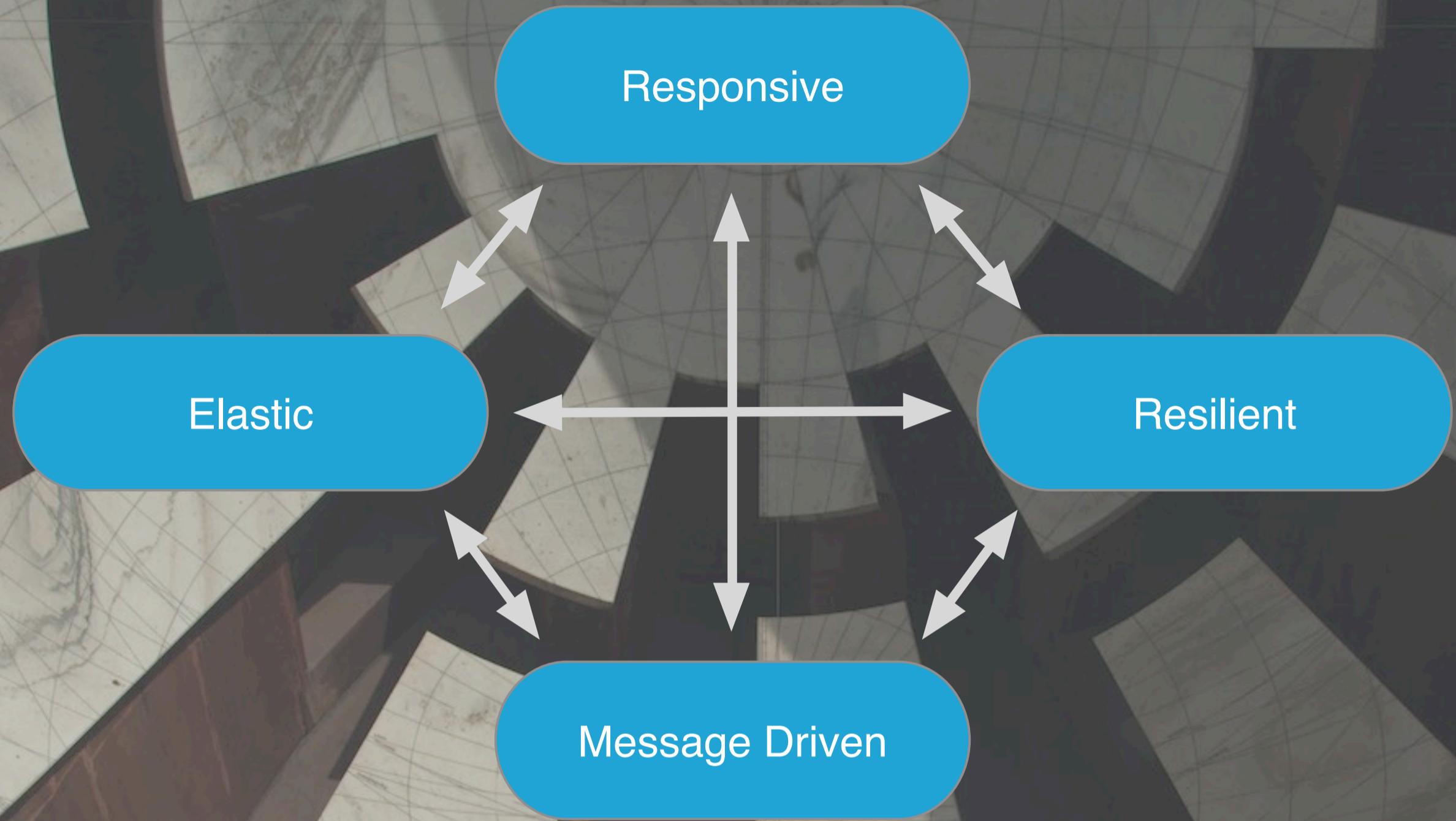


©Typesafe 2014-2015, All Rights Reserved

3

Wednesday, February 25, 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

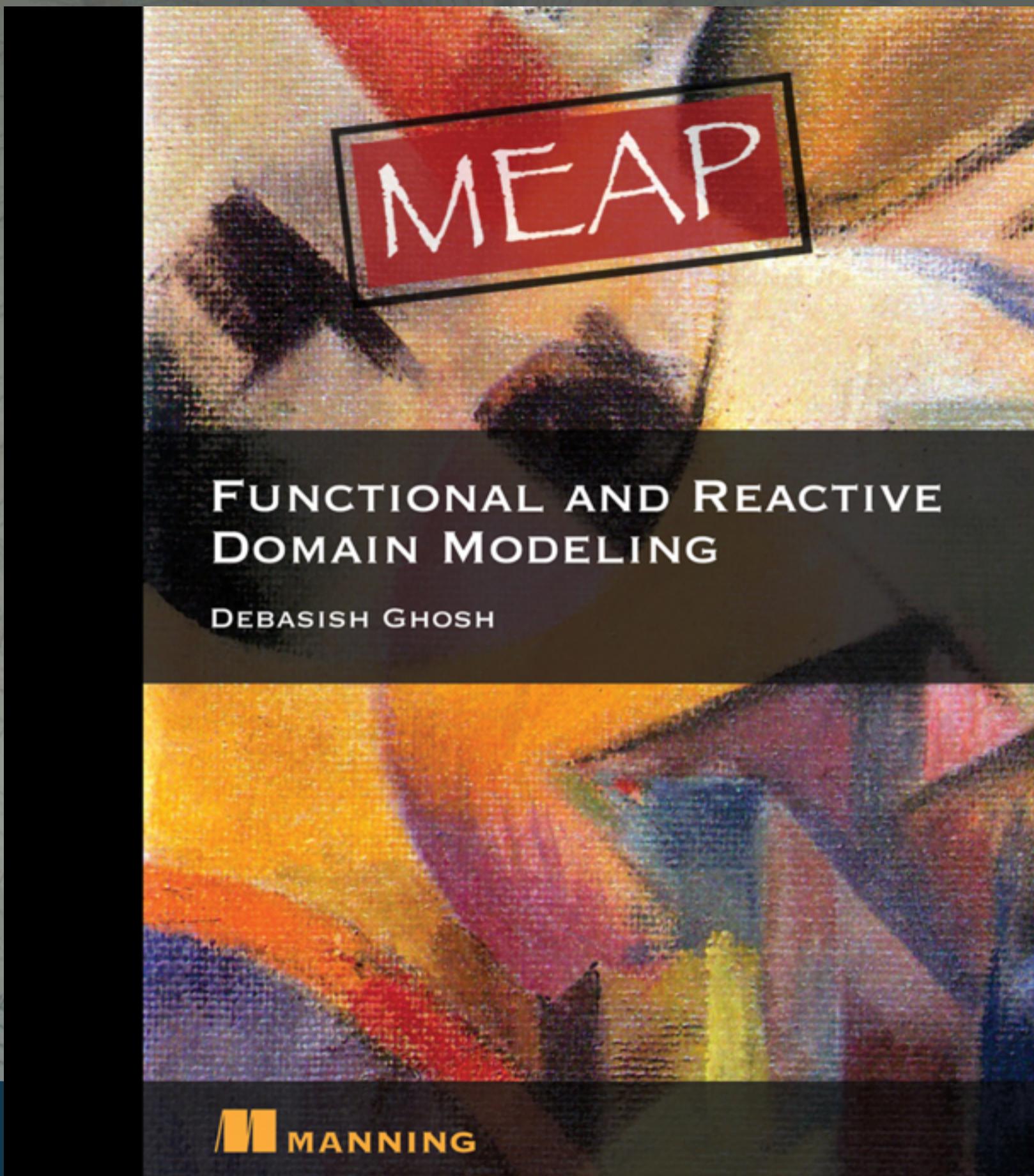
Resilience Driven

©Typesafe 2014-2015, All Rights Reserved

5

Wednesday, February 25, 15

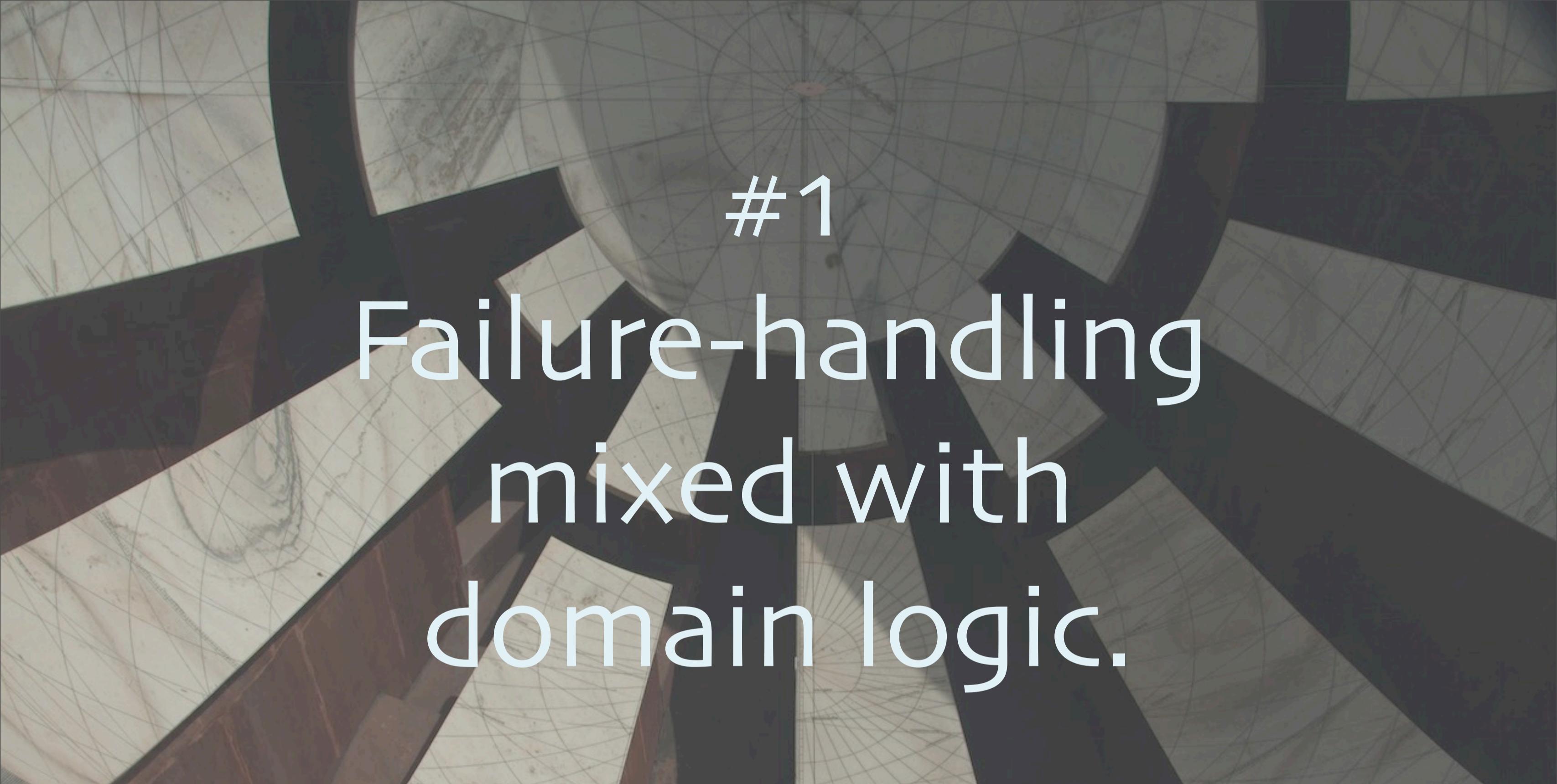
Truly resilient systems must make failures first class citizens, in some sense of the word, because they are inevitable when the systems are big enough and run long enough.



6

Wednesday, February 25, 15

I've structured this talk around some 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.



©Typesafe 2014-2015, All Rights Reserved

7

Wednesday, February 25, 15

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

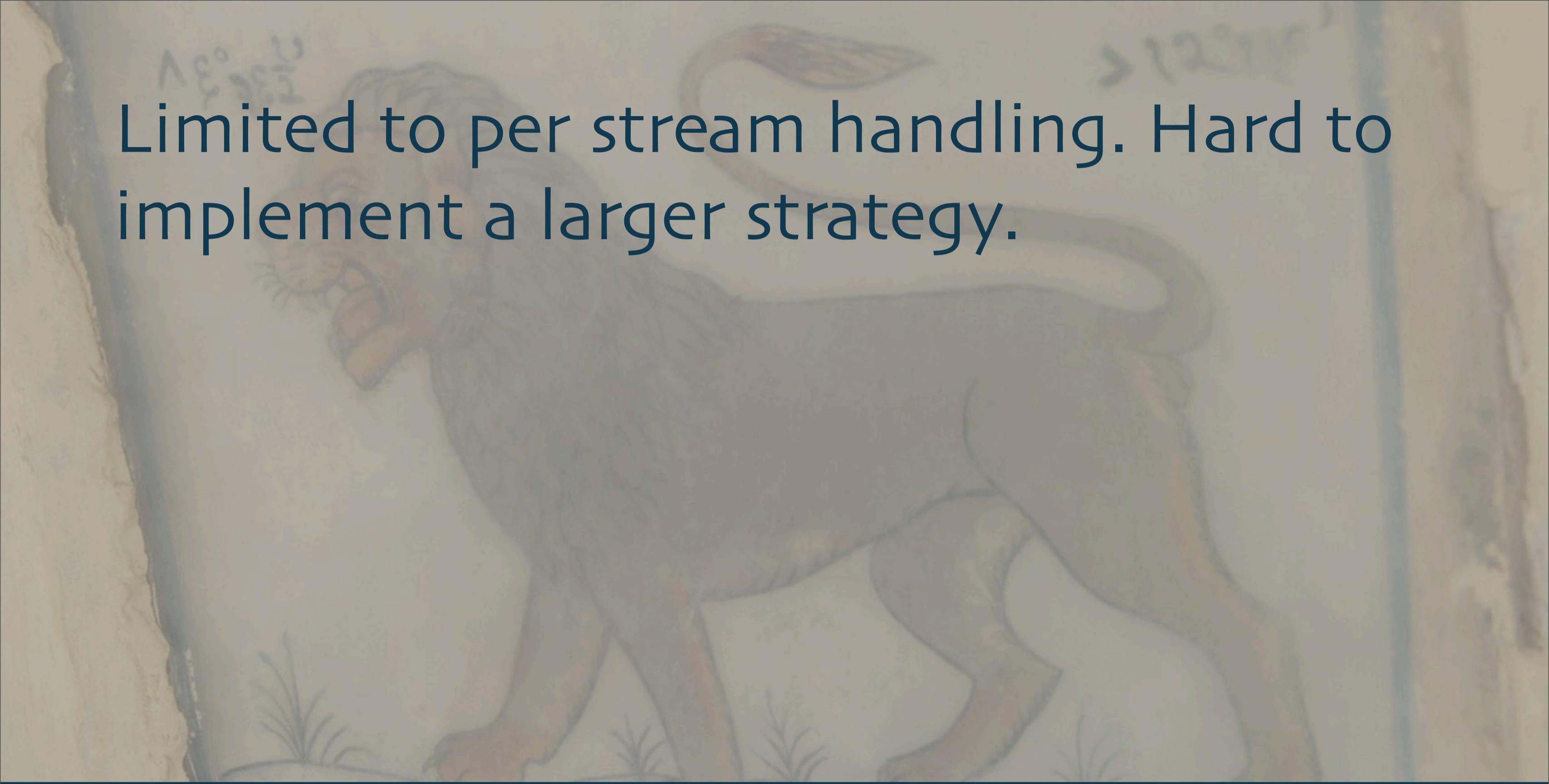
Best for narrowly-scoped errors.

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



©Typesafe 2014-2015, All Rights Reserved

8



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



©Typesafe 2014-2015, All Rights Reserved

9

Communicating Sequential Processes

Message passing
via channels



©Typesafe 2014-2015, All Rights Reserved

10

Wednesday, February 25, 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



©Typesafe 2014-2015, All Rights Reserved

11

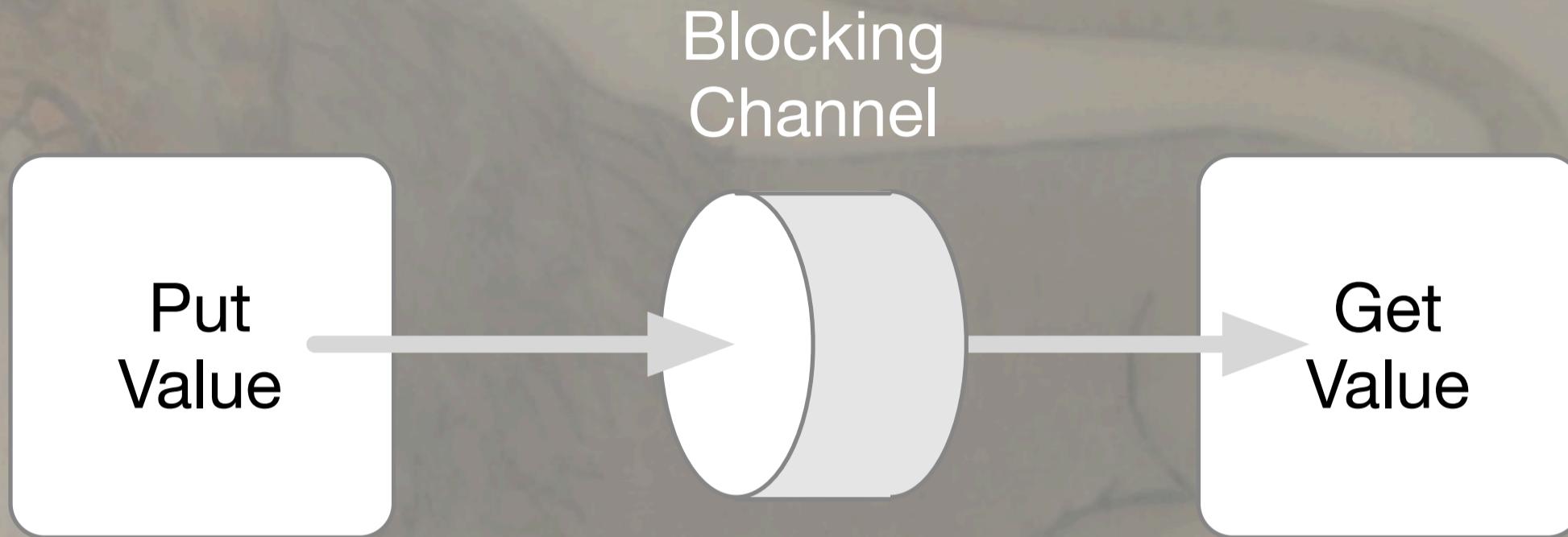
Wednesday, February 25, 15

<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



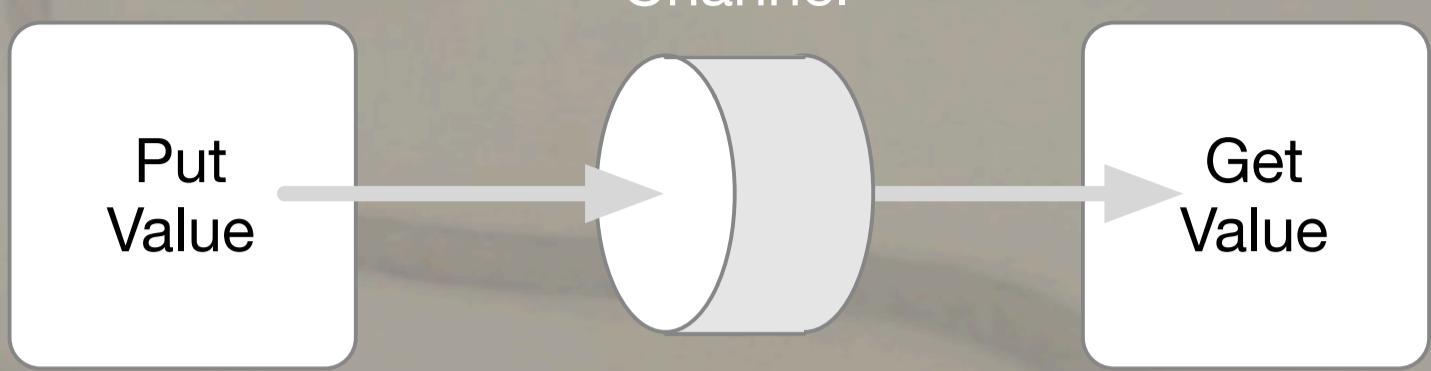


Wednesday, February 25, 15

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!



