O'REILLY Software Architecture Conference — MARCH 16-19, 2015 BOSTON, MA

# Error Handling in Reactive Systems

Dean Wampler, Ph.D.
Typesafe

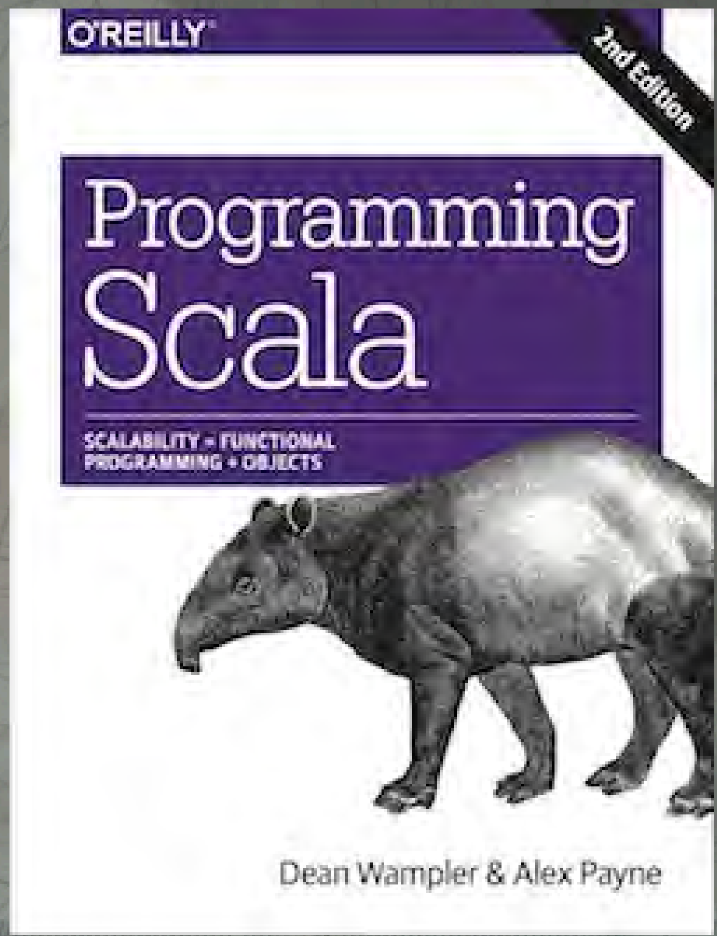
Typesafe

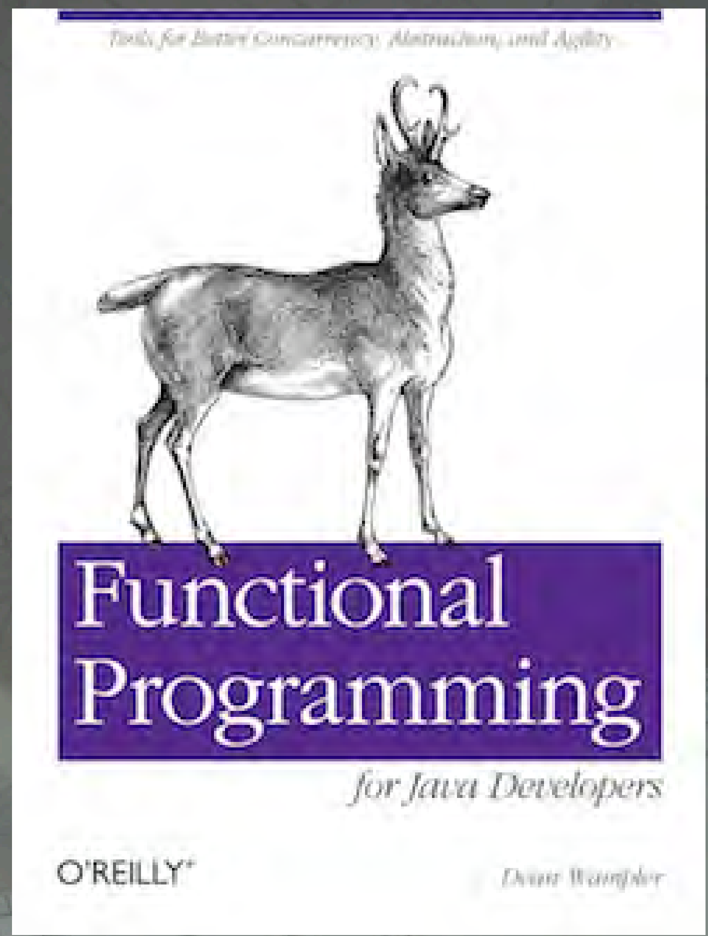


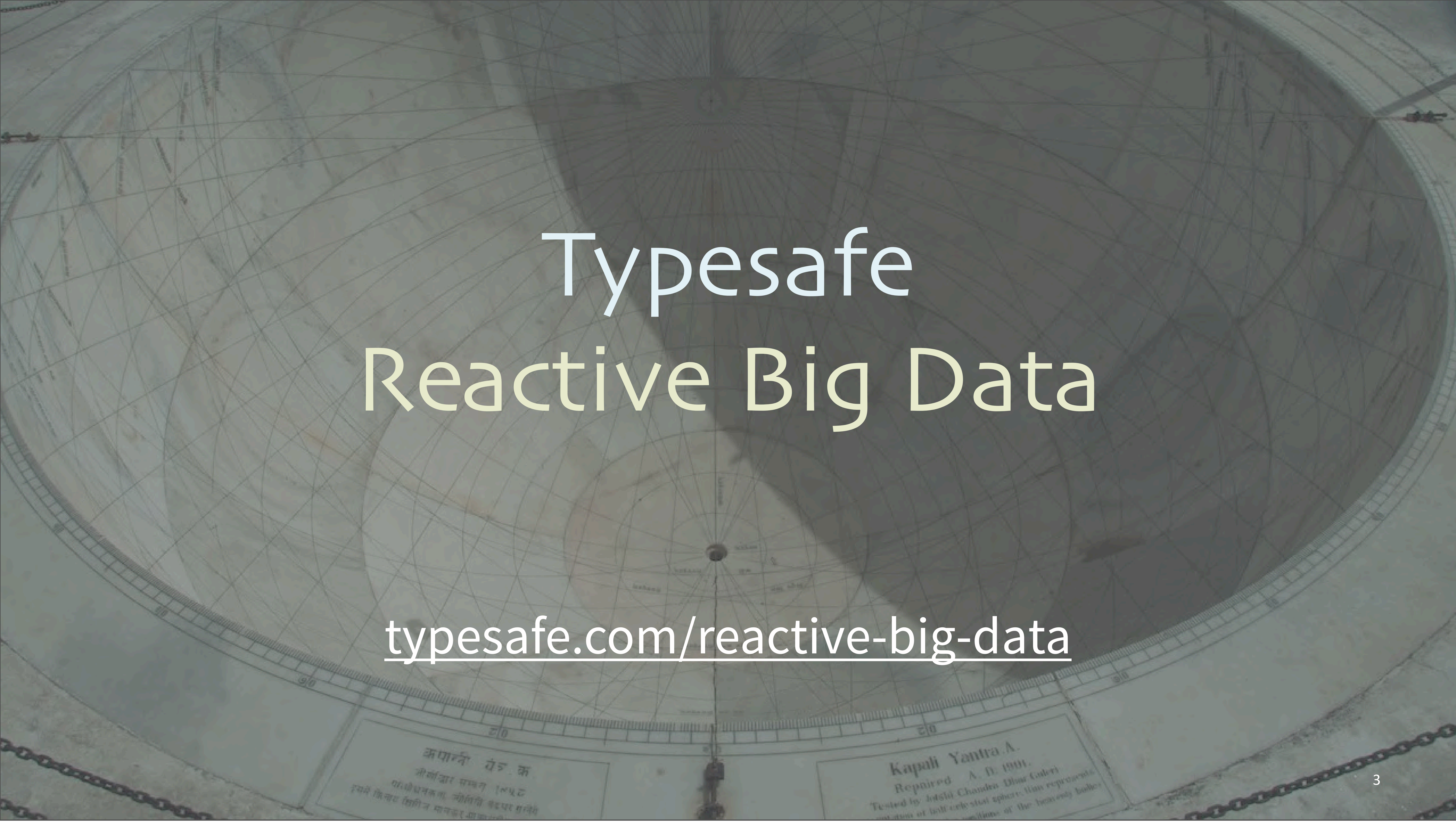
Wednesday, March 18, 15

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.

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

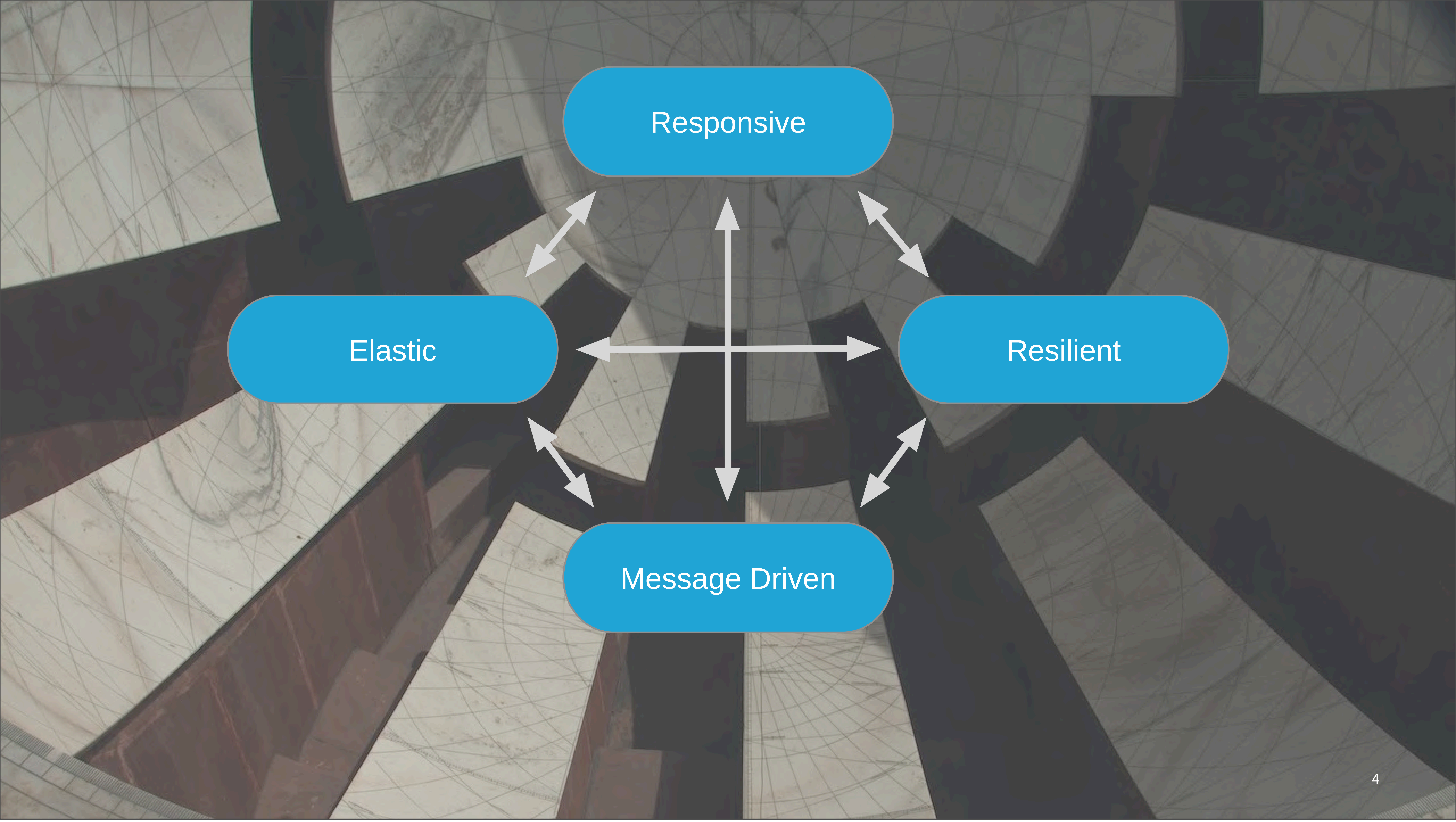






Wednesday, March 18, 15

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

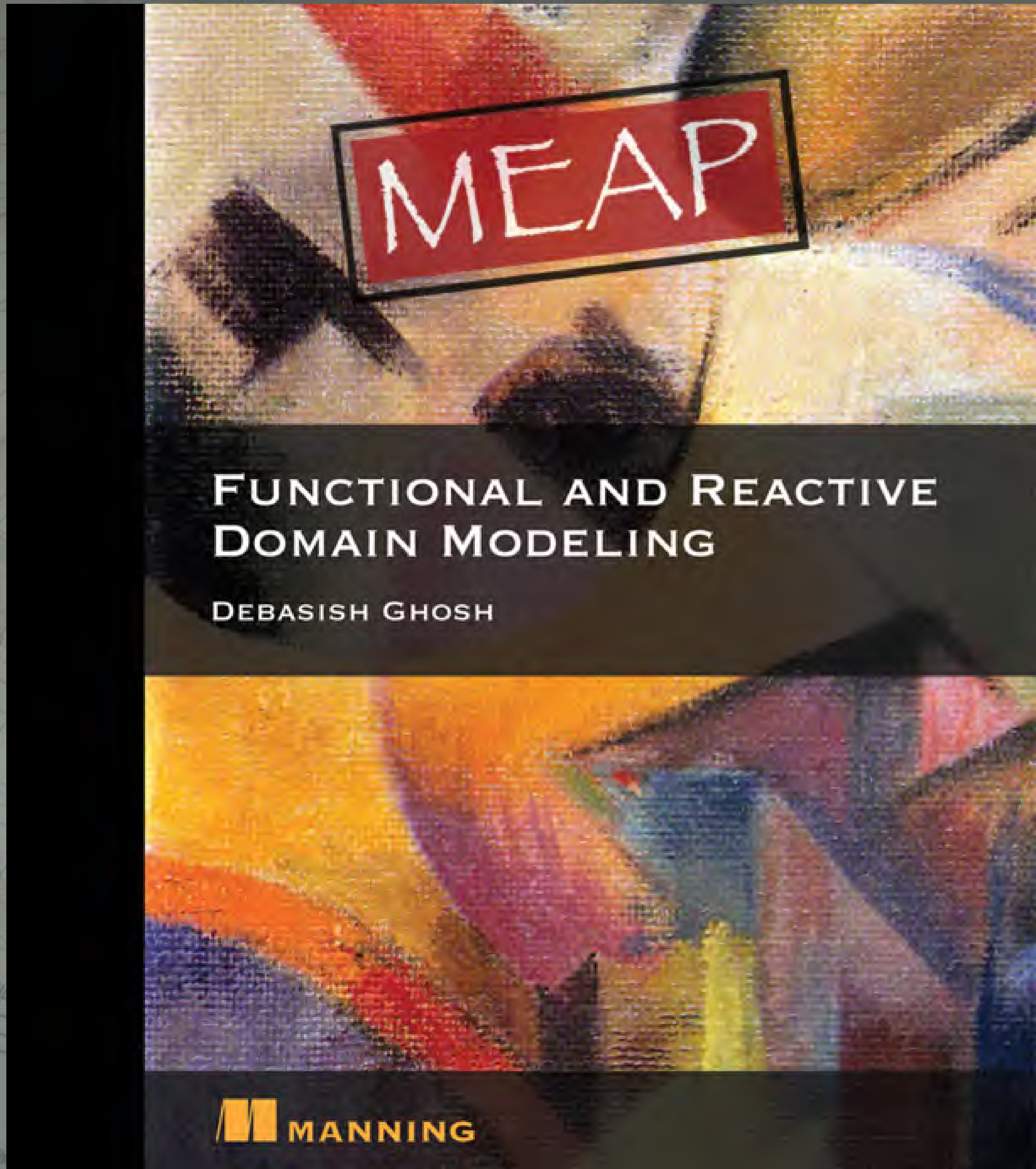





Wednesday, March 18, 15


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.







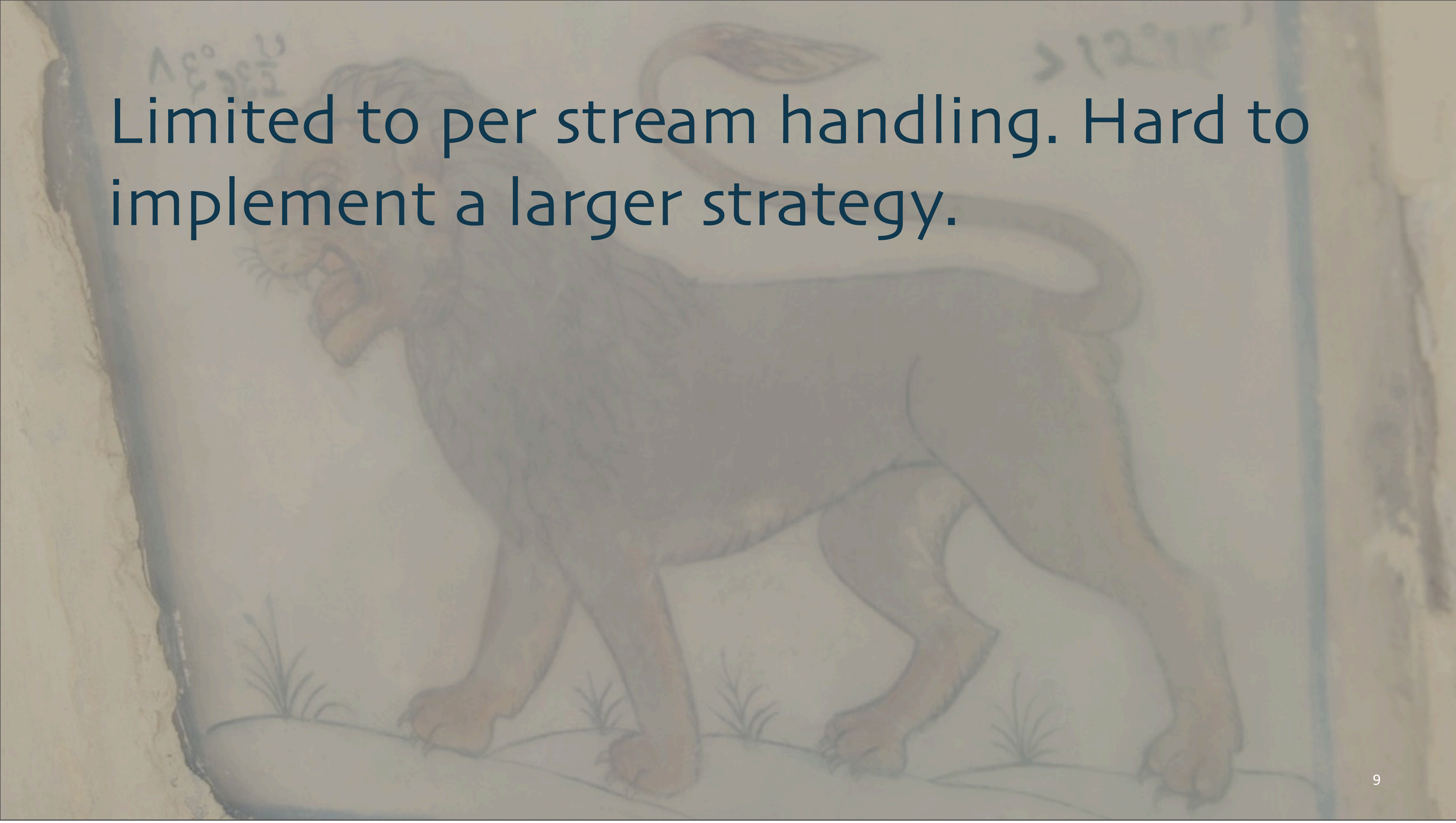

I’ve structured parts of this talk around points made in Debasish’s new book, which has lots of interesting practical ideas for combining functional programming and reactive approaches with classic Domain-Driven Design by Eric Evans.


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.




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.