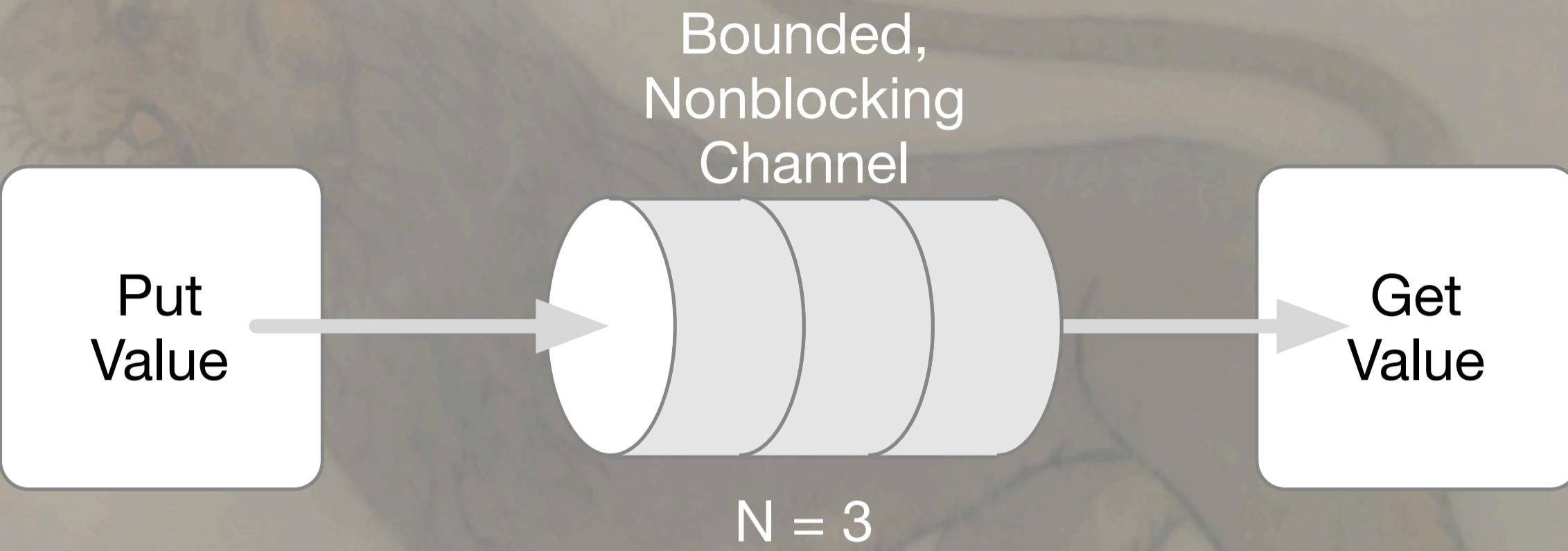
- Block on put if no one to get.
- Channel can be typed.
- Avoid passing mutable state!

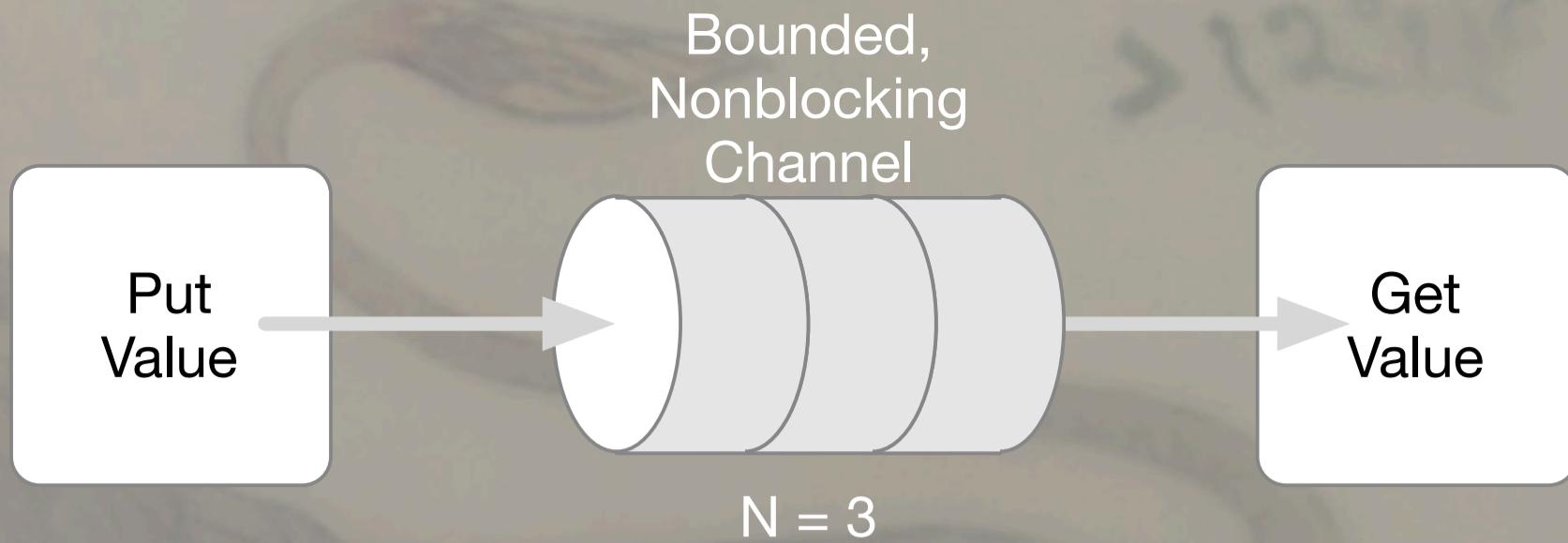
Wednesday, February 25, 15

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.

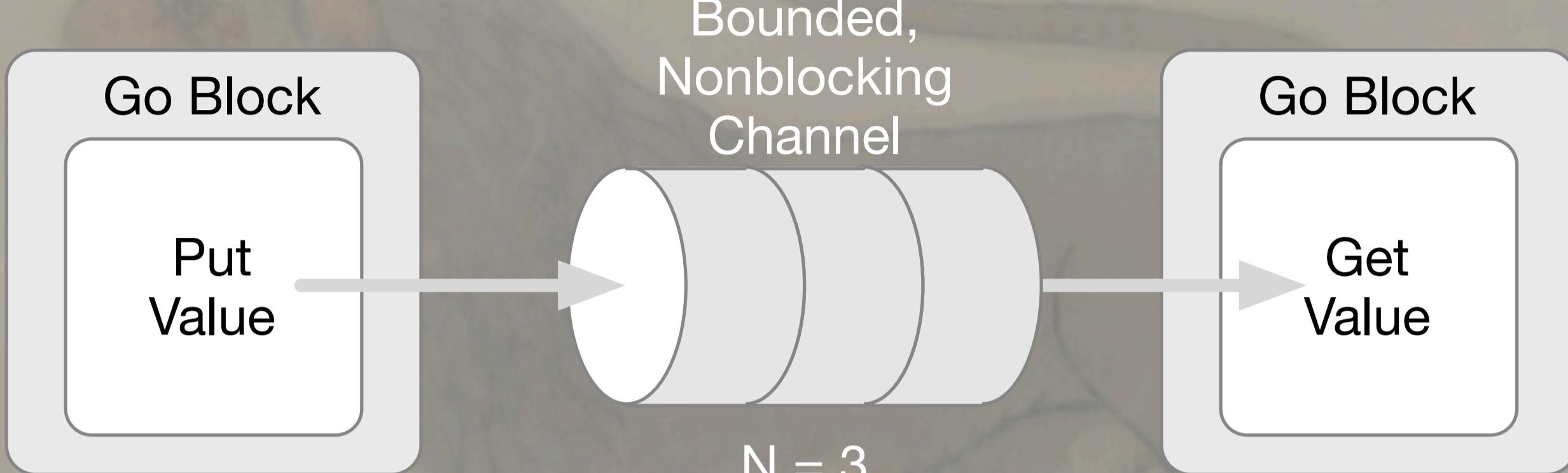
The channel can be typed (Go lang).

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).

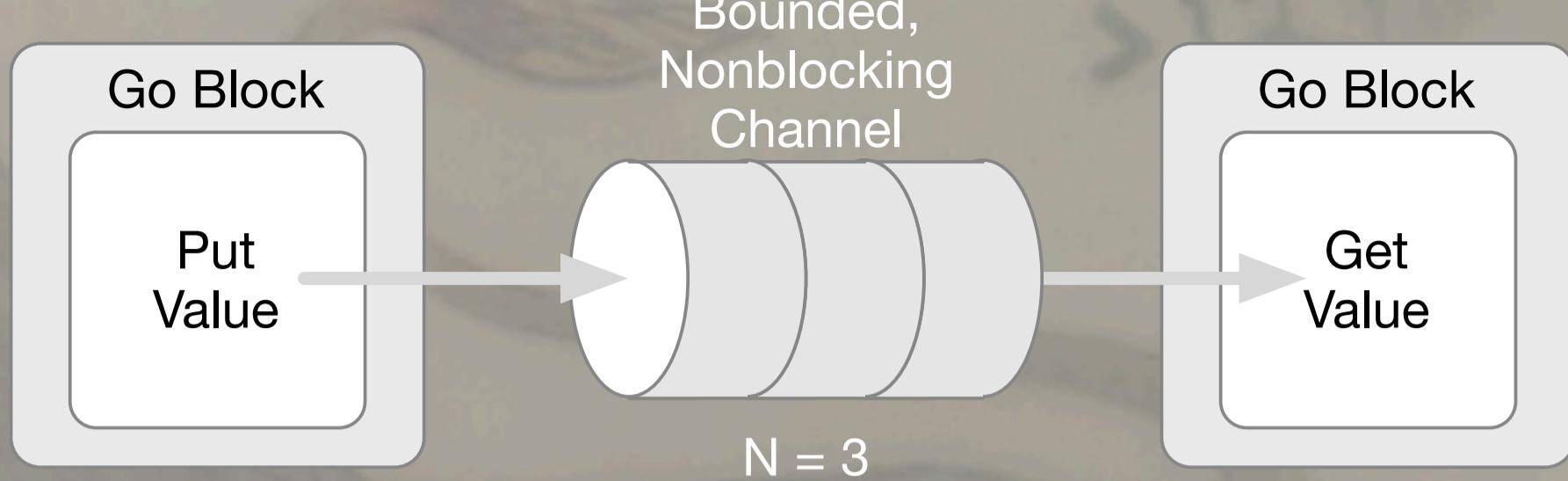


Wednesday, February 25, 15

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.



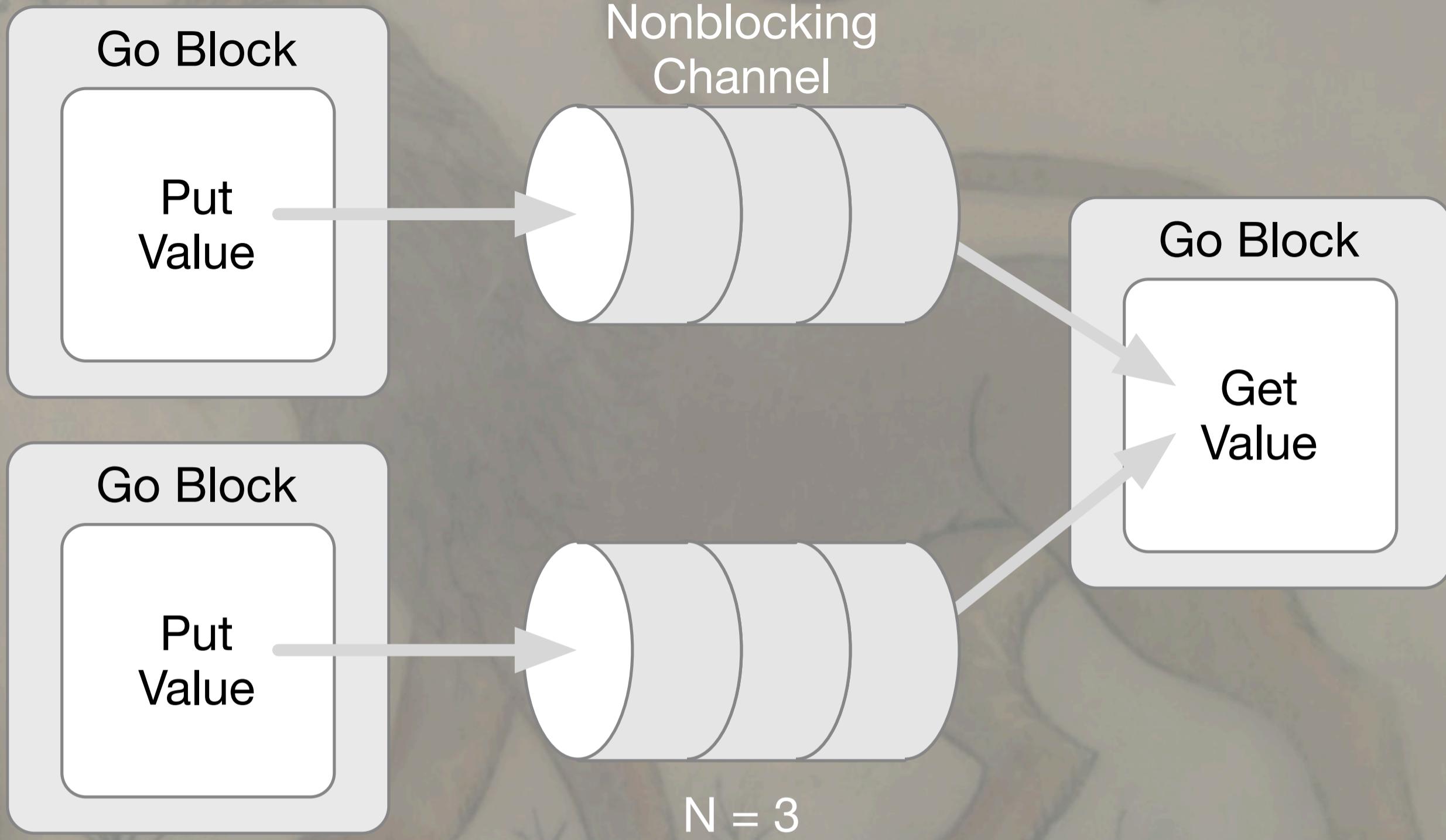
- Core Async: Go Blocks, Threads.
- Go: Go Routines.
- Analogous to futures.

Wednesday, February 25, 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".

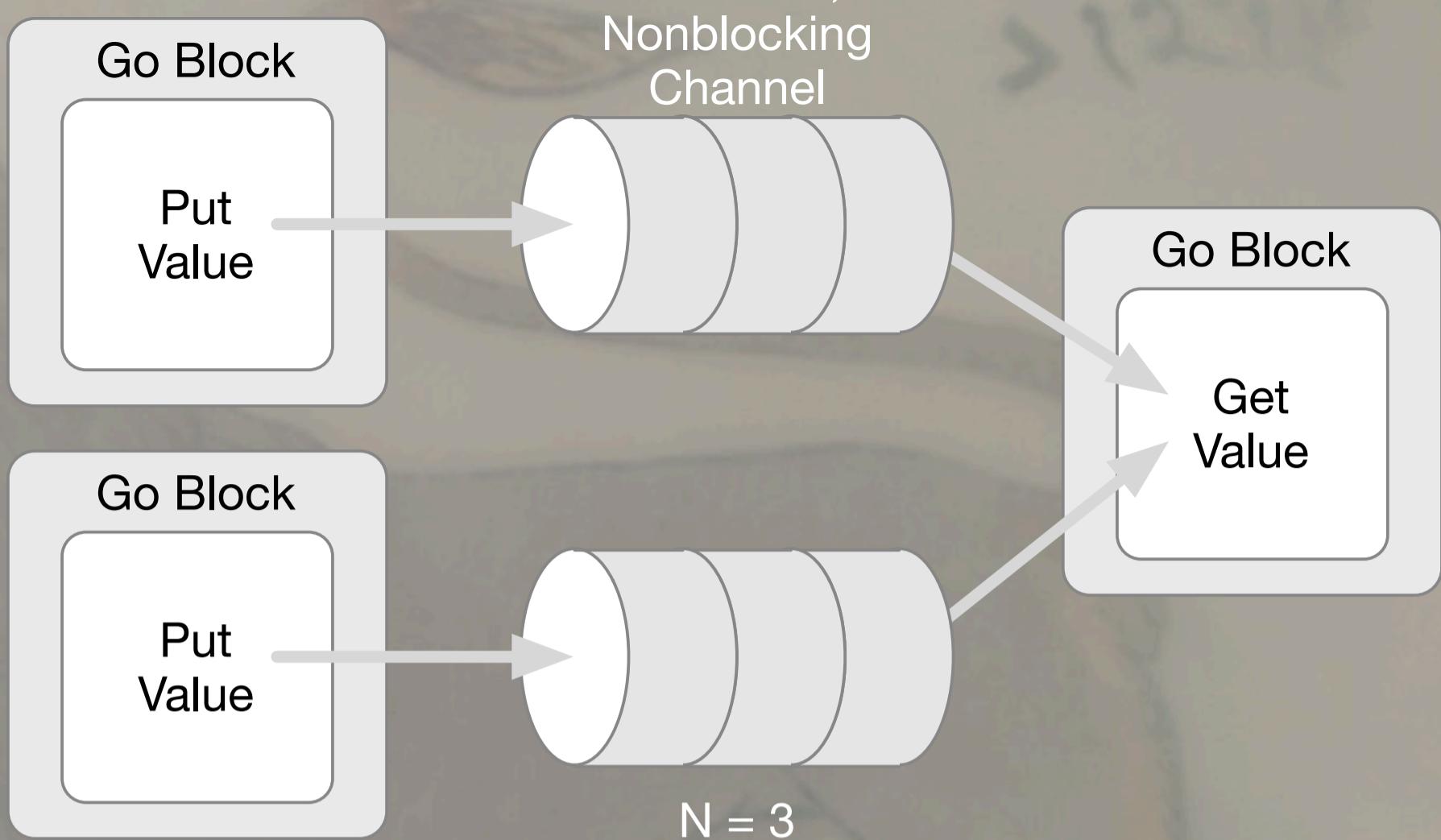
In all cases, they are analogous to Java/Scala futures.



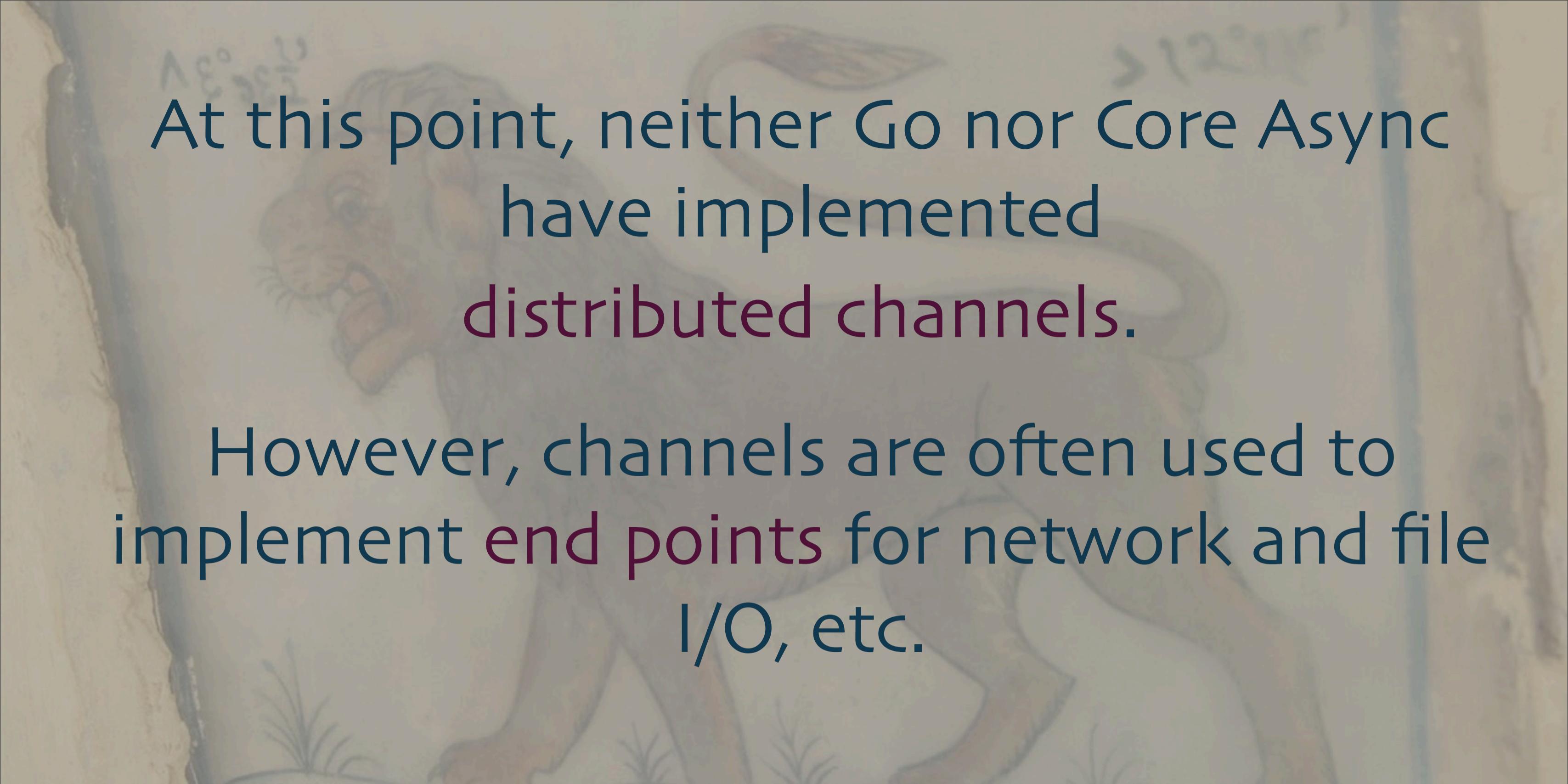
Wednesday, February 25, 15

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.



- Blocking or nonblocking.
- Like socket select.



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.



©Typesafe 2014-2015, All Rights Reserved

21

Wednesday, February 25, 15

In other words, no one has extended the channel formalism outside process boundaries (compare to Actors...), but channels are often used to handle blocking I/O, etc.

Failure Handling in Core Async



©Typesafe 2014-2015, All Rights Reserved

22

Wednesday, February 25, 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.



©Typesafe 2014-2015, All Rights Reserved

23

Wednesday, February 25, 15

Using this feature is almost always what you should do, because you have almost no other good options.

Which call stack?

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



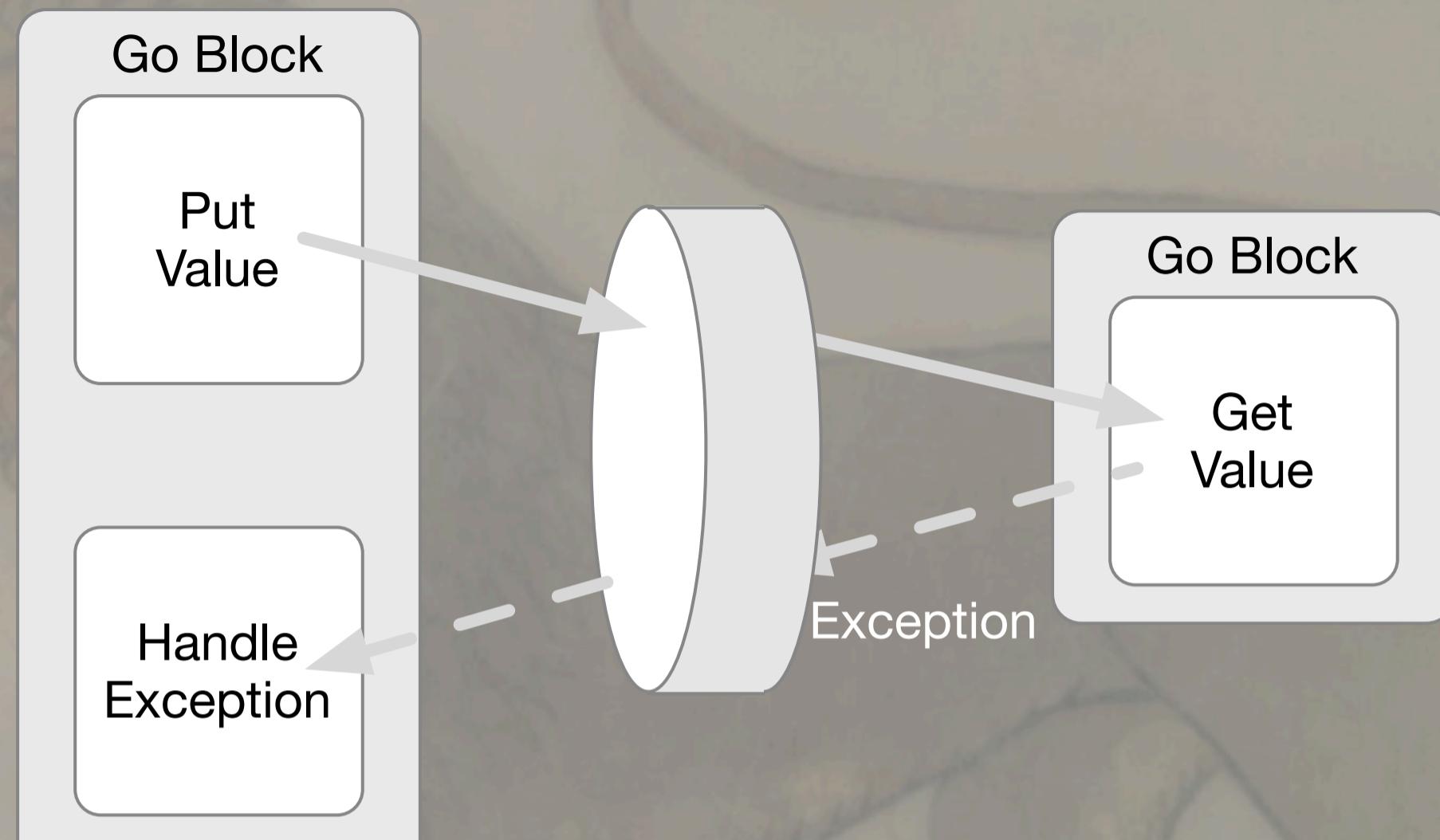
©Typesafe 2014-2015, All Rights Reserved

24

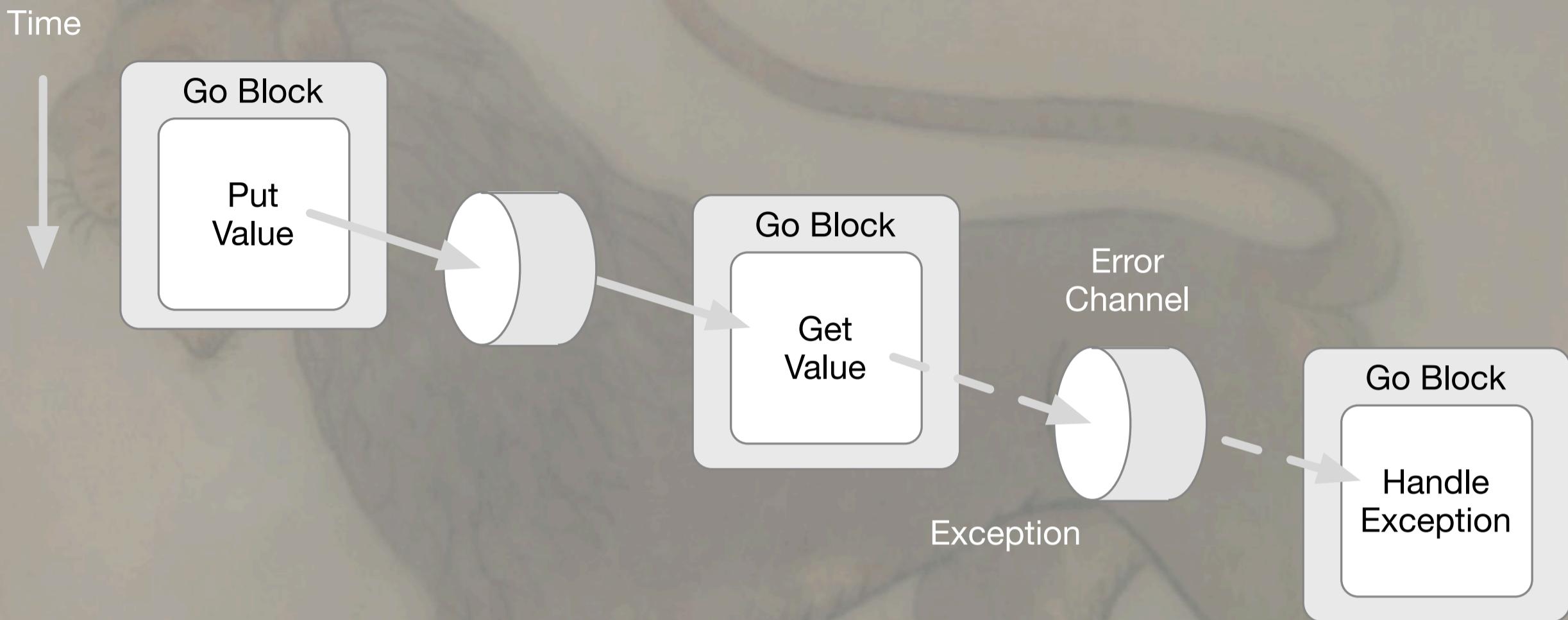
Wednesday, February 25, 15

Since it's not a distributed system, core async only needs to handle errors in a single process, but you still can have multiple threads.

Time



Propagate exceptions back through the channel.



Propagate exceptions to a special error channel.



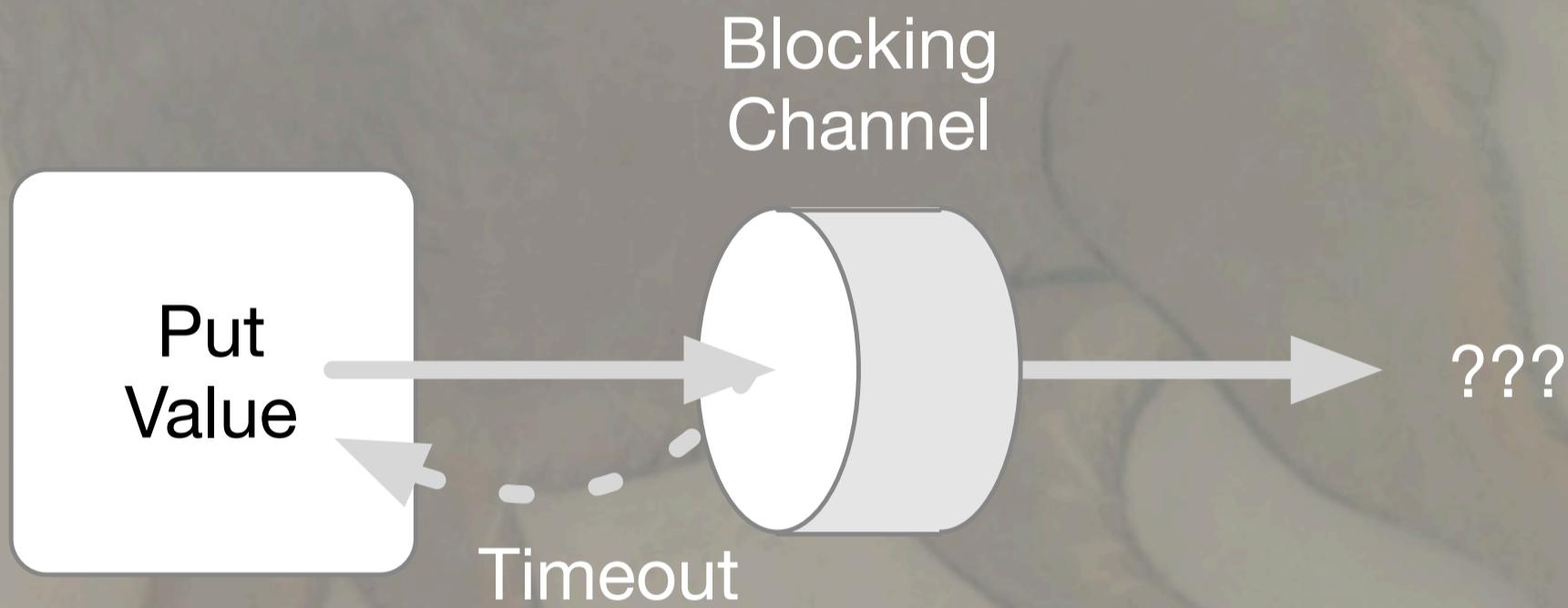
©Typesafe 2014-2015, All Rights Reserved

26

Wednesday, February 25, 15

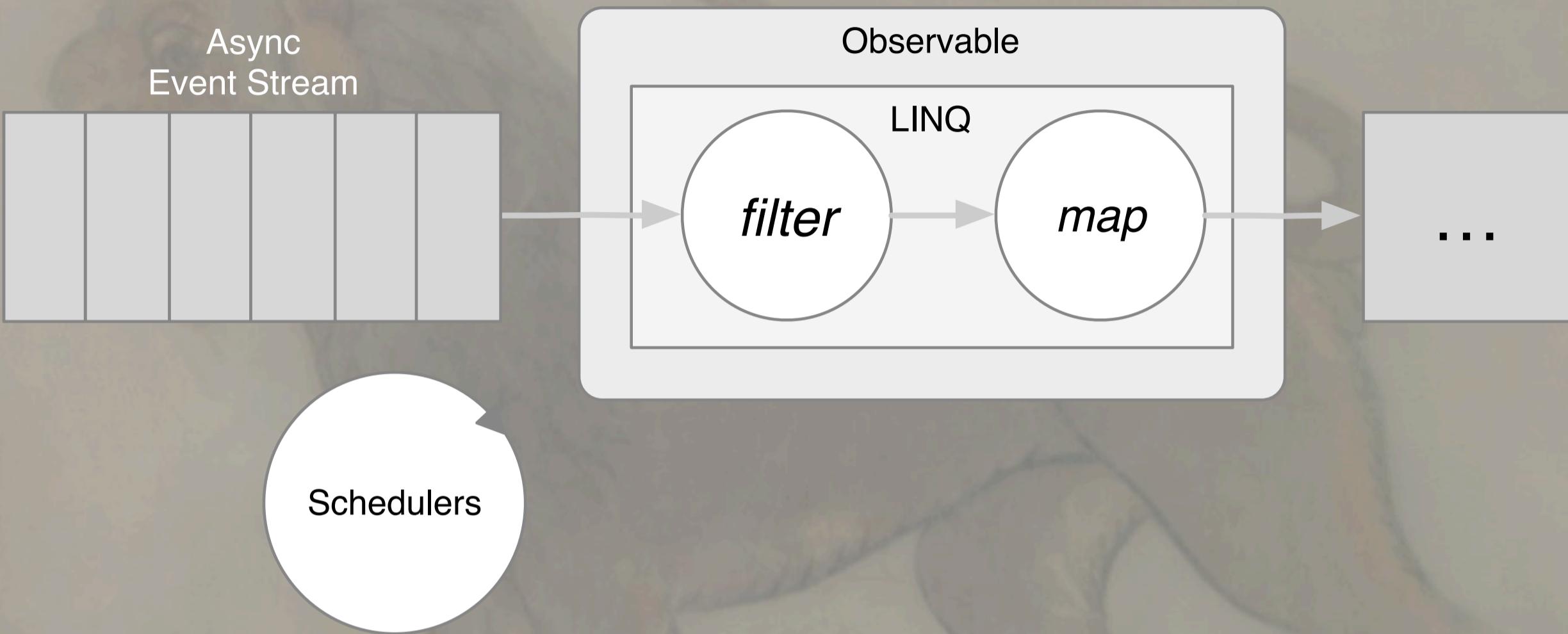
Another possible scheme is to send exceptions down a specialized error channel.

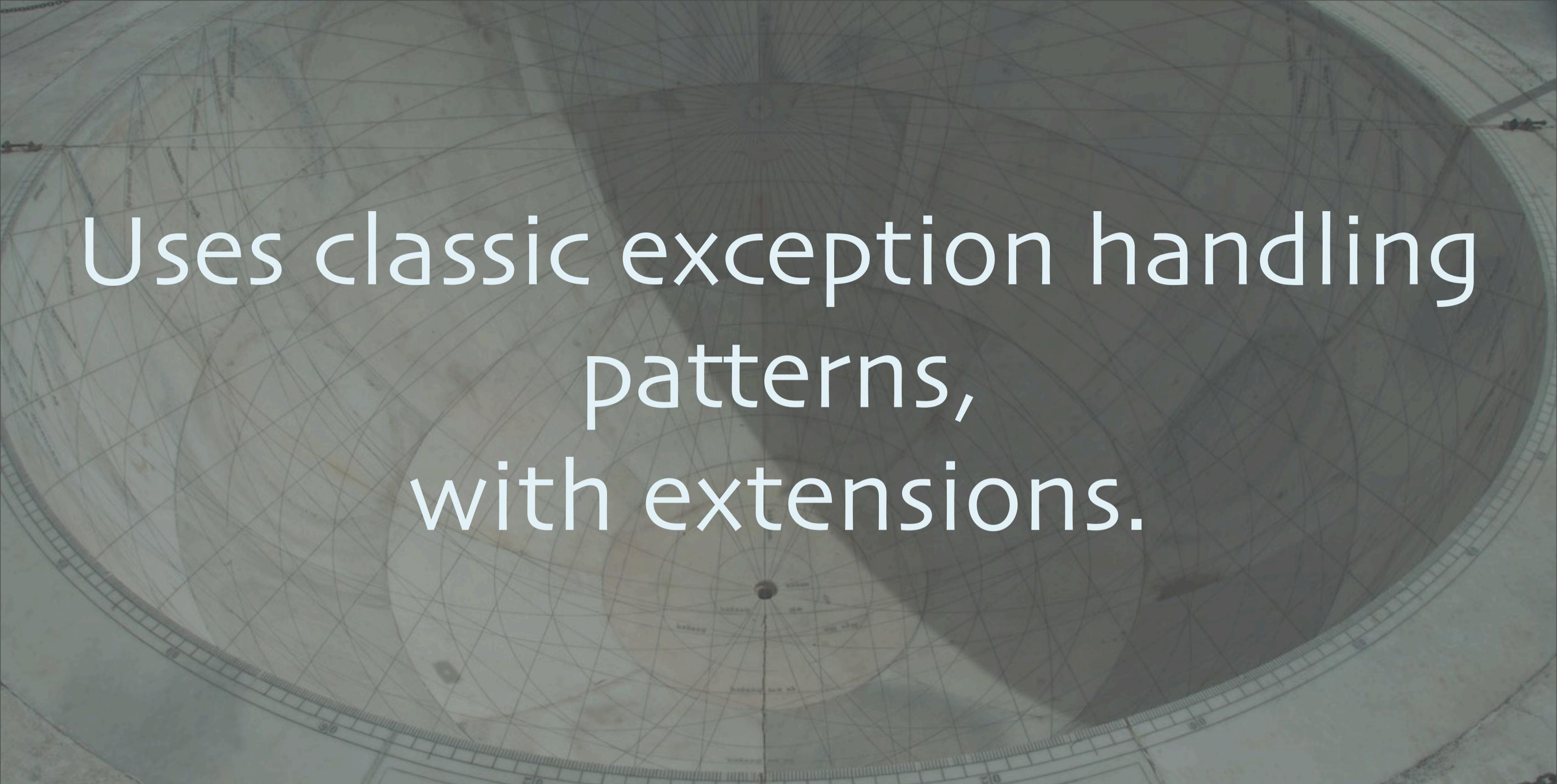
Deadlock are possible unless timeouts are used.



Reactive Extensions







Uses classic exception handling
patterns,
with extensions.



©Typesafe 2014-2015, All Rights Reserved

30

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 Exception("error"));  
// continue with otherStream.
```



©Typesafe 2014-2015, All Rights Reserved

31

Wednesday, February 25, 15

Adapted from http://www.introtorx.com/content/v1.0.10621.0/11_AdvancedErrorHandler.html

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

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



©Typesafe 2014-2015, All Rights Reserved

32

Wednesday, February 25, 15

Adapted from http://www.introtorx.com/content/v1.0.10621.0/11_AdvancedErrorHandler.html

There is also a Finally method.

Analogous to

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

clauses.



©Typesafe 2014-2015, All Rights Reserved

33

Wednesday, February 25, 15

Adapted from http://www.introtorx.com/content/v1.0.10621.0/11_AdvancedErrorHandler.html

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) {...}  
...
```



©Typesafe 2014-2015, All Rights Reserved

34

Wednesday, February 25, 15

Adapted from http://www.introtorx.com/content/v1.0.10621.0/11_AdvancedErrorHandler.html

2 of the 3 variants.