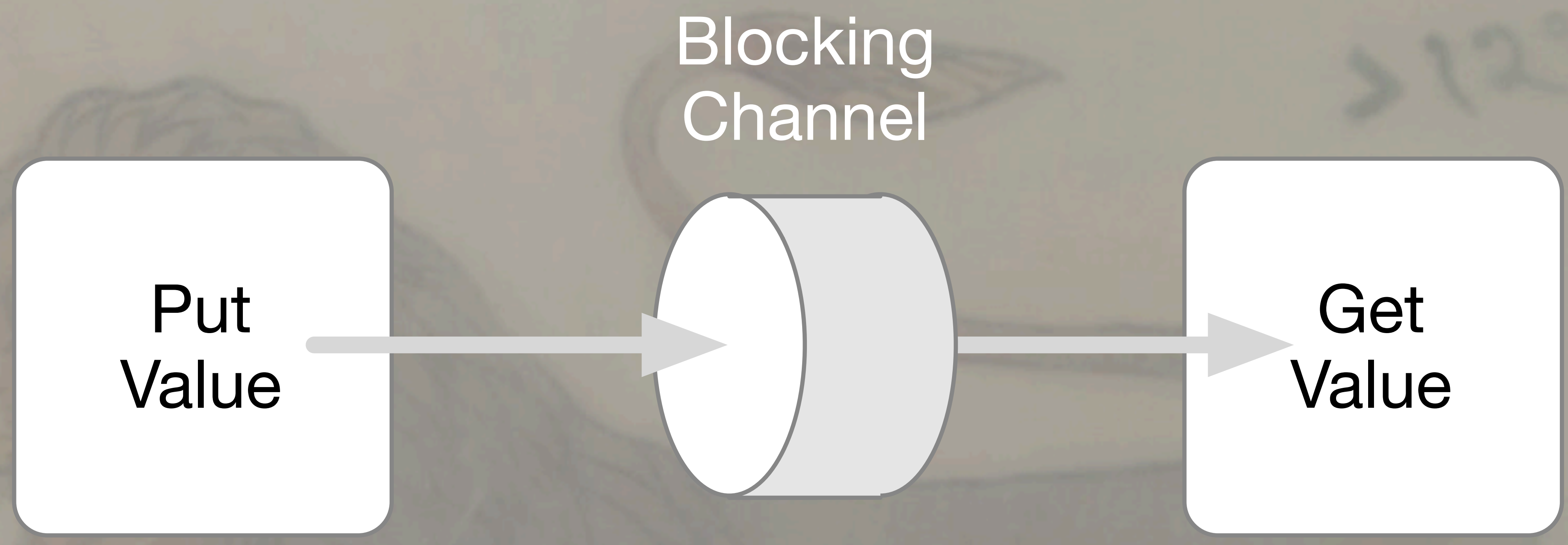



Wednesday, March 18, 15

http://www.youtube.com/watch?v=f6kdp27TYZs&feature=youtu.be

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

# CSP: inspired Go & Clojure’s core.async




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

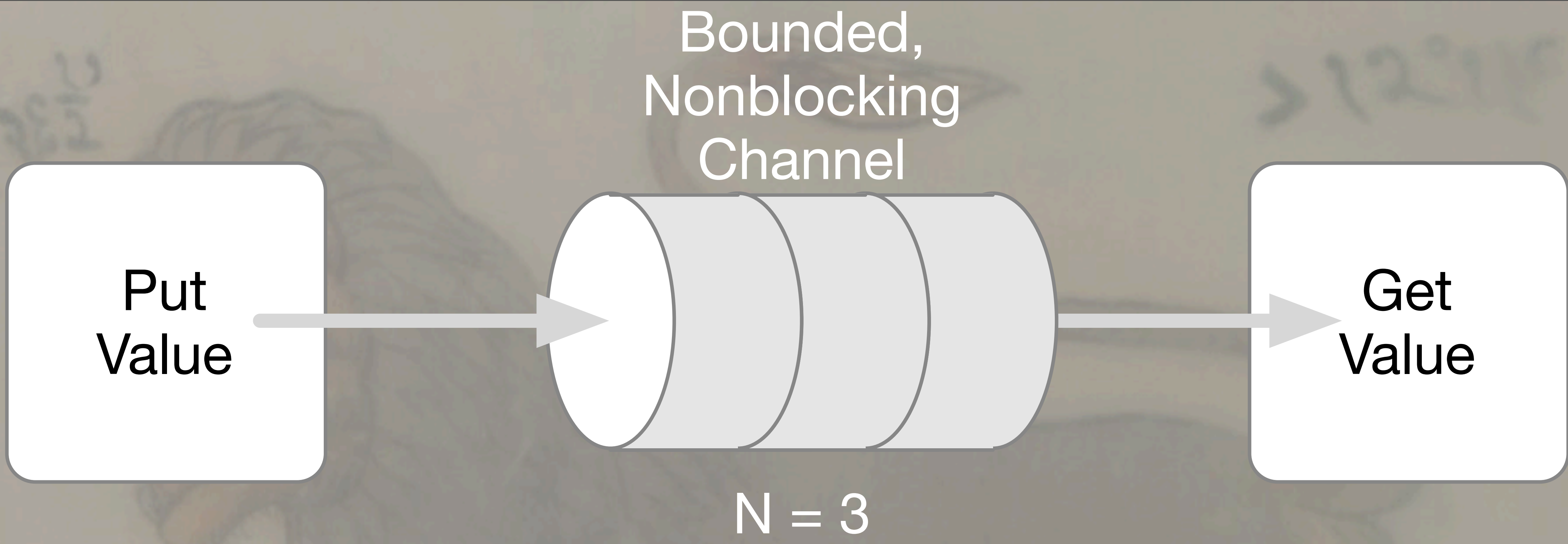




Simplest channel, a blocking, 1-element “connector” used to share values, one at a time between a source and a waiting sync. The put operation blocks if there is no sync waiting on the other end.

The channel can be typed (Go lang).

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

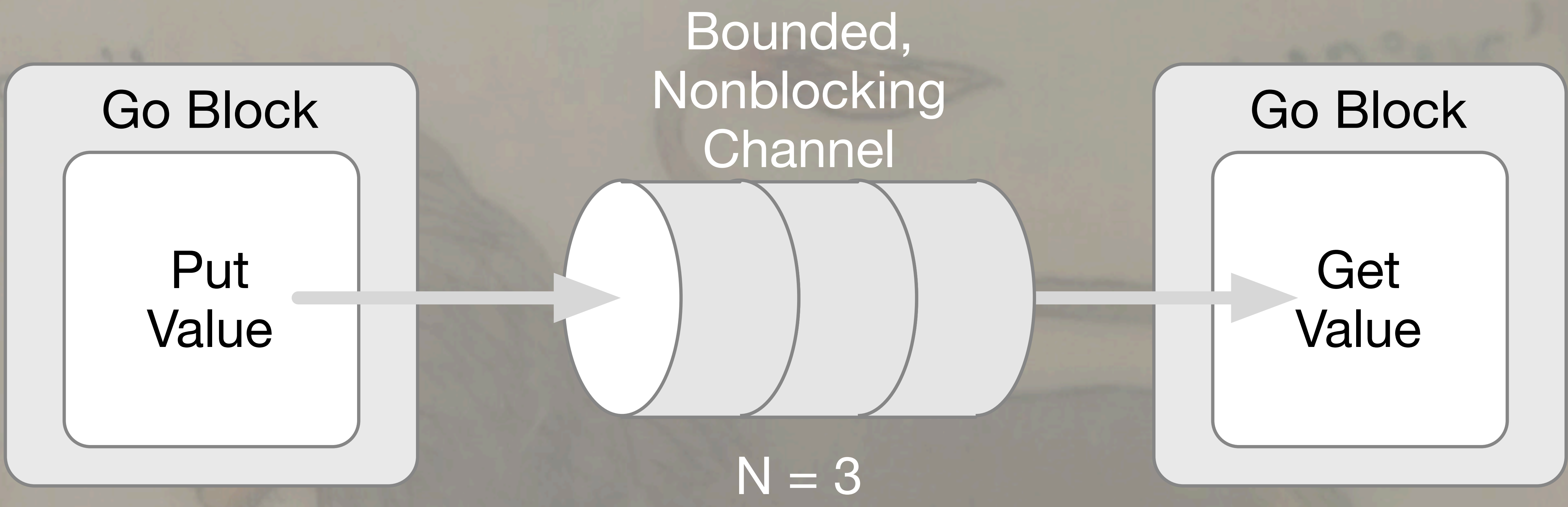



- When full:
  - Block on put.
  - Drop newest (i.e., put value).
  - Drop oldest ("sliding" window).





A non-blocking queue, but bounded in size. Normally, N wouldn't be this small. You DON'T want it to be infinite either, because eventually you'll fill it and run out of memory! So, what should you do when it's full? We'll come back to this question when we discuss Reactive Streams later.




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

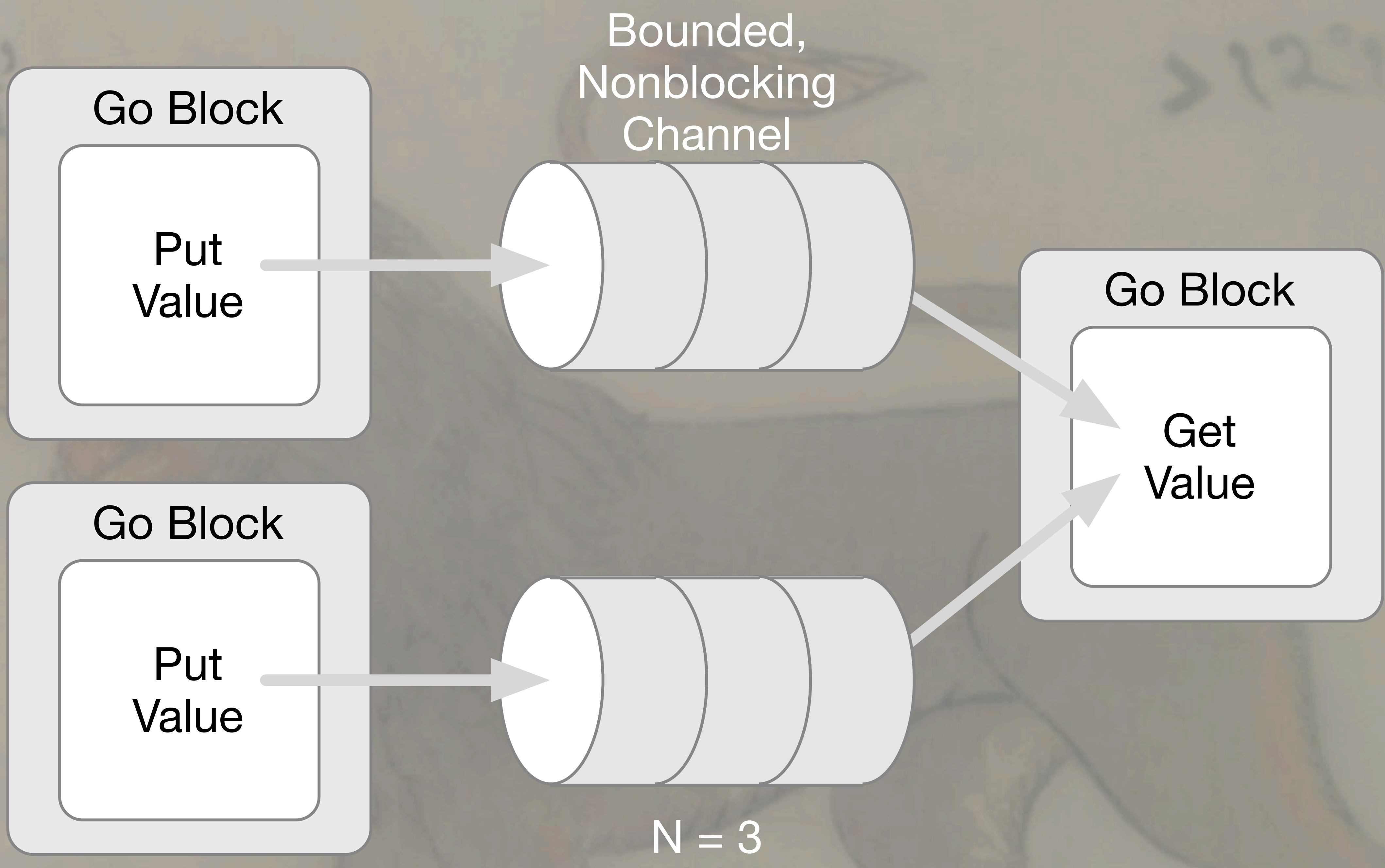


Wednesday, March 18, 15

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.