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();  
}
```



©Typesafe 2014-2015, All Rights Reserved

35

Wednesday, February 25, 15

Adapted from http://www.introtorx.com/content/v1.0.10621.0/11_AdvancedErrorHandler.html

CSP & Rx: Failure management is per-stream, mixed with domain logic.



©Typesafe 2014-2015, All Rights Reserved

36

Wednesday, February 25, 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.



©Typesafe 2014-2015, All Rights Reserved

37

Wednesday, February 25, 15

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



Reactive Streams



©Typesafe 2014-2015, All Rights Reserved

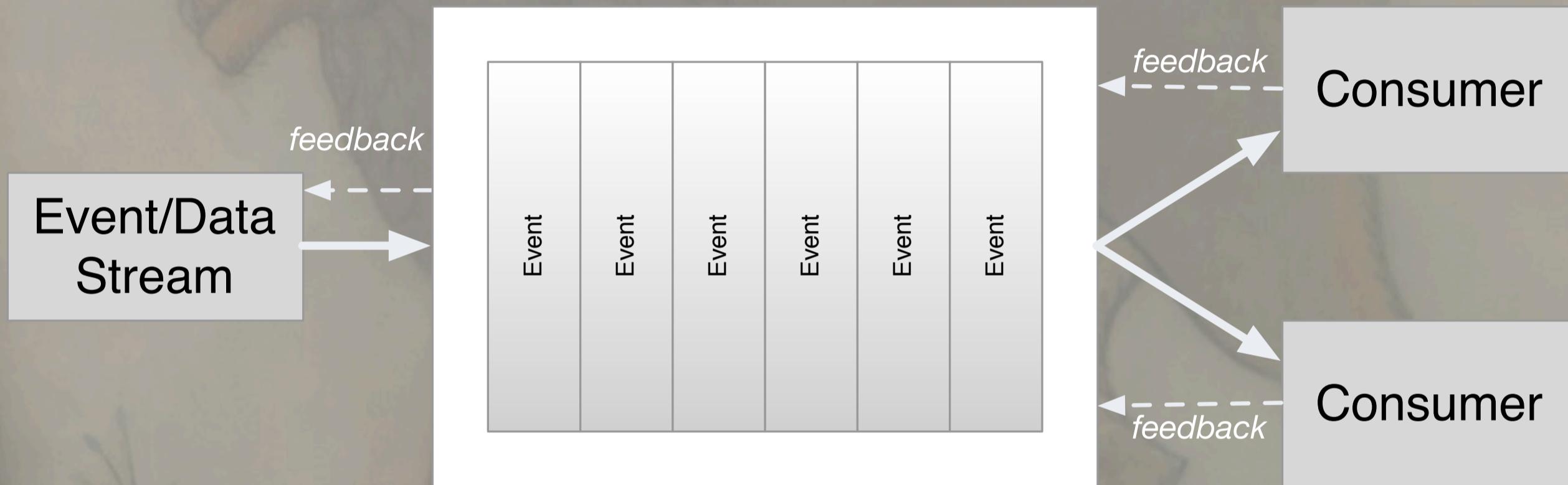
38

Wednesday, February 25, 15

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

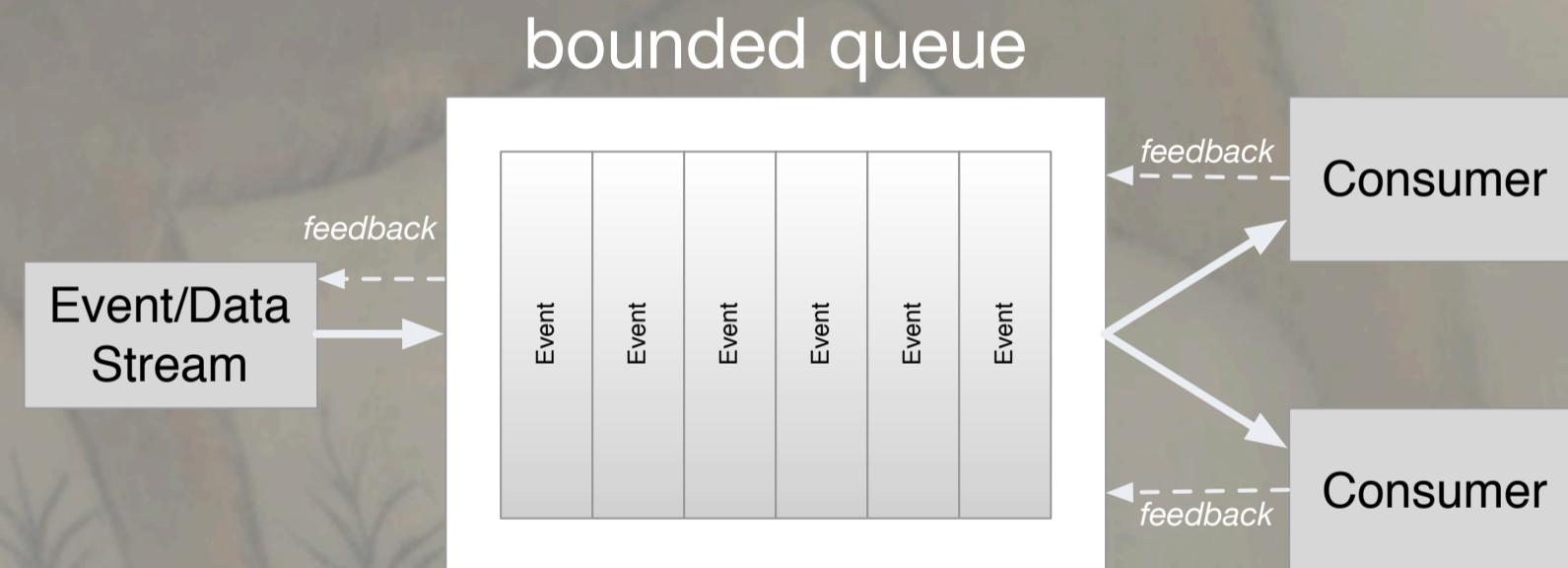
Reactive Streams

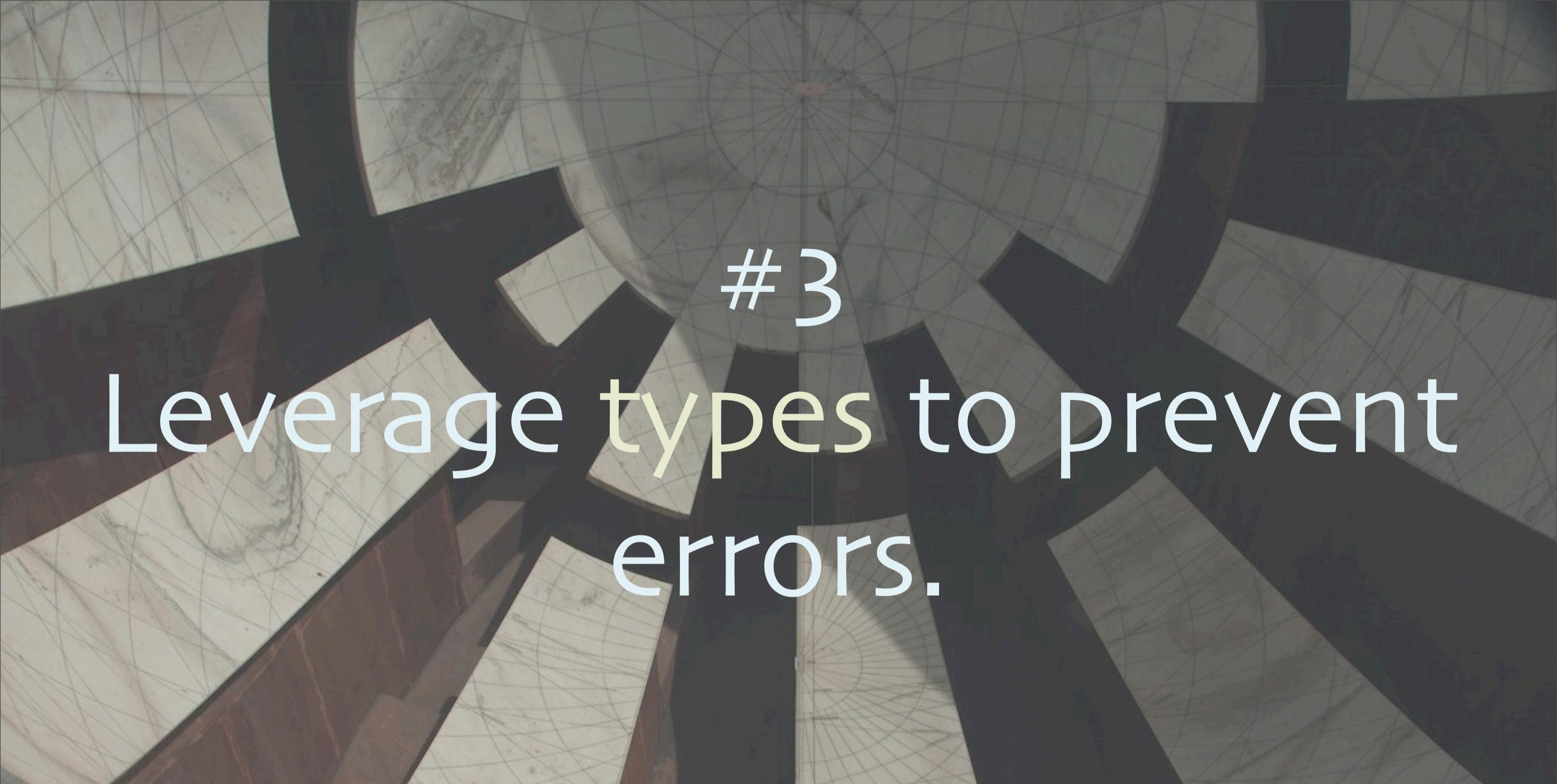
bounded queue



Reactive Streams

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





#3

Leverage types to prevent errors.



©Typesafe 2014-2015, All Rights Reserved

41

Wednesday, February 25, 15

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

Express what's really happening with types.



©Typesafe 2014-2015, All Rights Reserved

42

Wednesday, February 25, 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")  
    // raise exceptions for bad records  
    new Order(...)  
  }  
}
```



©Typesafe 2014-2015, All Rights Reserved

43

Wednesday, February 25, 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... */ }  
...
```



©Typesafe 2014-2015, All Rights Reserved

44

Wednesday, February 25, 15

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

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
```



©Typesafe 2014-2015, All Rights Reserved

45

Wednesday, February 25, 15

Futures can be sequenced “monadically”, so our code has a nice synchronous feel to it, but we can exploit async. execution.

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")  
}
```

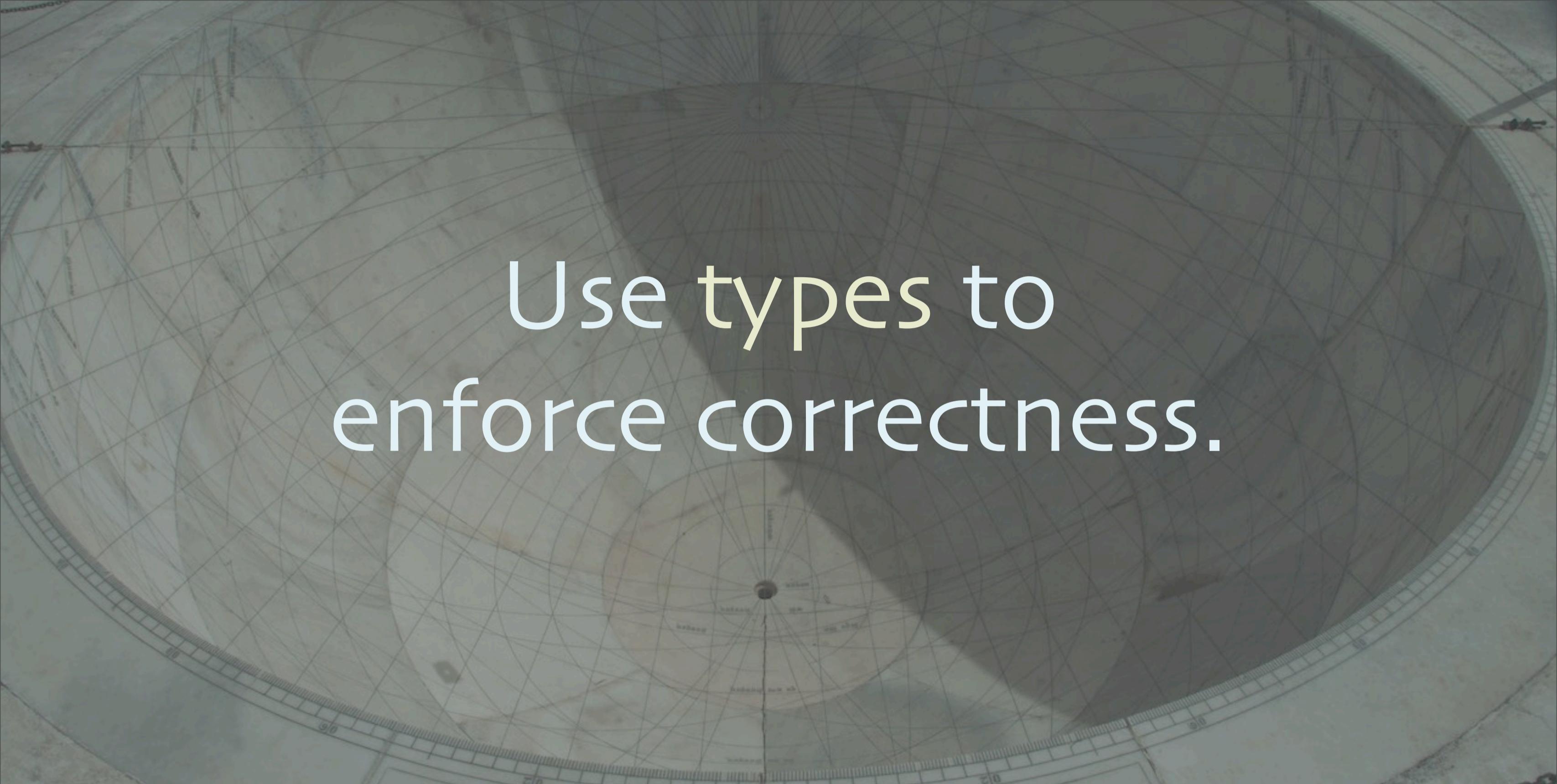


©Typesafe 2014-2015, All Rights Reserved

46

Wednesday, February 25, 15

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



Use types to enforce correctness.



©Typesafe 2014-2015, All Rights Reserved

47

Wednesday, February 25, 15

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



Functional Reactive Programming



©Typesafe 2014-2015, All Rights Reserved

48

Wednesday, February 25, 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.](#)



©Typesafe 2014-2015, All Rights Reserved

49

Wednesday, February 25, 15

Draw a line on a UI from the initial point to the current mouse point, as the mouse moves.

This API is from a research paper. I could have used Elm (FRP for JavaScript) or one of the Haskell FRP APIs (where FRP was pioneered), but this DSL is reasonably easy to understand. Here, we have a stream of data points, so it resembles Rx in its concepts.

Can you declaratively prevent errors?

Sculthorpe and Nilsson, Safe functional reactive programming through dependent types

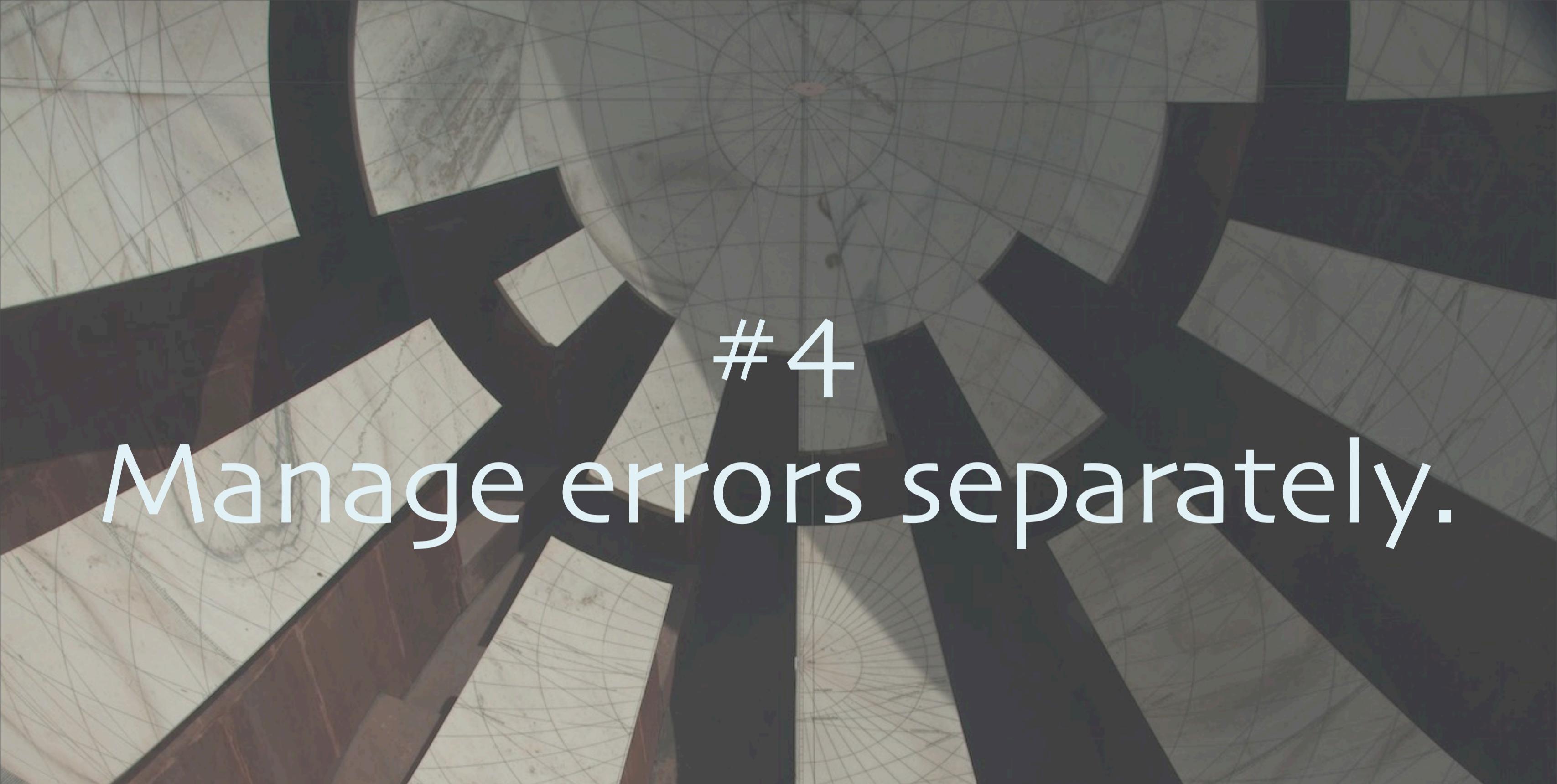


©Typesafe 2014-2015, All Rights Reserved

50

Wednesday, February 25, 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.



©Typesafe 2014-2015, All Rights Reserved

51

Wednesday, February 25, 15

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

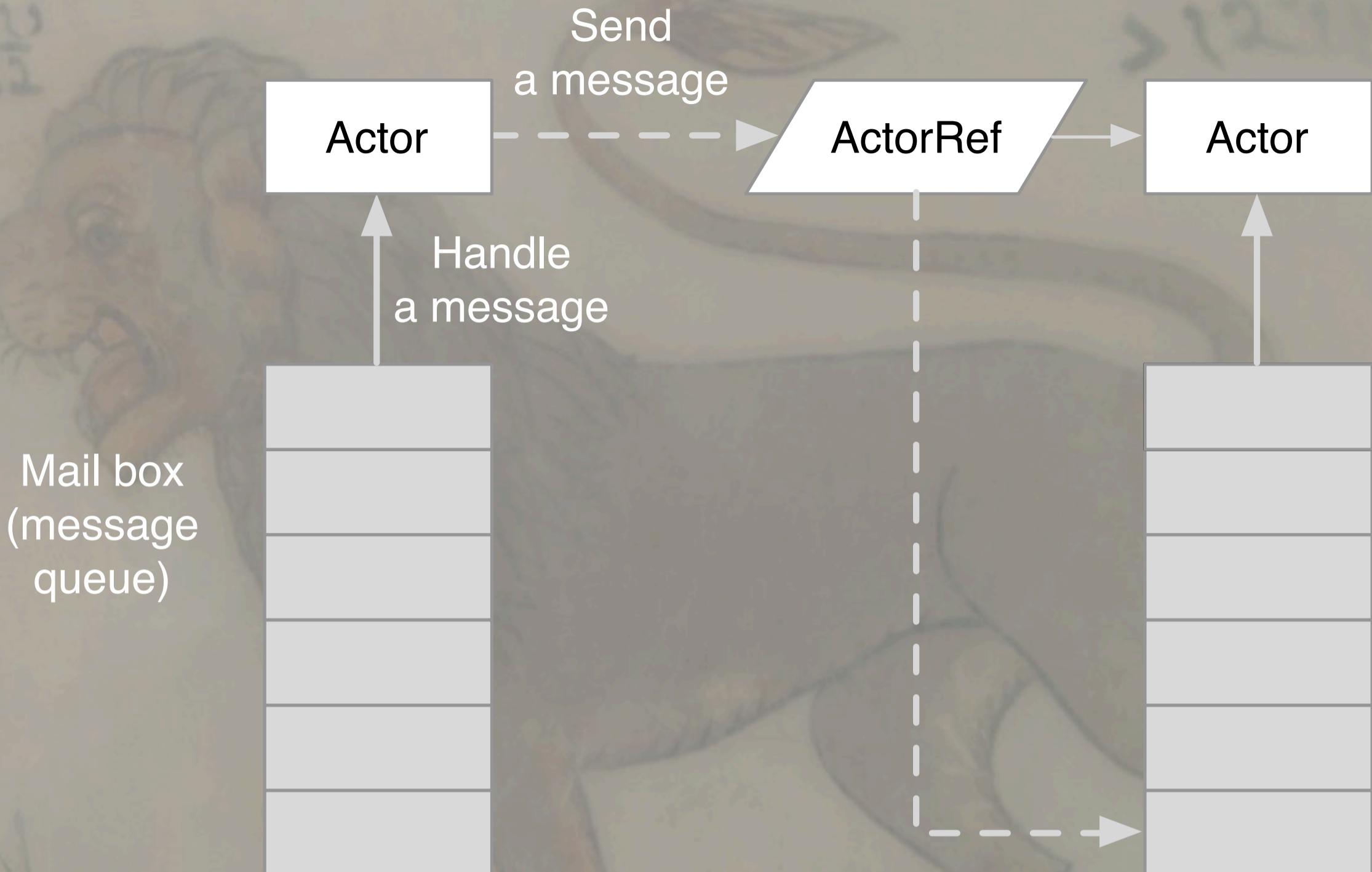
Actor Model



©Typesafe 2014-2015, All Rights Reserved

52

Wednesday, February 25, 15



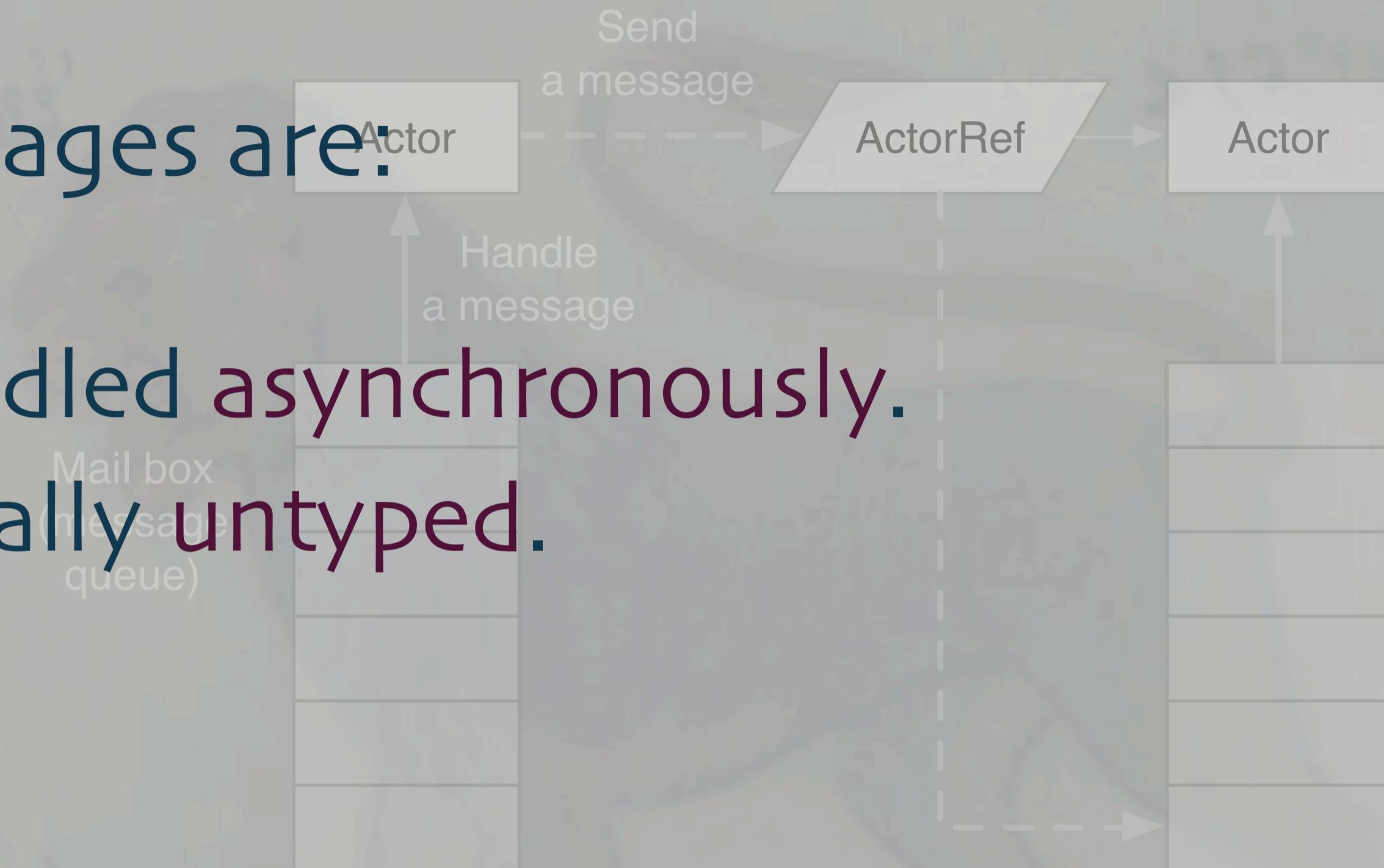
Superficially similar to channels.

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



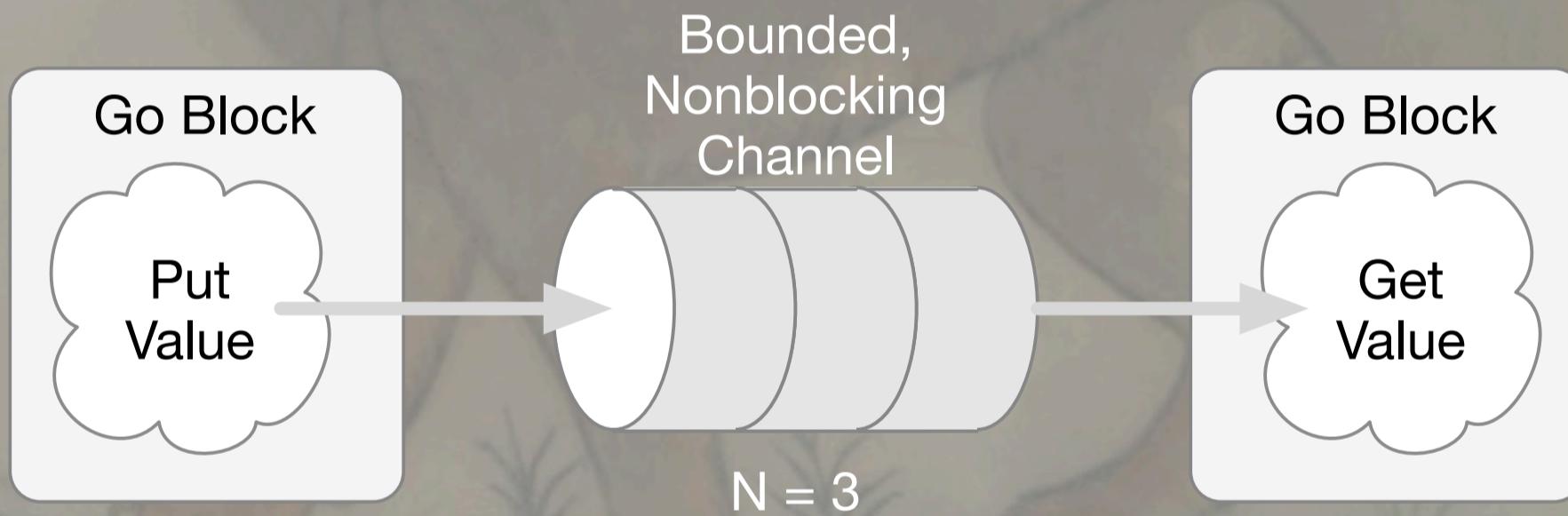
©Typesafe 2014-2015, All Rights Reserved

56

Wednesday, February 25, 15

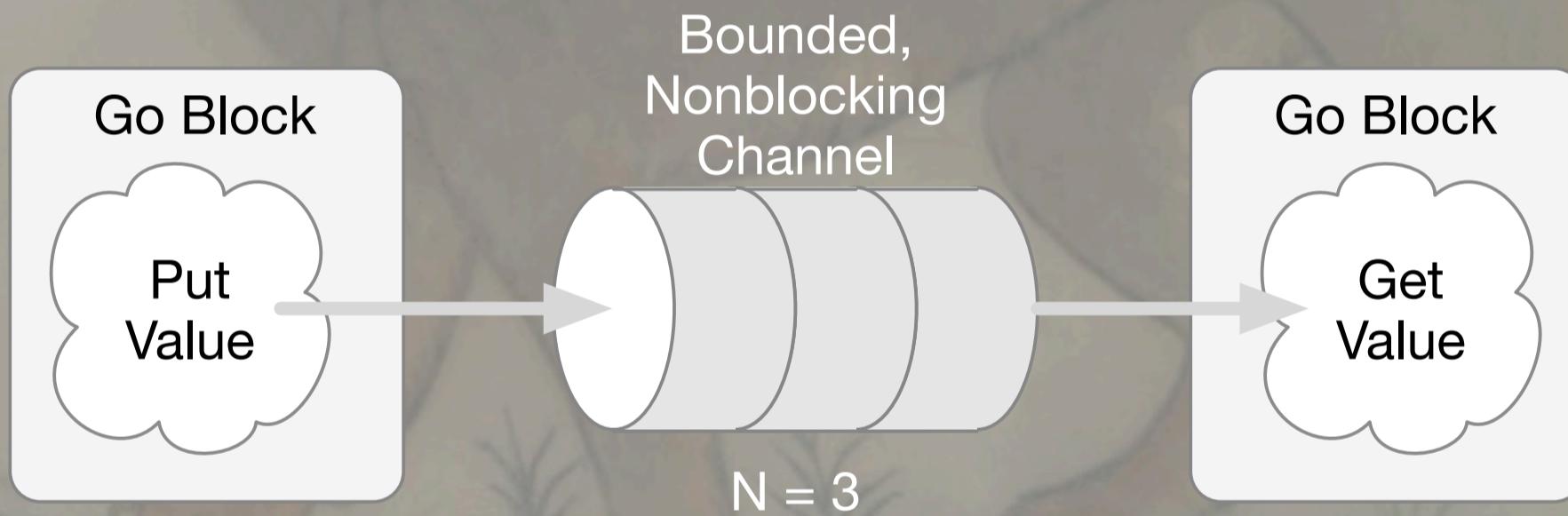
CSP Processes are anonymous

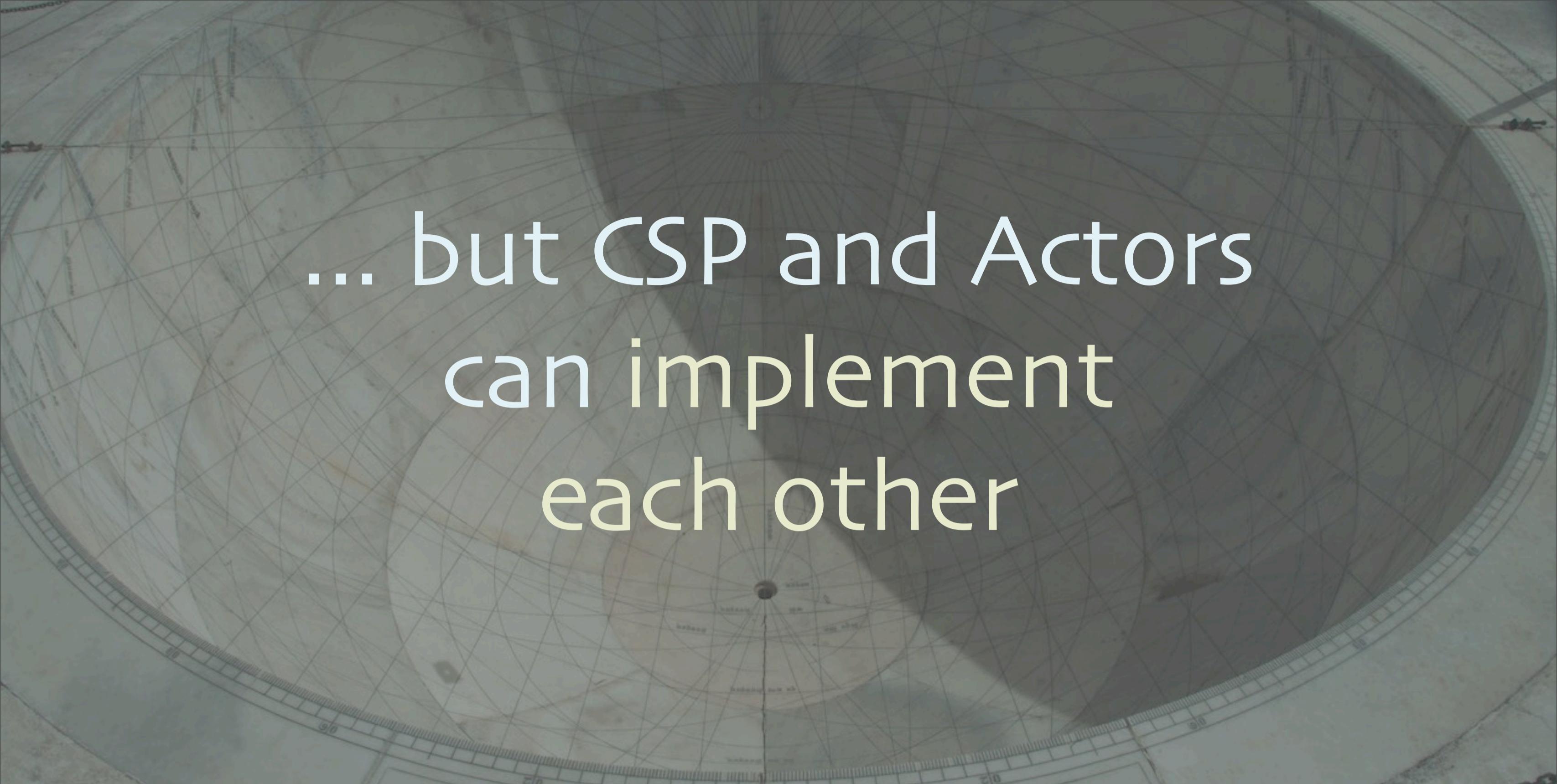
... while actors have identities.



CSP messaging is synchronous

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





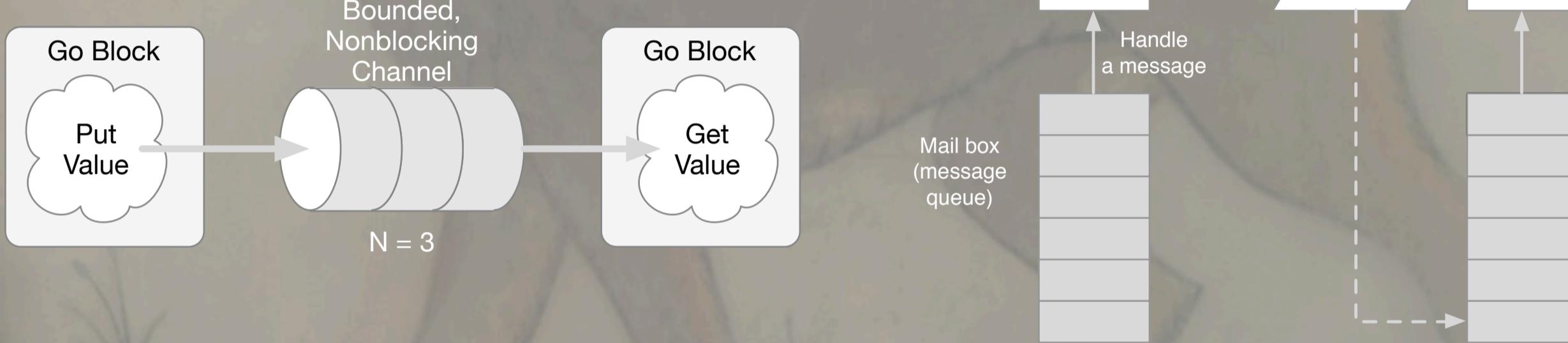
... but CSP and Actors
can implement
each other

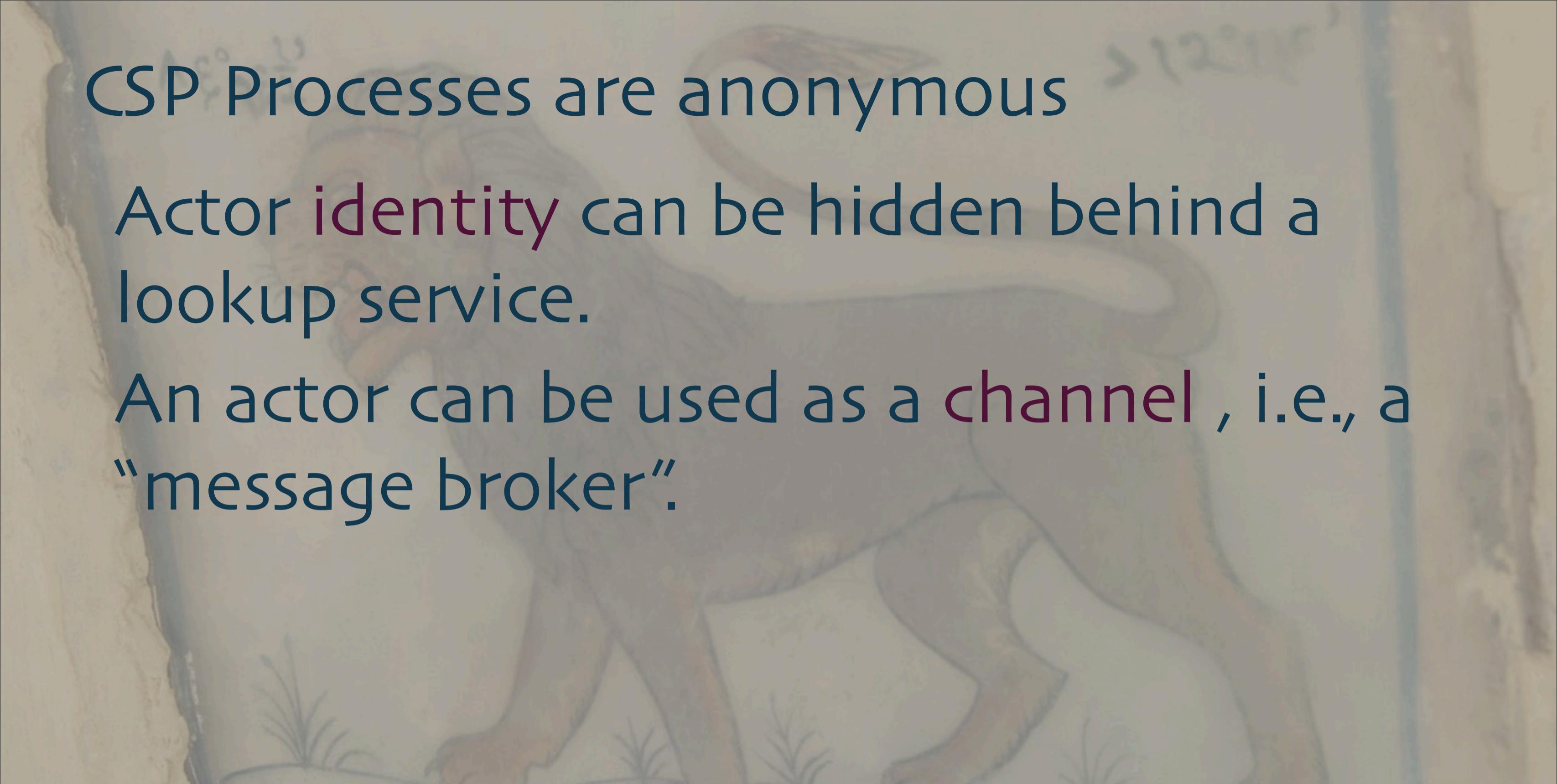


©Typesafe 2014-2015, All Rights Reserved

59

An actor mailbox looks a lot like a channel.





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.