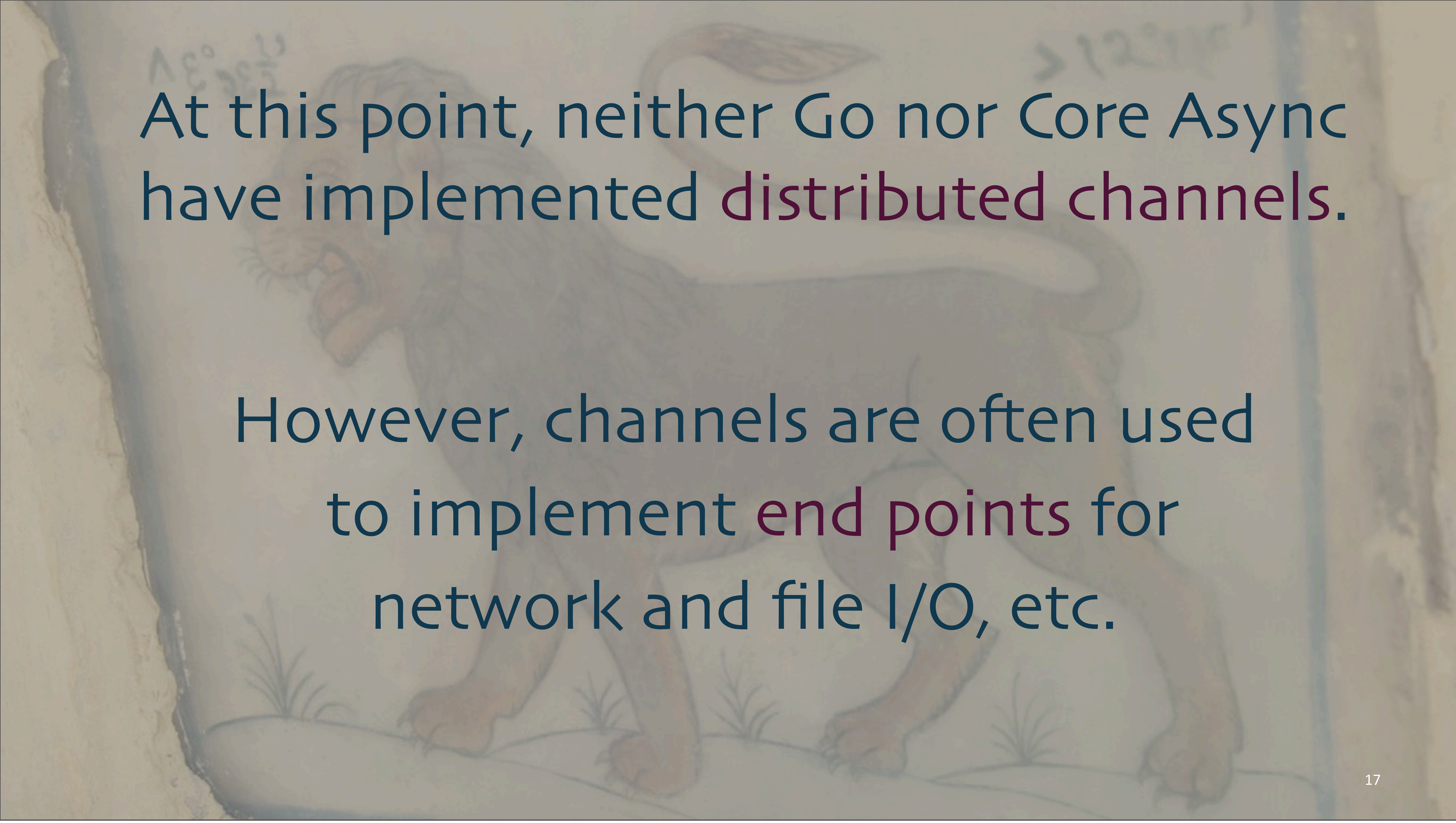

- Blocking or nonblocking.
- Like socket select.



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.



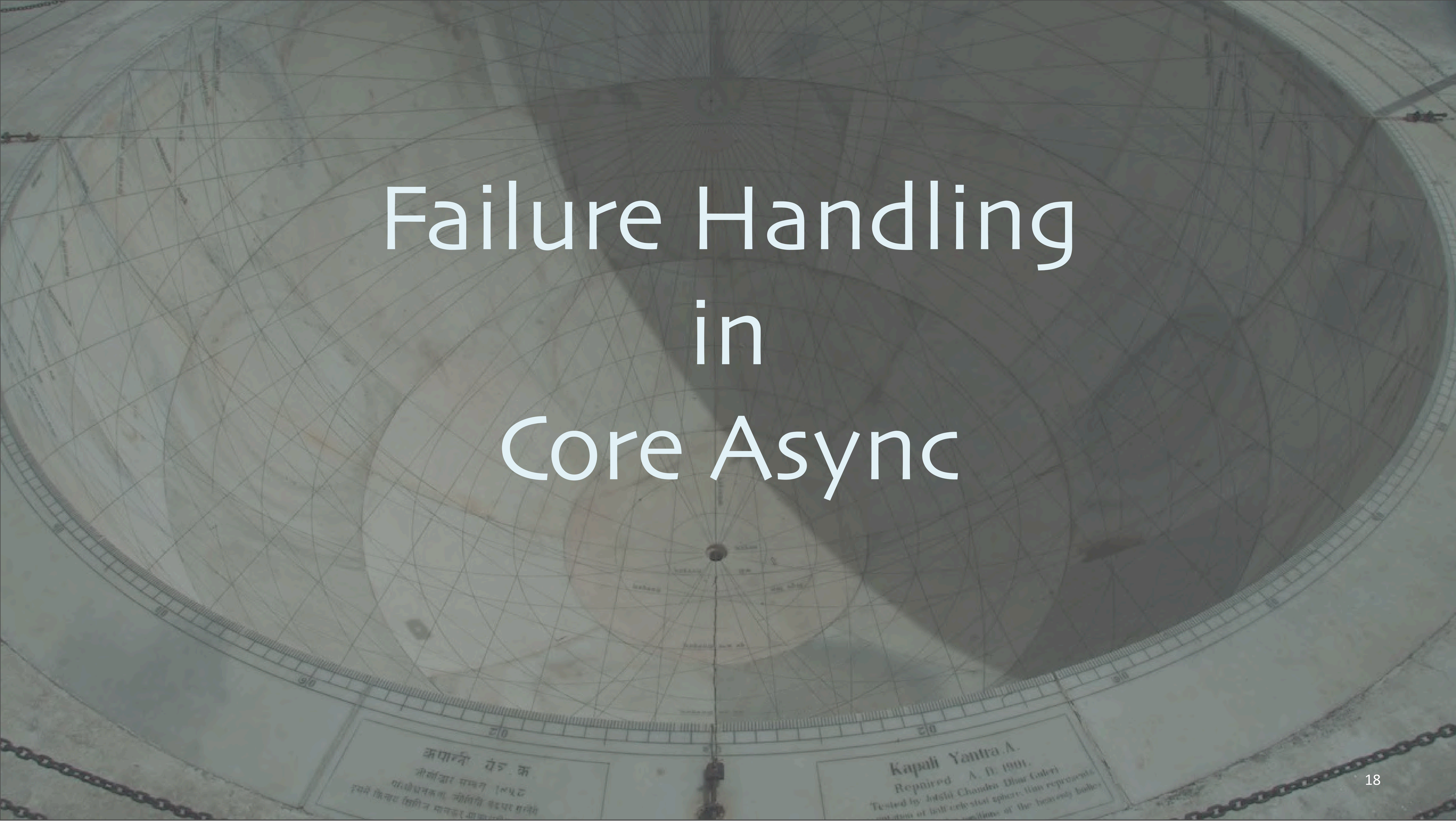
At this point, neither Go nor Core Async have implemented distributed channels.

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





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.




Wednesday, March 18, 15

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

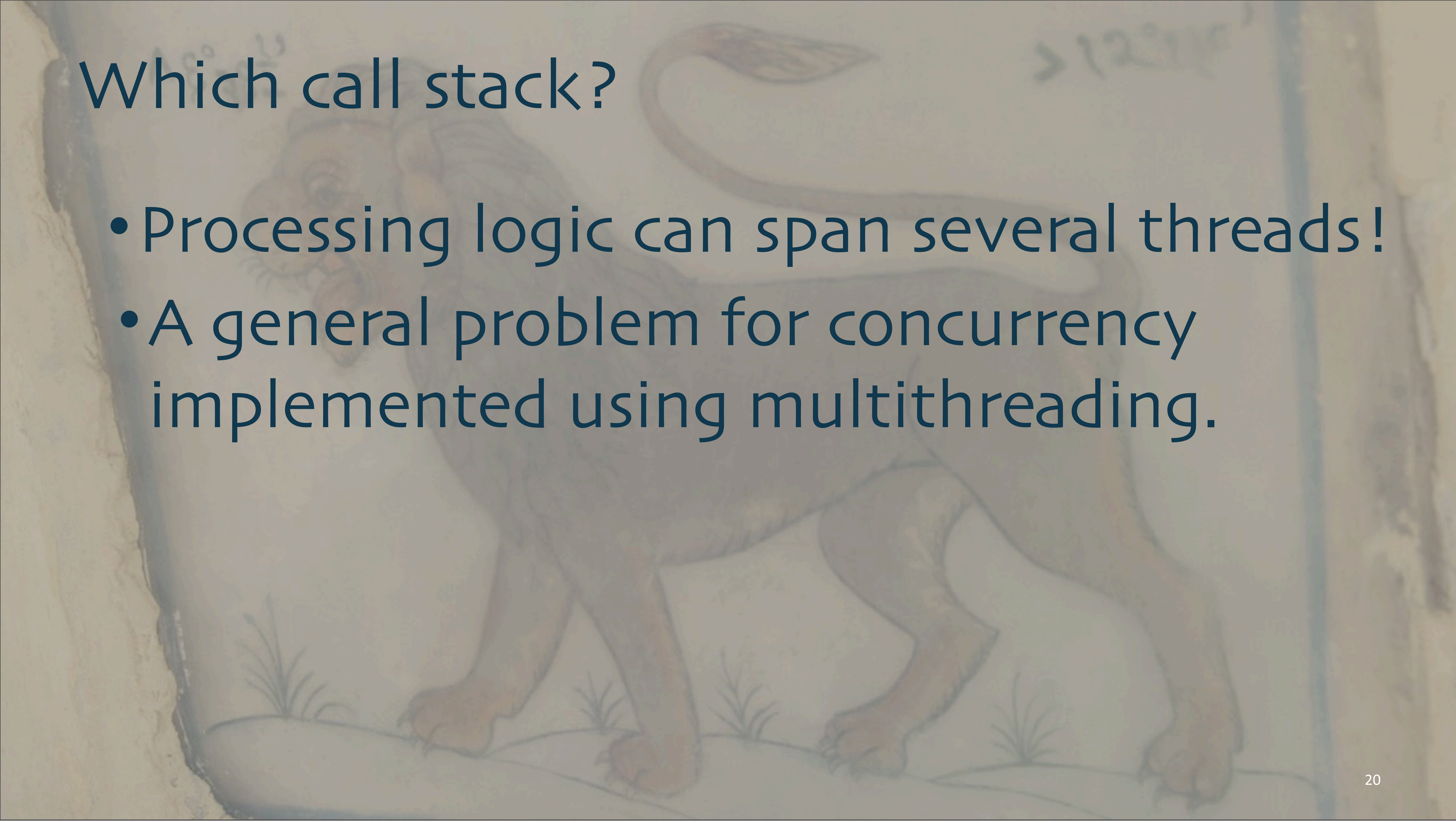
# Channel construction takes an optional exception function.

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





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



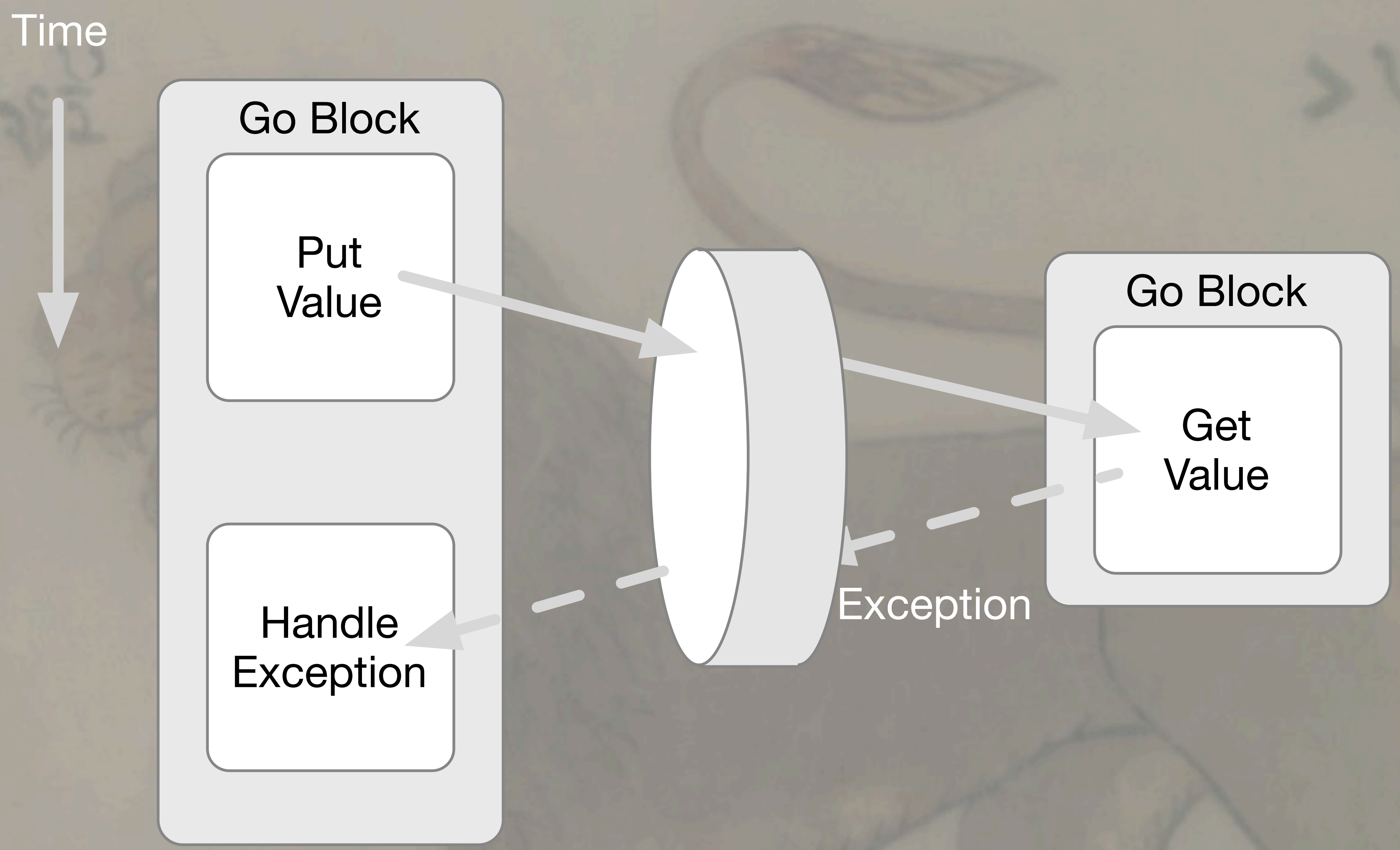
# Which call stack?