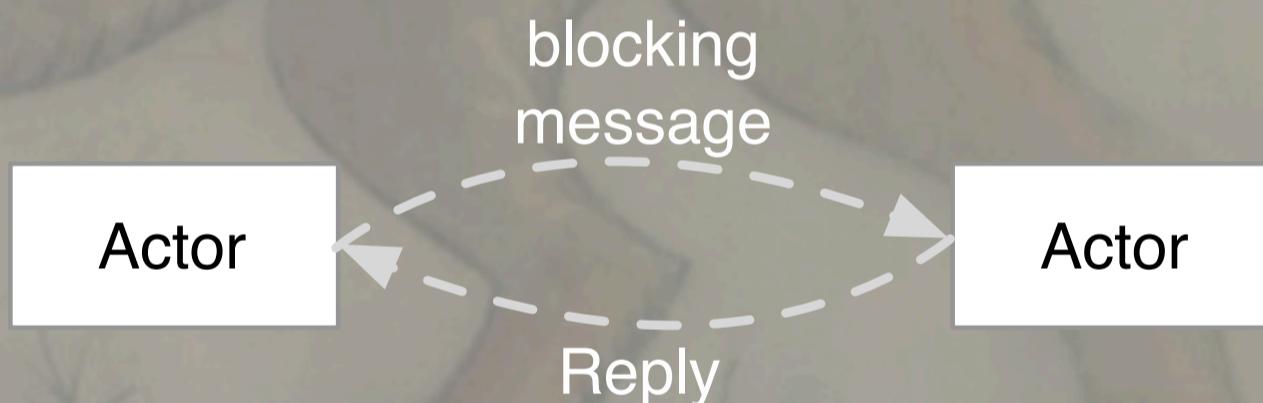


©Typesafe 2014-2015, All Rights Reserved

62

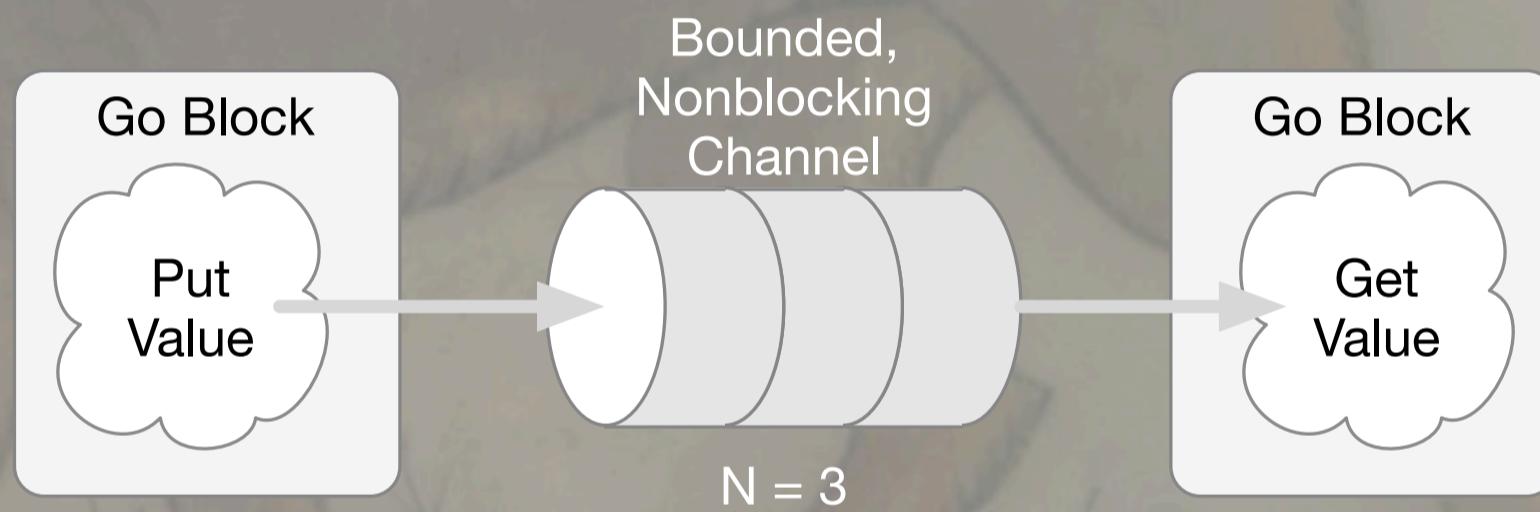
CSP messaging is synchronous

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



©Typesafe 2014-2015, All Rights Reserved

65

Wednesday, February 25, 15

Distributed Actors

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



©Typesafe 2014-2015, All Rights Reserved

66

Wednesday, February 25, 15

URL vs. URI?? See [http://danielmiessler.com/study/
url_vs_uri/](http://danielmiessler.com/study/url_vs_uri/)

Failure-handling in Actor Systems



©Typesafe 2014-2015, All Rights Reserved

67

Wednesday, February 25, 15

Let it Crash!



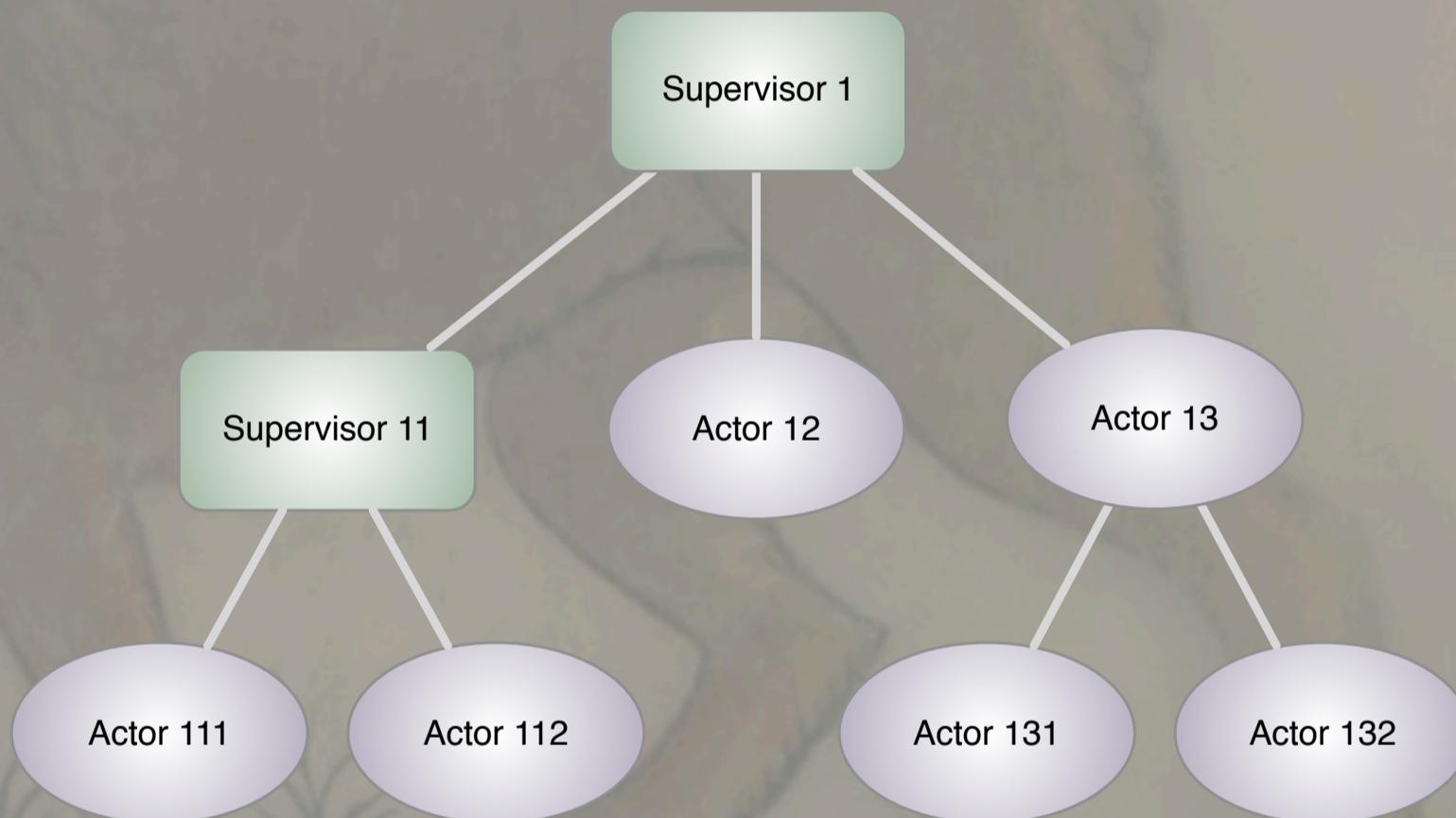
©Typesafe 2014-2015, All Rights Reserved

68

Wednesday, February 25, 15

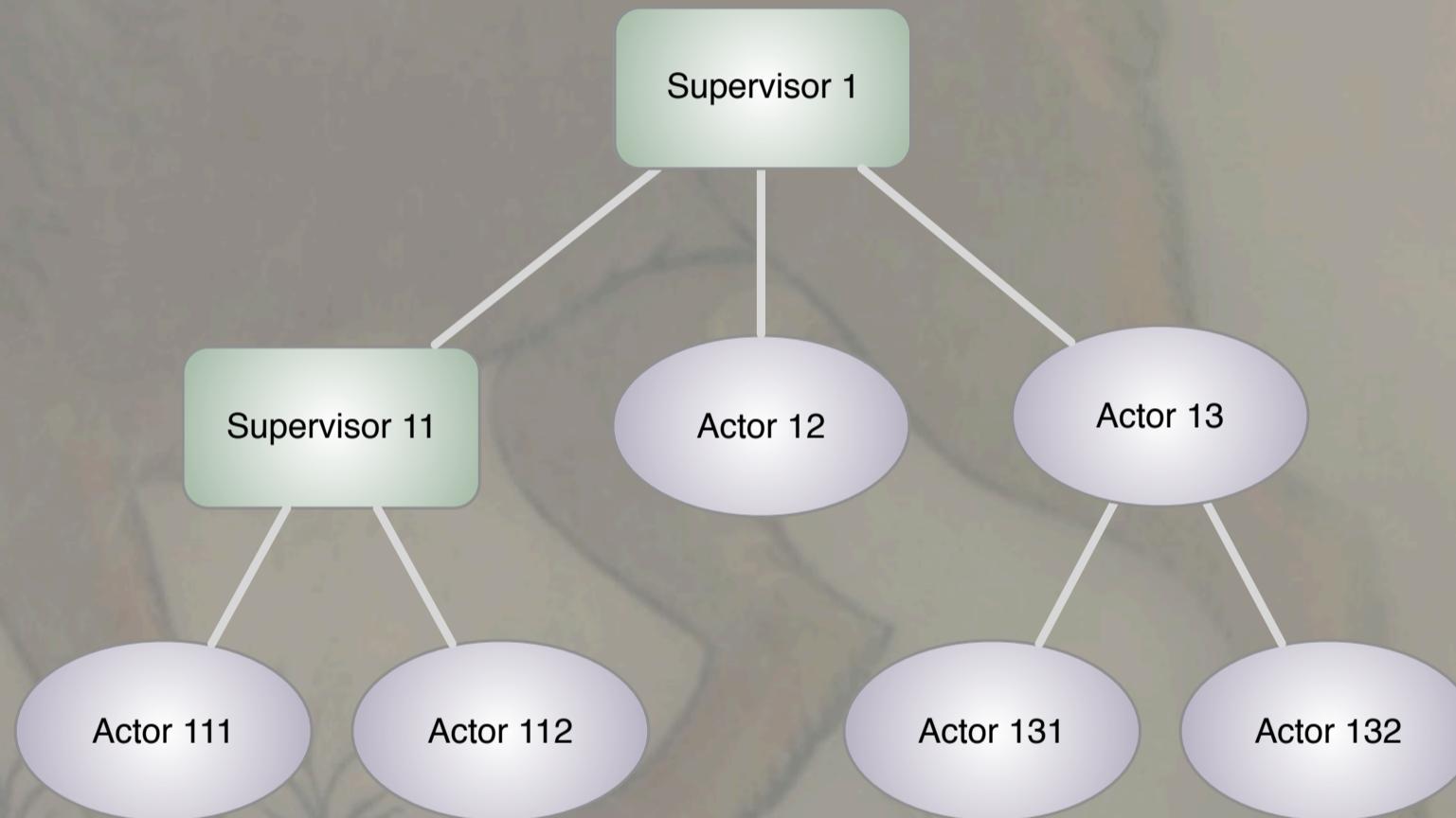
Erlang introduced supervisors

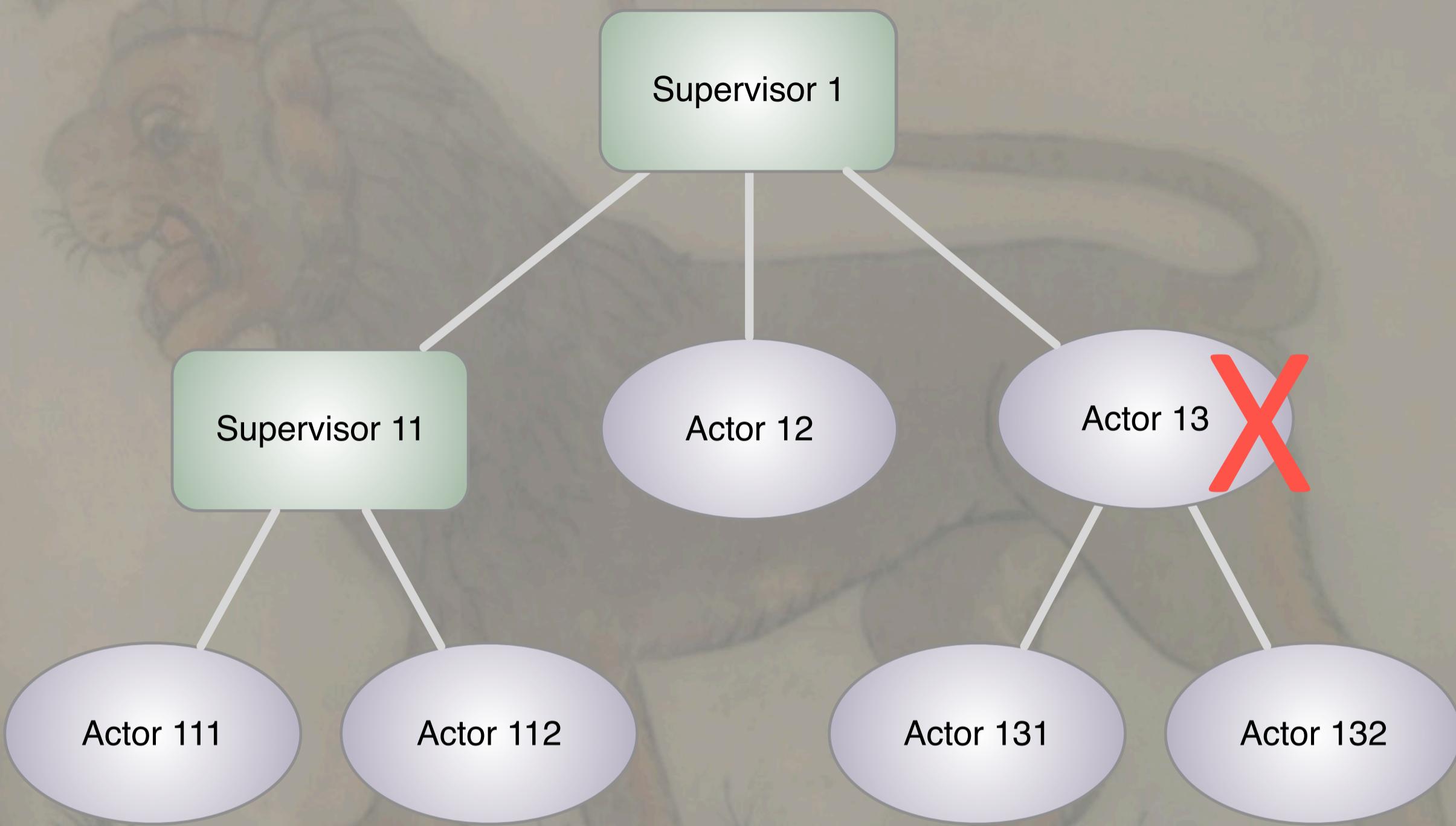
A hierarchy of actors that manage each “worker” actor’s lifecycle.

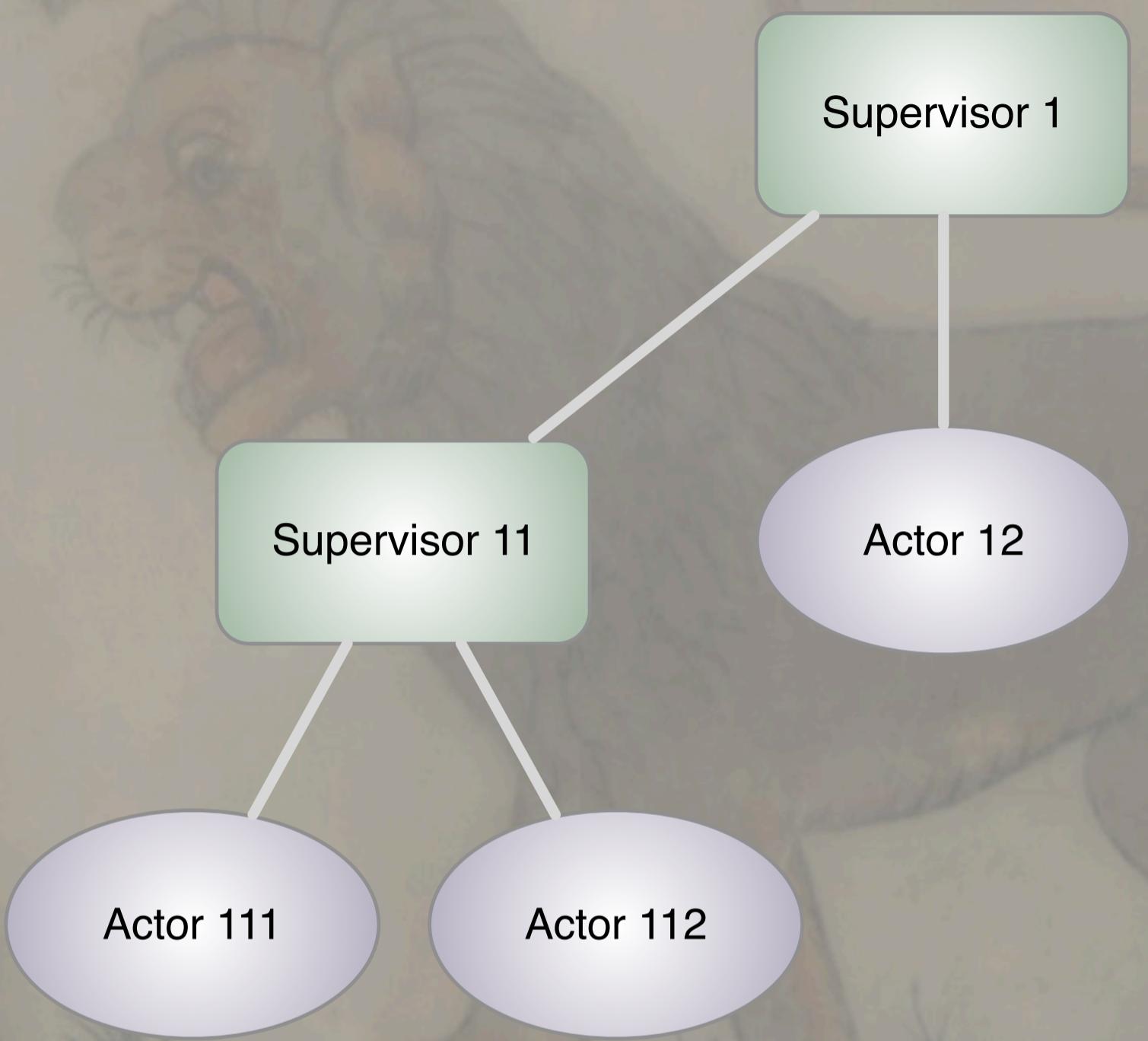


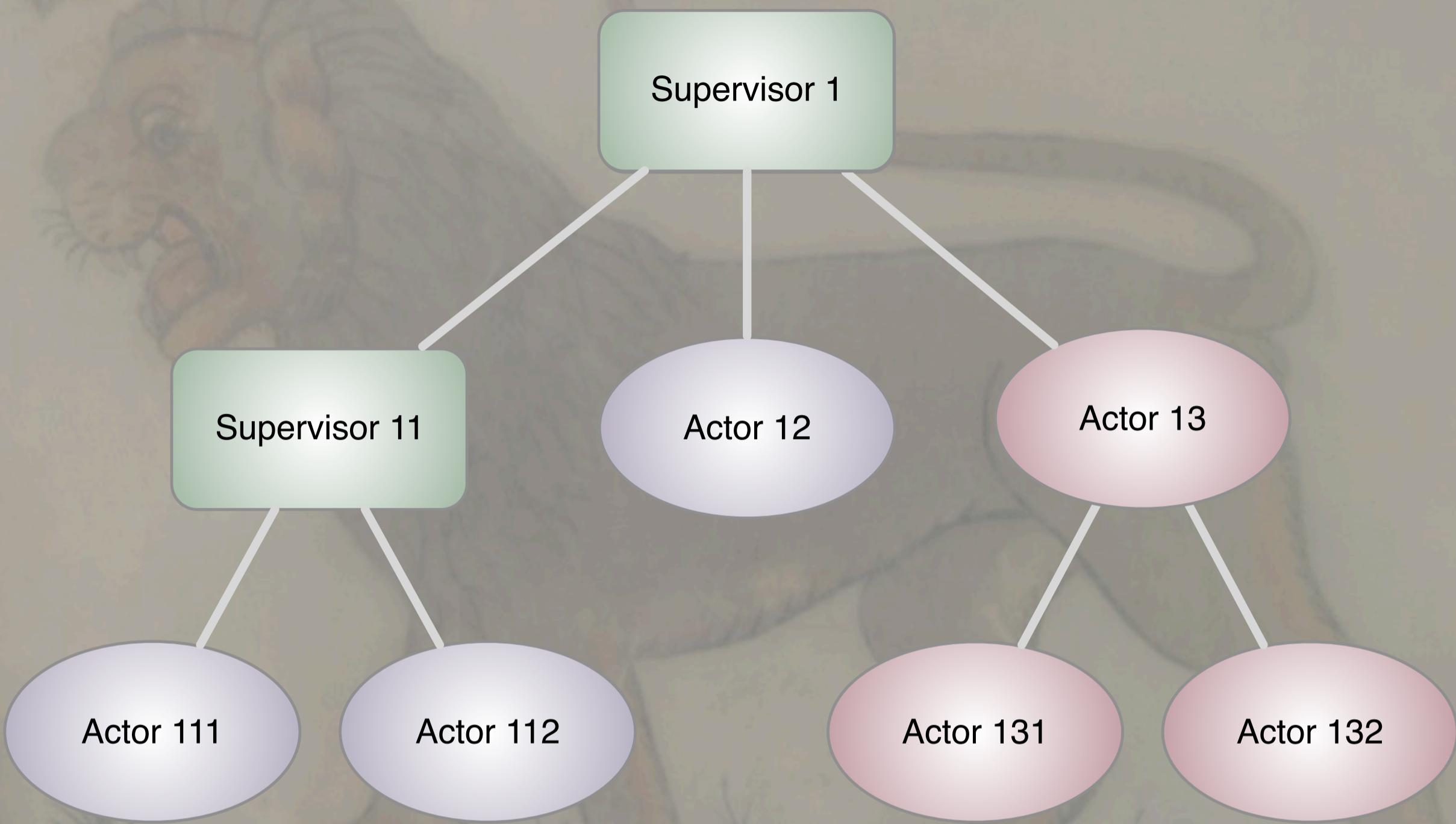
Erlang introduced supervisors

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



©Typesafe 2014-2015, All Rights Reserved

75

Wednesday, February 25, 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.



©Typesafe 2014-2015, All Rights Reserved

76

Wednesday, February 25, 15

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

Tim Baldridge's 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”.



©Typesafe 2014-2015, All Rights Reserved

77

Wednesday, February 25, 15

From https://github.com/halgari/clojure-conj-2013-core.async-examples/blob/master/src/clojure_conj_talk/core.clj

Most of these are based on his toy example, not a production-calibre implementation.

I'll add...

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

Answers



©Typesafe 2014-2015, All Rights Reserved

79

Wednesday, February 25, 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.



©Typesafe 2014-2015, All Rights Reserved

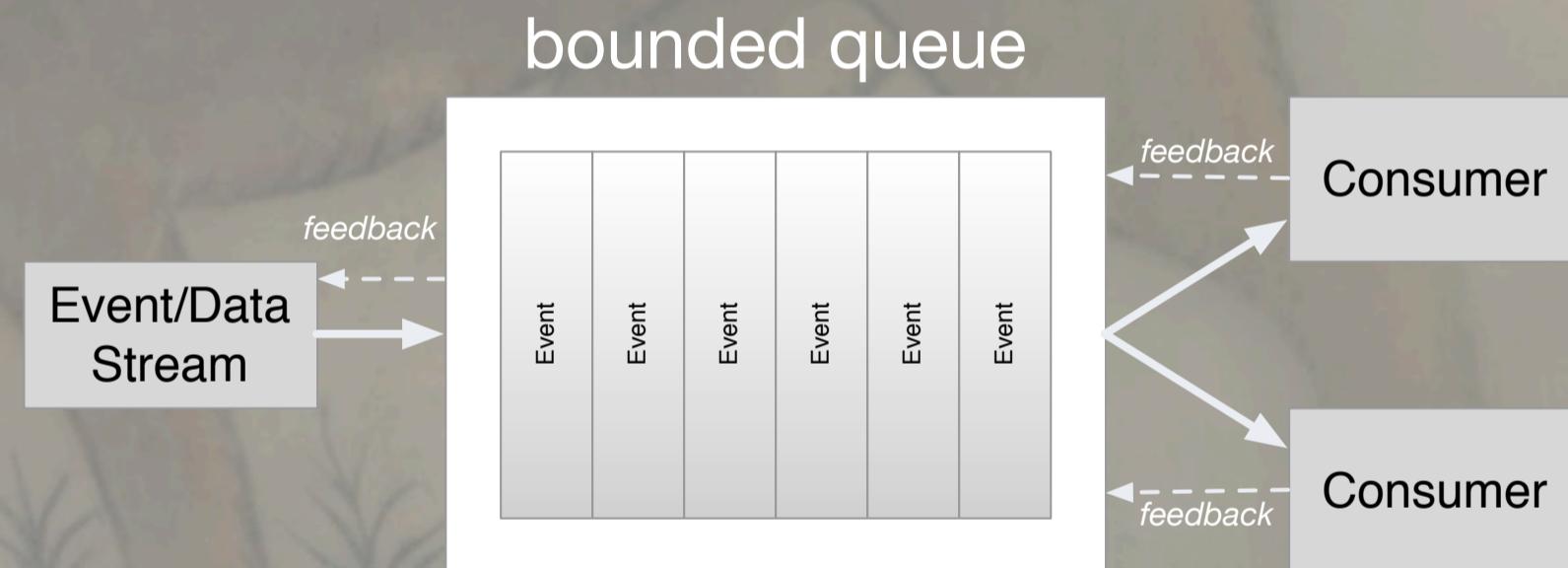
80

Wednesday, February 25, 15

In other words, ignore toy examples.

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 reads and writes.



©Typesafe 2014-2015, All Rights Reserved

83

Couples a queue, mutable state, and a process

- Production systems provide as much decoupling as you need.