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



Wednesday, March 18, 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.

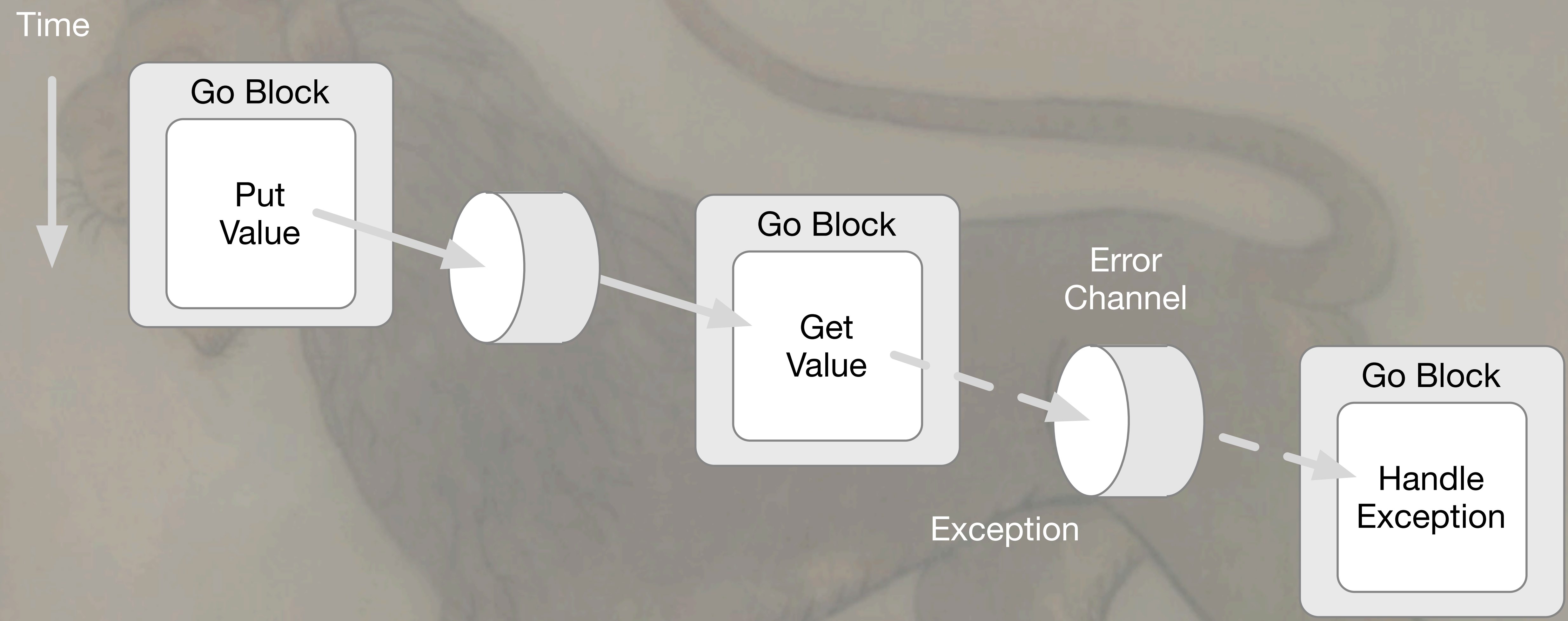



# Propagate exceptions back through the channel.



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.



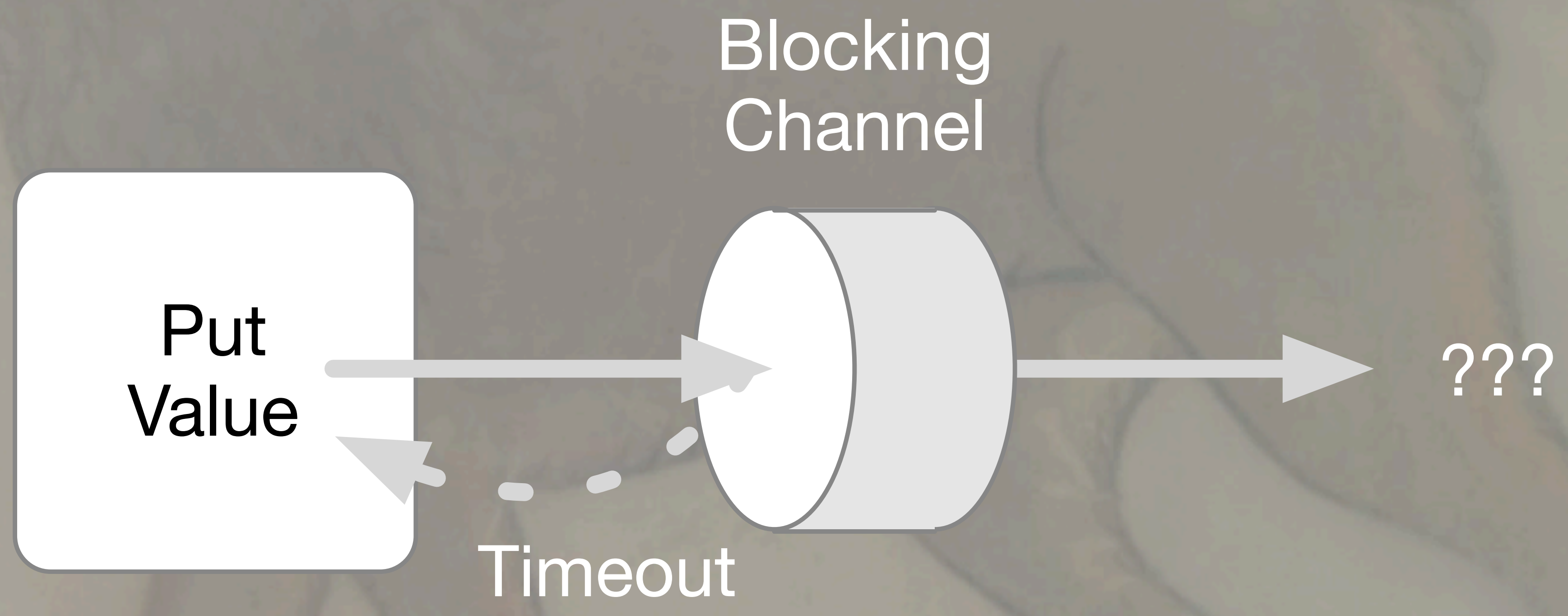

# Propagate exceptions to a special error channel.



Wednesday, March 18, 15

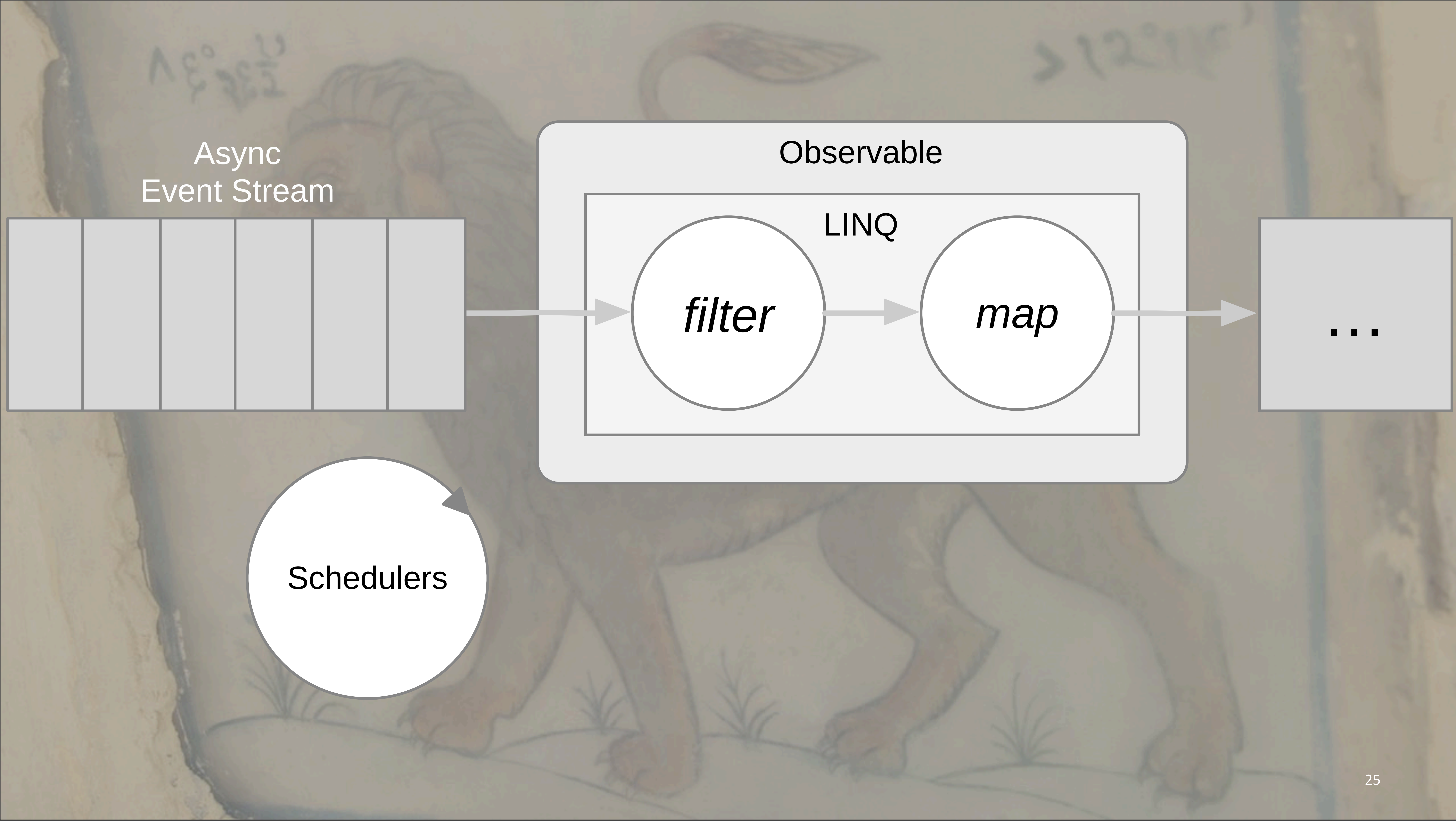
Another possible scheme is to send exceptions down a specialized error channel.