©Typesafe 2014-2015, All Rights Reserved

84

Actors are untyped

- While there have been typed actor implementations, actors are analogous to OS processes:
 - Clear abstraction boundaries.
 - But careful handling of inputs required.

Actors are untyped

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



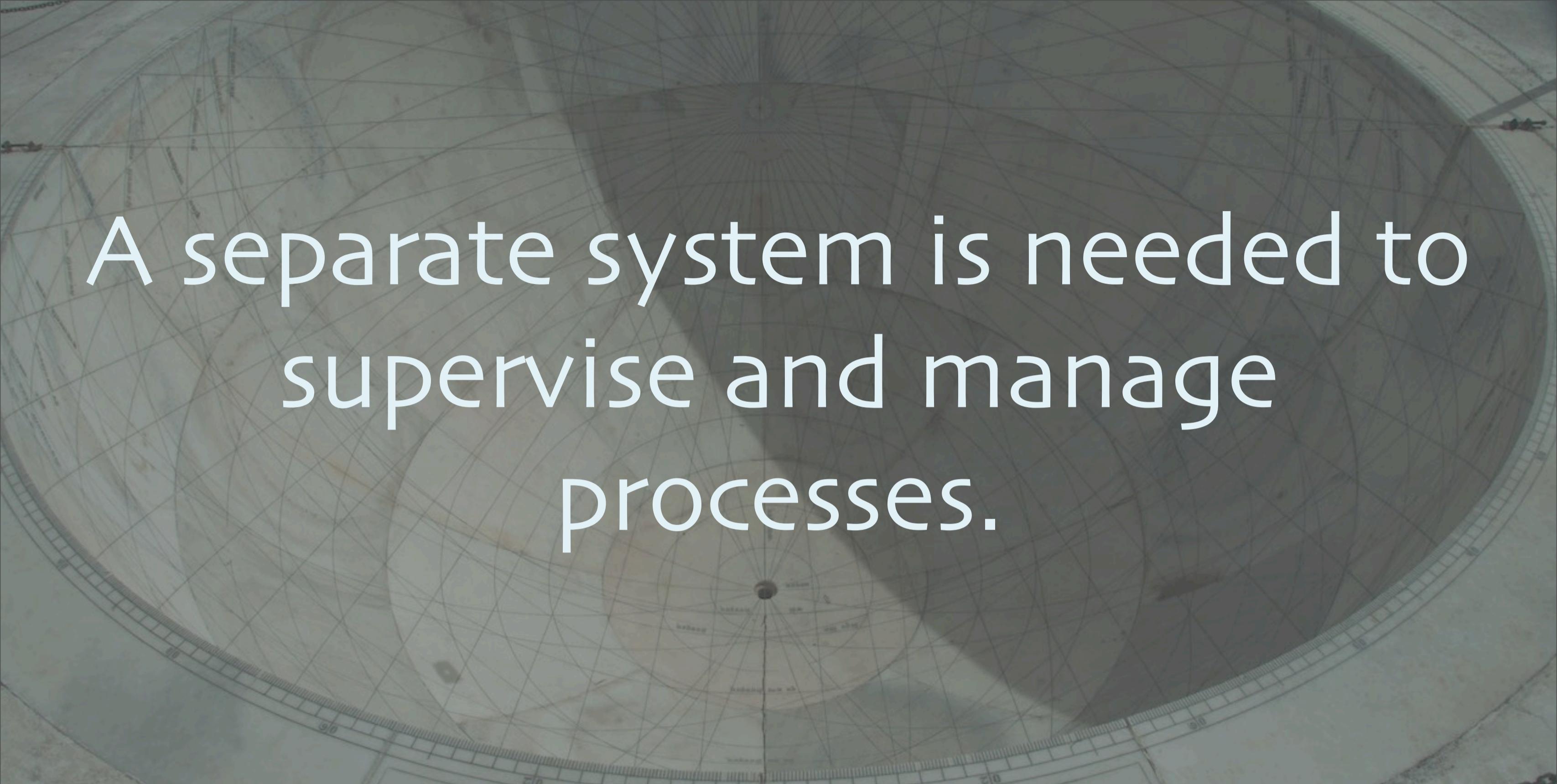
How should we handle failures?



©Typesafe 2014-2015, All Rights Reserved

87

Wednesday, February 25, 15

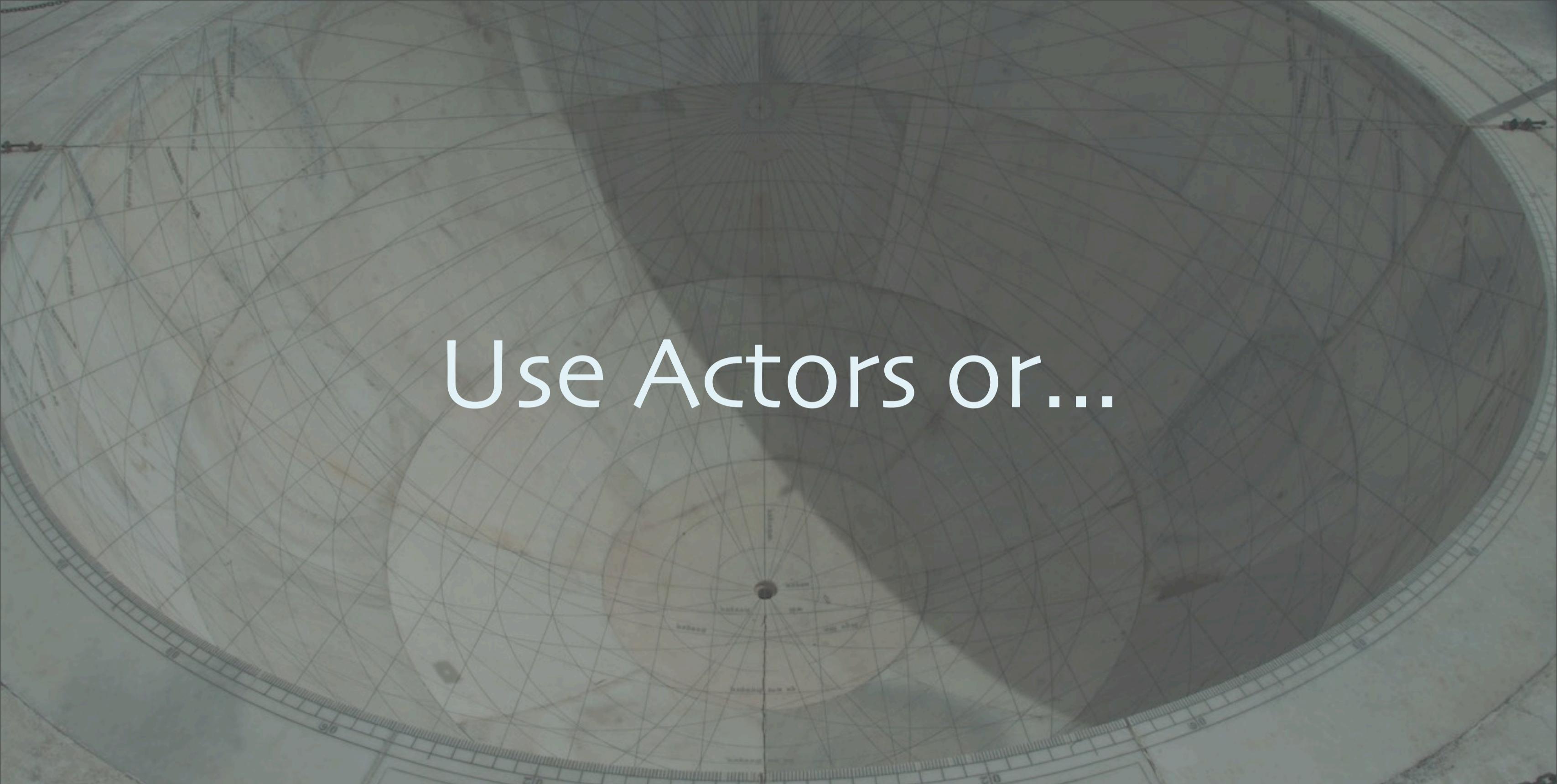


A separate system is needed to supervise and manage processes.



©Typesafe 2014-2015, All Rights Reserved

88



Use Actors or...



©Typesafe 2014-2015, All Rights Reserved

89

Wednesday, February 25, 15



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



©Typesafe 2014-2015, All Rights Reserved

90

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

Conclusions



Actors

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

Actors

- + Industry proven scalability and resiliency.
- + Native asynchrony.

Best-in-class strategy for failure handling.

CSP, Rx, etc.

- Limited failure handling facilities.
- Distributed channels?



©Typesafe 2014-2015, All Rights Reserved

95

Wednesday, February 25, 15

I would include futures in the list of derivatives.

CSP, Rx, etc.

+ Emphasize flow of control.

+ Typed channels.

Optimal replacement for multithreaded
(intra-process) programming.



Typesafe

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

©Typesafe 2014-2015, All Rights Reserved

Wednesday, February 25, 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) 2012-2014, Dean Wampler. All Rights Reserved.

Bonus Slides



©Typesafe 2014-2015, All Rights Reserved

98

Wednesday, February 25, 15



Communicating Sequential Processes

Message passing
via channels



©Typesafe 2014-2015, All Rights Reserved

99

Wednesday, February 25, 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



©Typesafe 2014-2015, All Rights Reserved

100

Wednesday, February 25, 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.



©Typesafe 2014-2015, All Rights Reserved

101

Wednesday, February 25, 15

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



©Typesafe 2014-2015, All Rights Reserved

102

Wednesday, February 25, 15

Prefix

$$a \rightarrow P$$