# Deadlock is possible unless timeouts are used.




Wednesday, March 18, 15

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


Async
Event Stream
Observable
LINQ
filter
map
...
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.
```





Adapted from http://www.introtorx.com/content/v1.0.10621.0/11_AdvancedErrorHandling.html
"observer" will watch for raised exceptions. If caught, it will switch to "otherStream". OnNext and OnError generate events onto "stream".

# 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.
```



Wednesday, March 18, 15

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

In this case, we only want to watch for MyException instances. The function is passed the caught exception "ex" and it must return a new stream of the same "MyType".

There is also a Finally method.

Analogous to

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

clauses.





Adapted from http://www.introtorx.com/content/v1.0.10621.0/11_AdvancedErrorHandling.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) {...}
...
```





Adapted from http://www.introtorx.com/content/v1.0.10621.0/11_AdvancedErrorHandling.html
2 of the 3 variants.

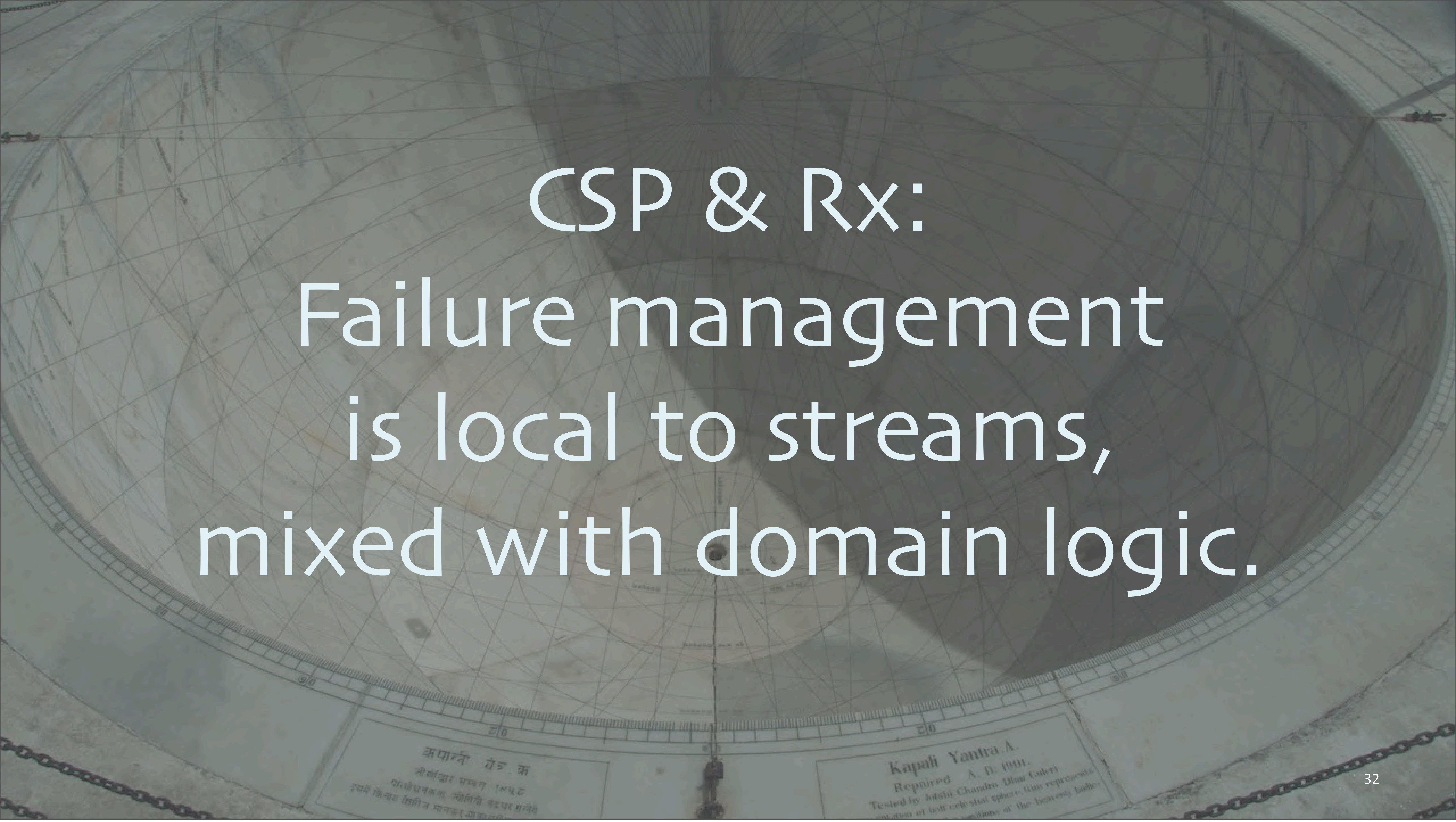
# Retry: Are some exceptions expected, e.g., I/O "hiccups". Keeps trying. Optional max retries.

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





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




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.



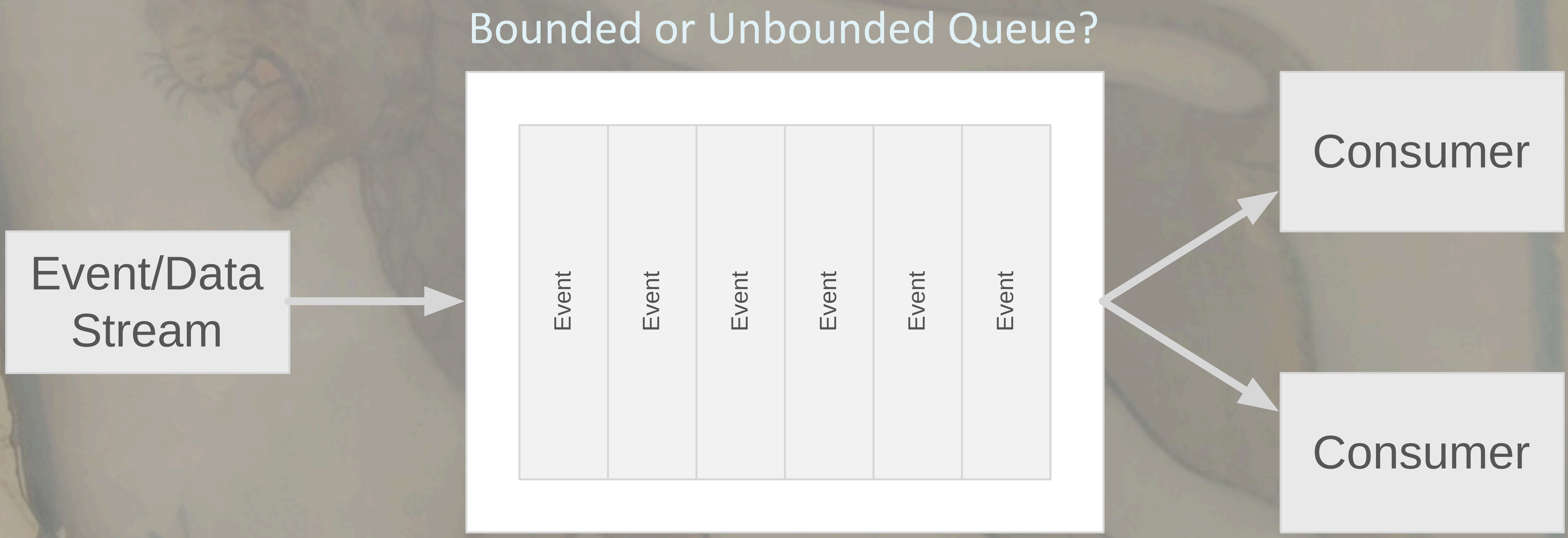

Wednesday, March 18, 15

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


Wednesday, March 18, 15

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

# Reactive Streams



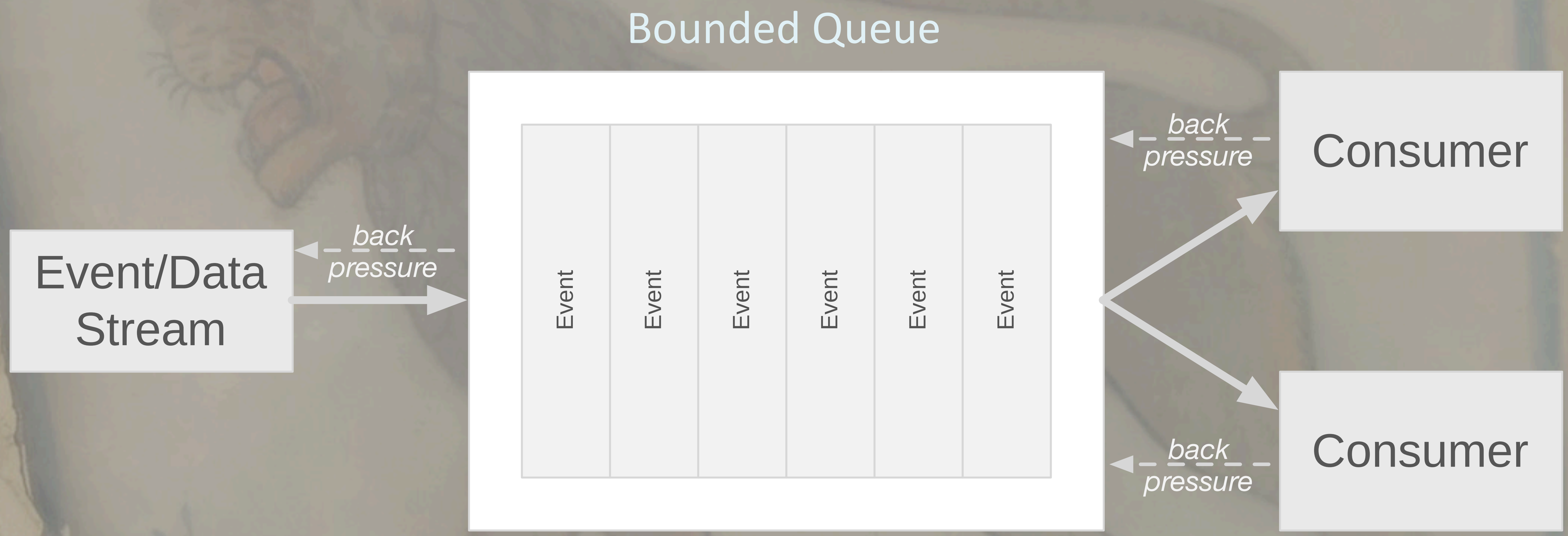

Streams (data flows) are a natural model for many distributed problems, i.e., one-way CSP channels at scale.





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




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





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.





Benefits of back pressure.

# Back pressure:

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

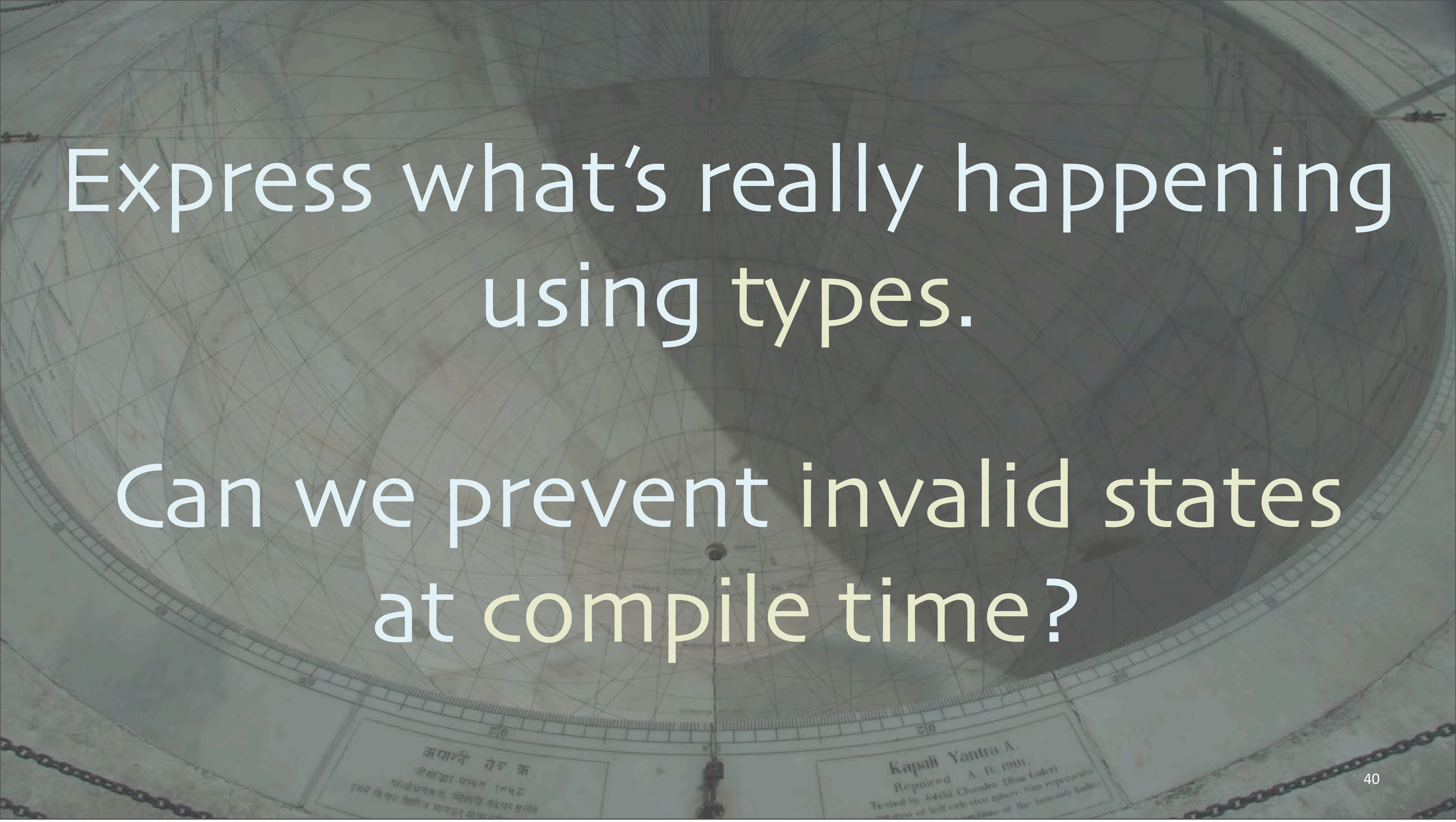




Clarification of an implementation detail.


Wednesday, March 18, 15

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








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 = {...}
}
```



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... */ }
...
```



Wednesday, March 18, 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.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.

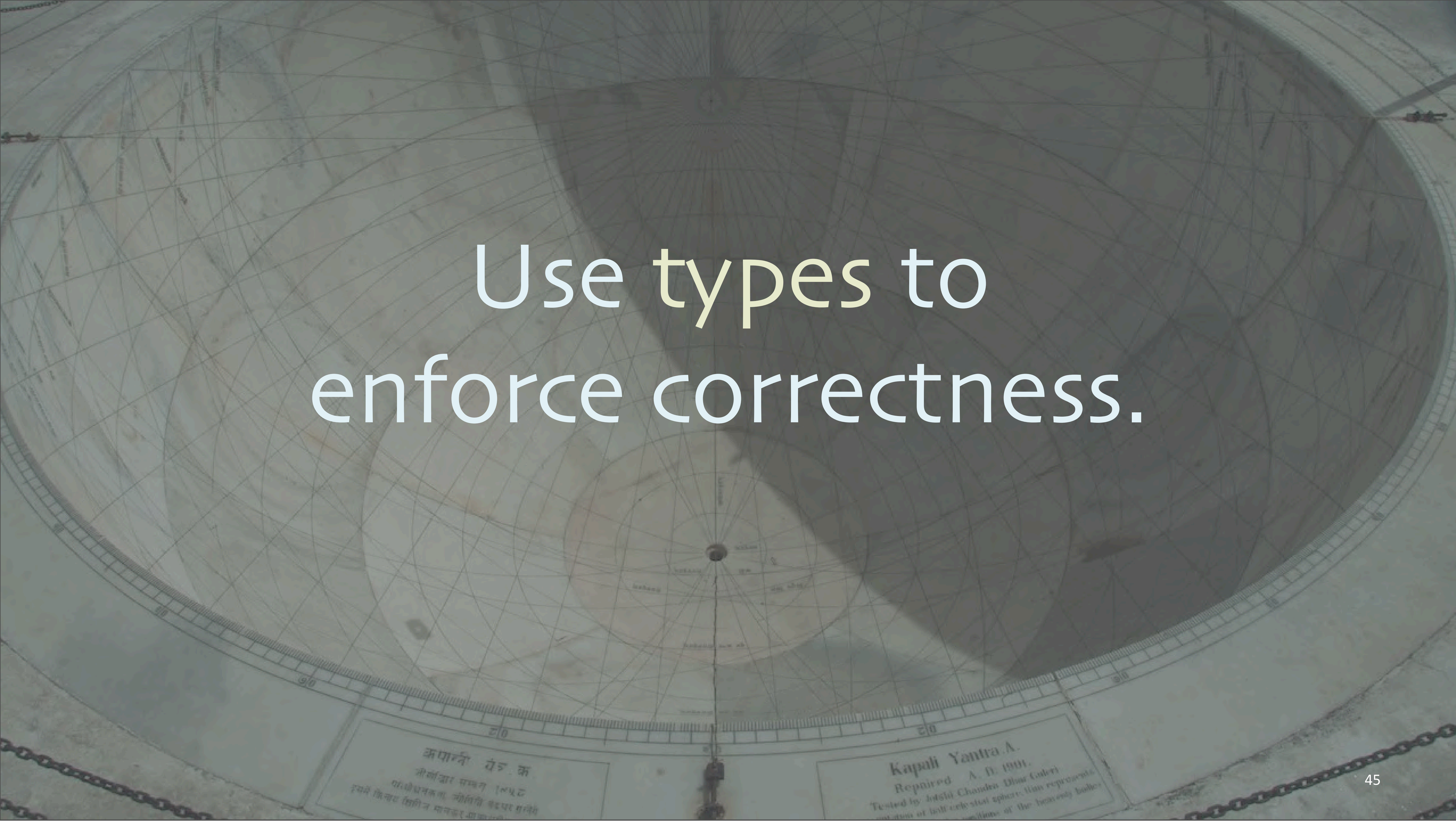
```
val accountId = ...
val ordersFuture = ordersForAccount(accountId)

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





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




Wednesday, March 18, 15

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


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.

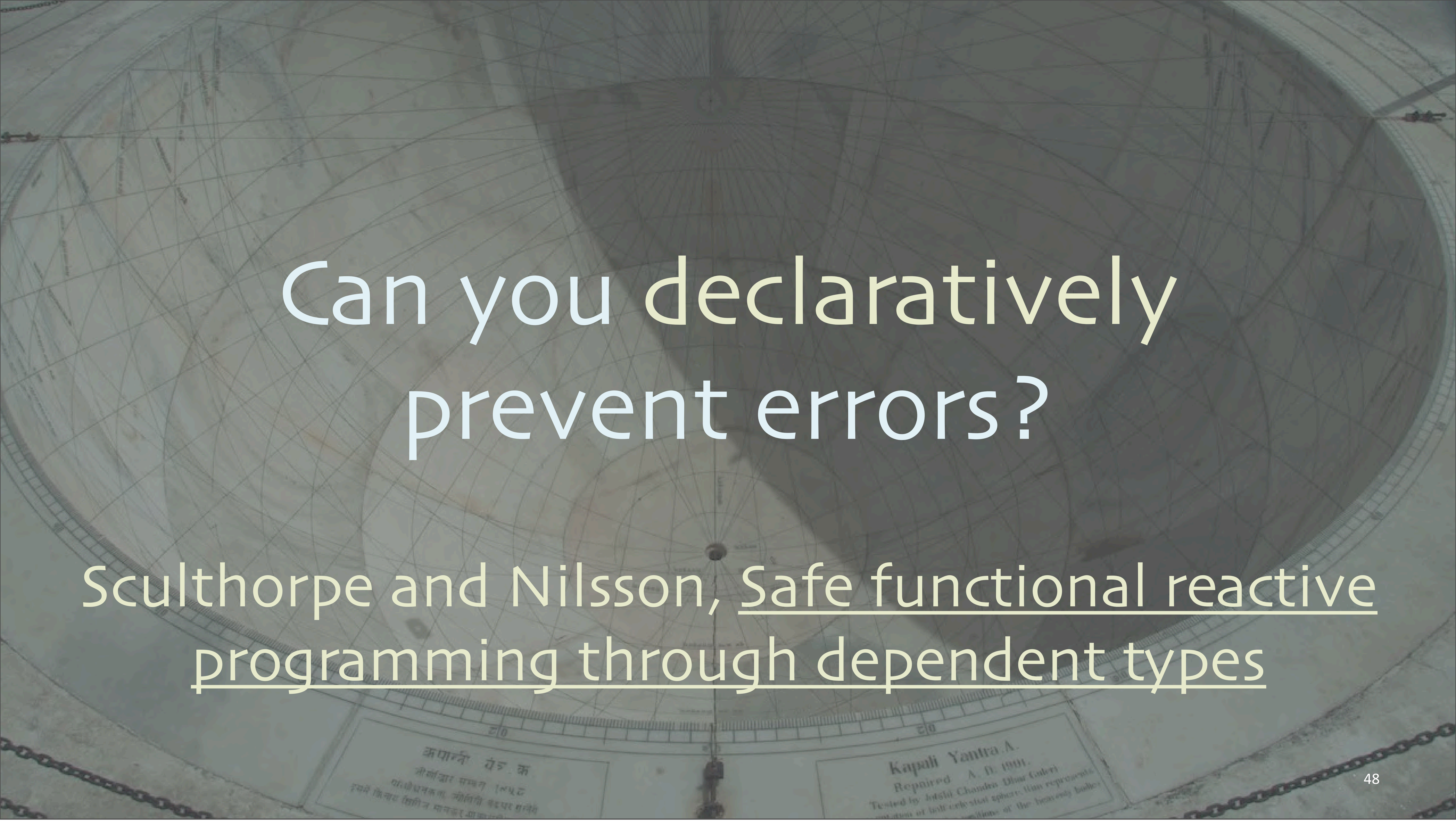
```
Reactor.flow { reactor =>
  val path = new Path(
   (reactor.await(mouseDown)).position)
  reactor.loopUntil(mouseUp) {
    val m = reactor.awaitNext(mouseMove)
    path.lineTo(m.position)
    draw(path)
  }
  path.close()
  draw(path)
}
```

From *Deprecating the Observer Pattern with Scala.React*.



Wednesday, March 18, 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.