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



©Typesafe 2014-2015, All Rights Reserved

103

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.



©Typesafe 2014-2015, All Rights Reserved

106

Interface Parallel

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

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 .

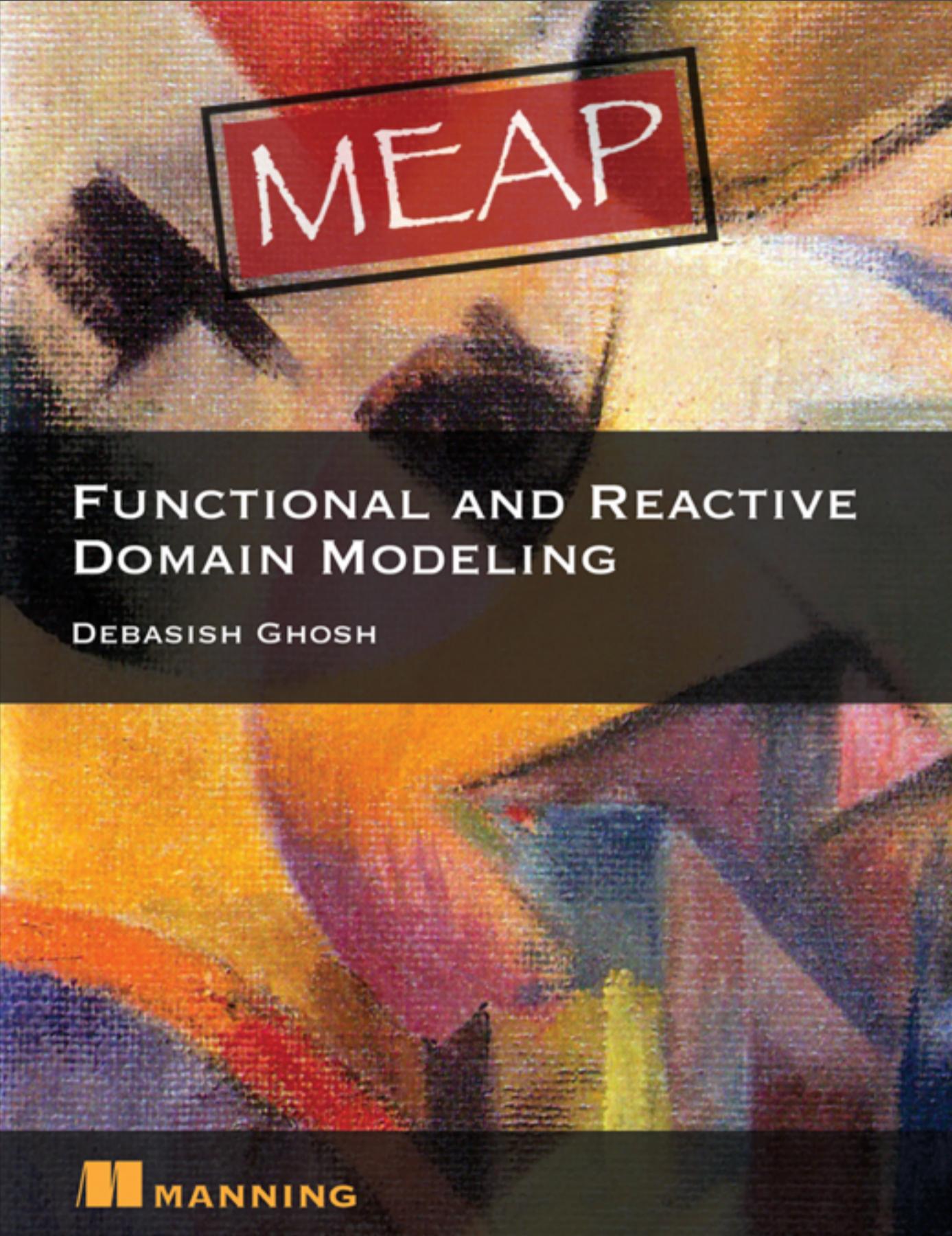
References



©Typesafe 2014-2015, All Rights Reserved

109

Wednesday, February 25, 15



©Typesafe 2014-2015, All Rights Reserved

110

Wednesday, February 25, 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



©Typesafe 2014-2015, All Rights Reserved

111

Wednesday, February 25, 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.



©Typesafe 2014-2015, All Rights Reserved

112

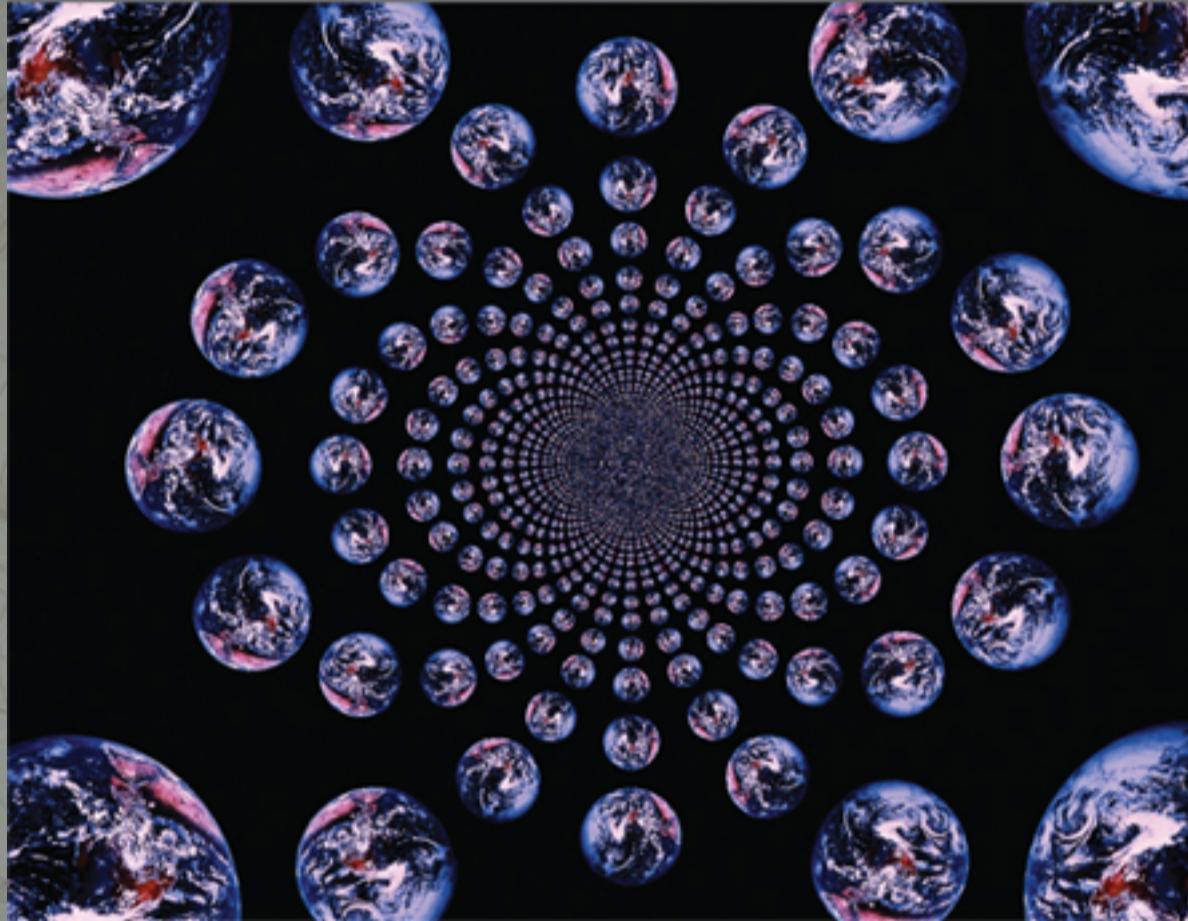
Wednesday, February 25, 15

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

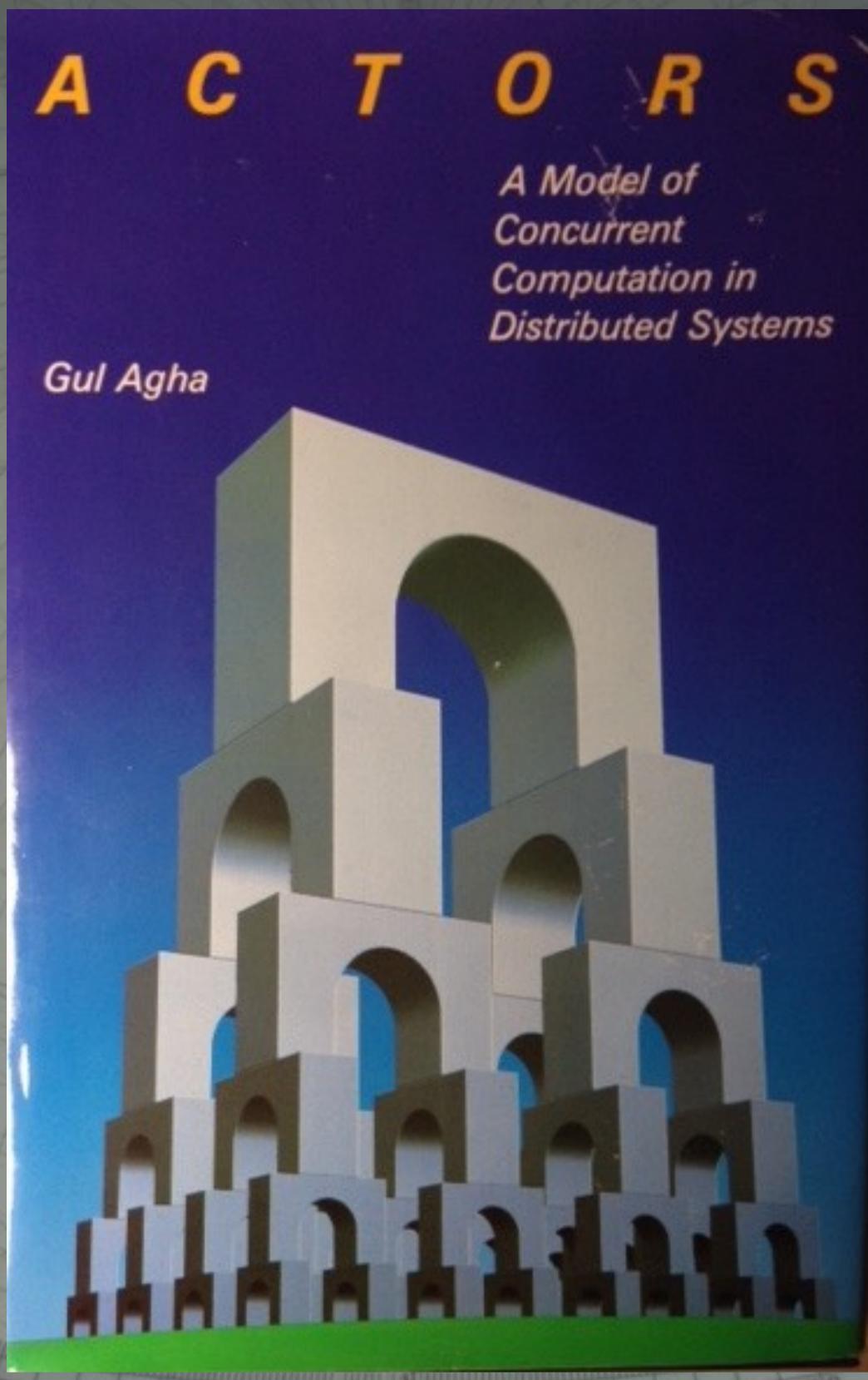


©Typesafe 2014-2015, All Rights Reserved

113

Wednesday, February 25, 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, February 25, 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

Urheberrechtlich geschütztes Material



©Typesafe 2014-2015, All Rights Reserved

115

Wednesday, February 25, 15

Survey of the classic graph traversal algorithms, algorithms for detecting failures in a cluster, leader election, etc.

DISTRIBUTED ALGORITHMS

AN INTUITIVE APPROACH



WAN FOKKINK

Copyrighted material

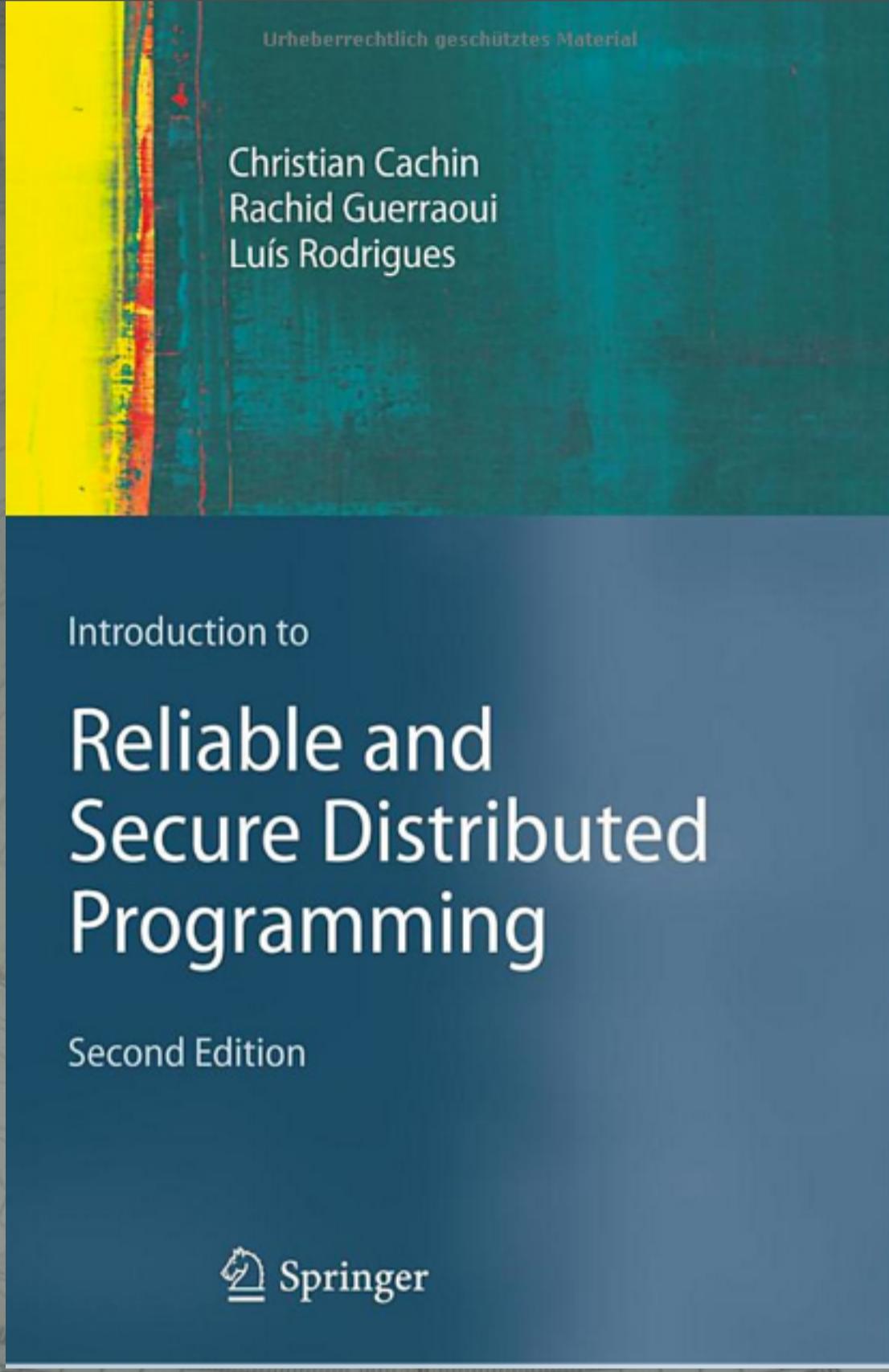


©Typesafe 2014-2015, All Rights Reserved

116

Wednesday, February 25, 15

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



©Typesafe 2014-2015, All Rights Reserved

117

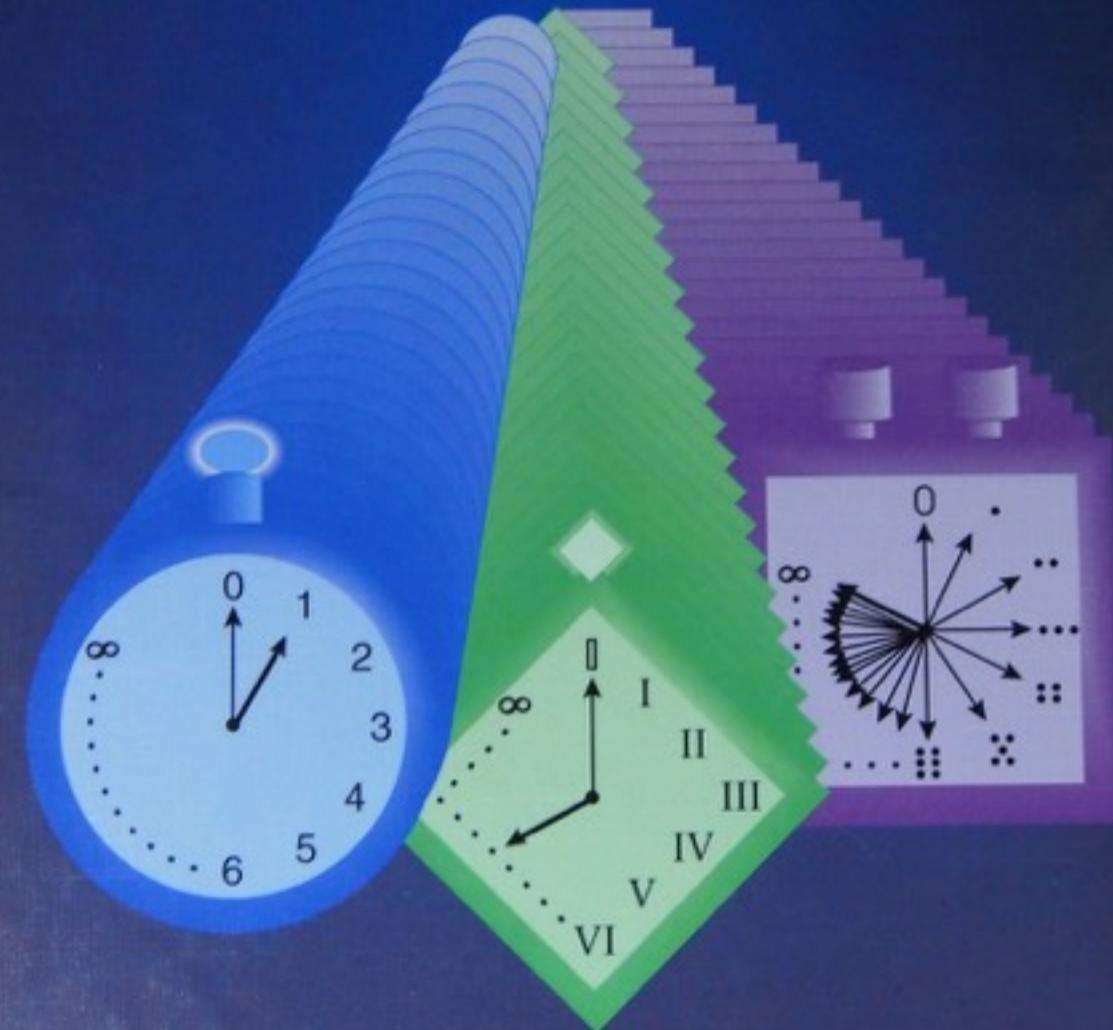
Wednesday, February 25, 15

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•



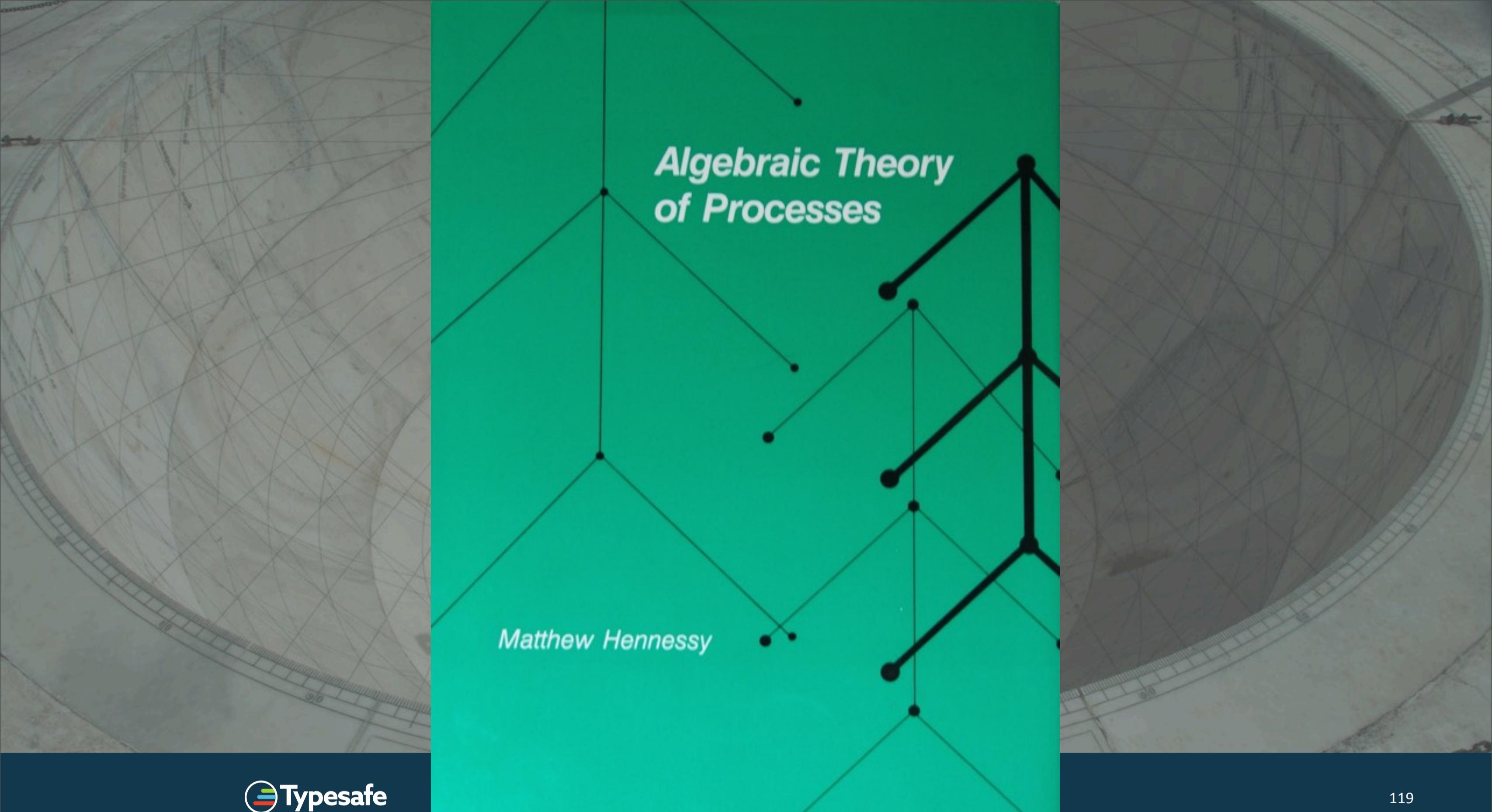
Springer-Verlag



118

Wednesday, February 25, 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.



DISTRIBUTED COMPUTING *through* COMBINATORIAL TOPOLOGY



Maurice Herlihy
Dmitry Kozlov
Sergio Rajsbaum

Copyrighted Material

© Typesafe 2014-2015, All Rights Reserved

120



Wednesday, February 25, 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



©Typesafe 2014-2015, All Rights Reserved

121

Wednesday, February 25, 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/>