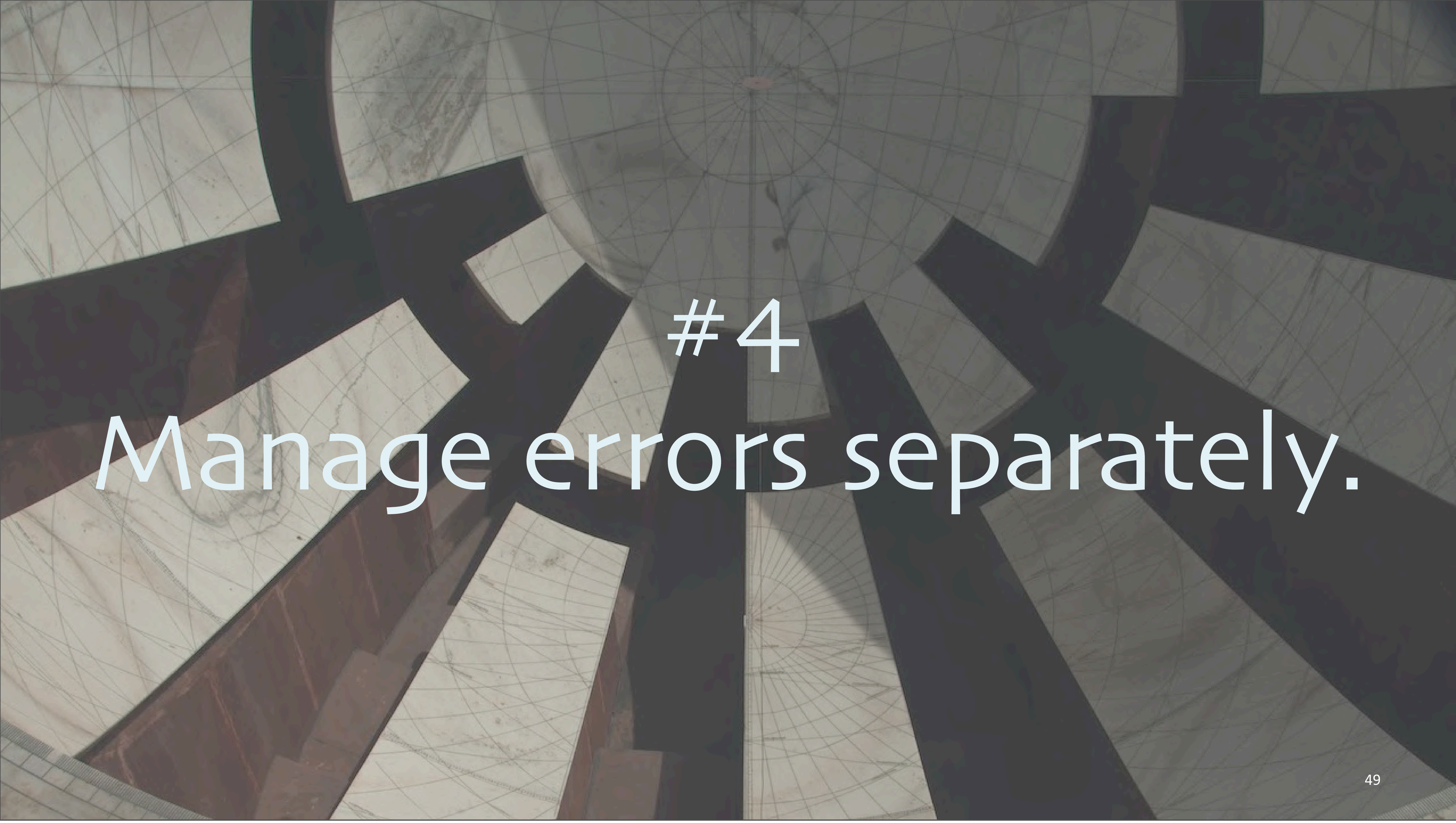
# Can you declaratively prevent errors?

Sculthorpe and Nilsson, Safe functional reactive programming through dependent types





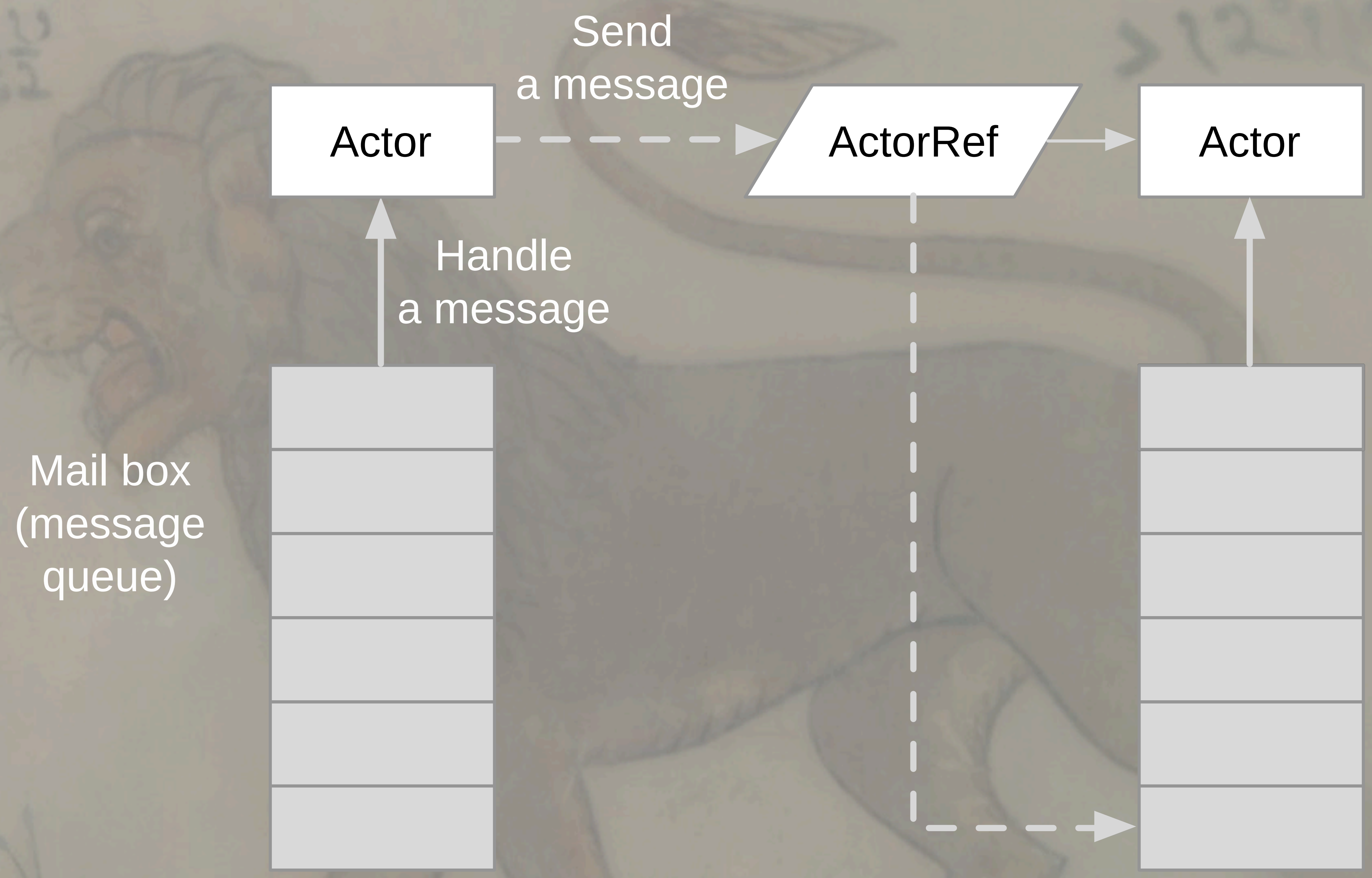
True to its Haskell routes, FRP tries to use the type system to explicitly
http://dl.acm.org/citation.cfm?doid=1596550.1596558




Wednesday, March 18, 15

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

# Actor Model

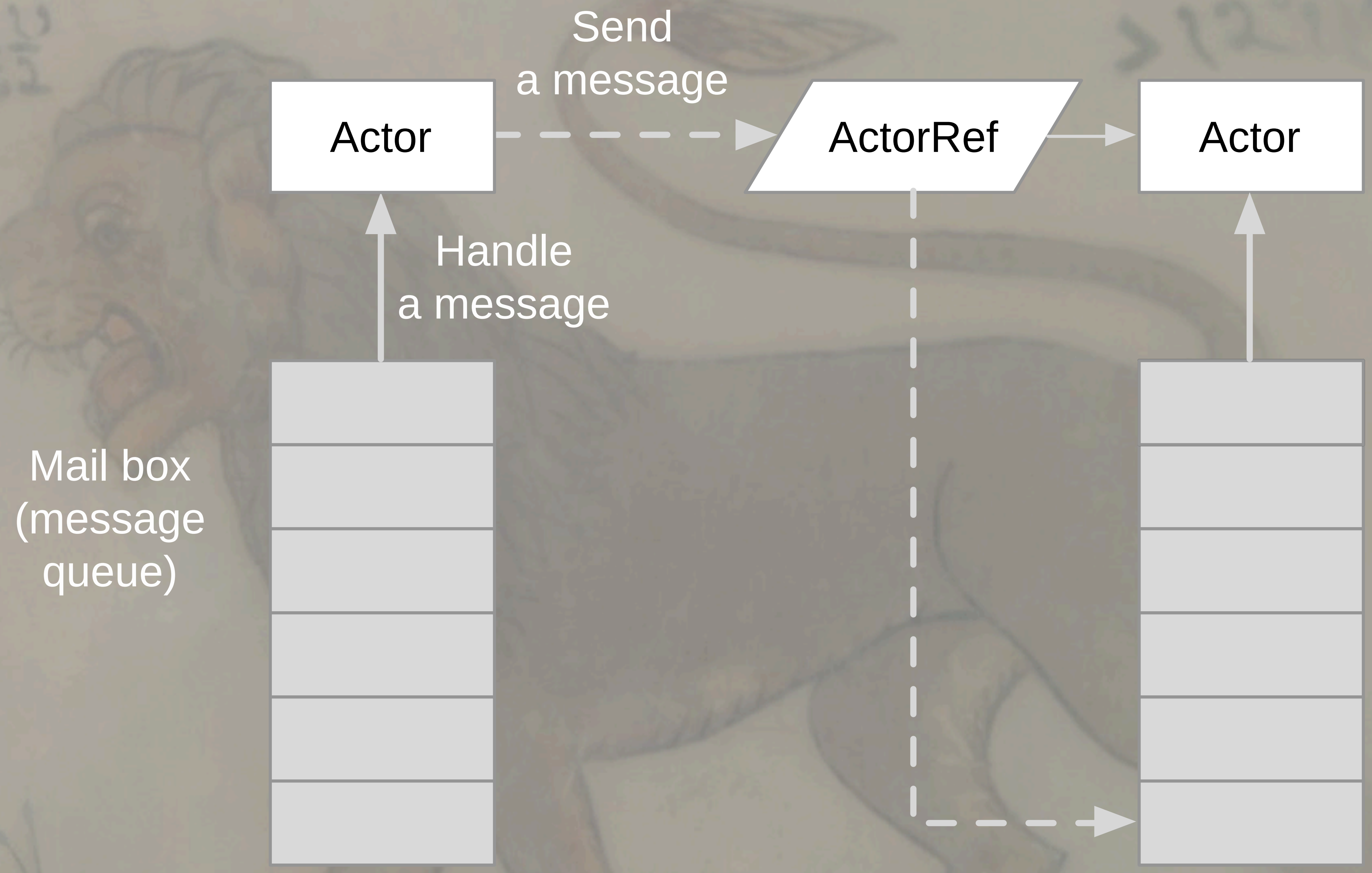



# (Akka example - akka.io)



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.



Wednesday, March 18, 15

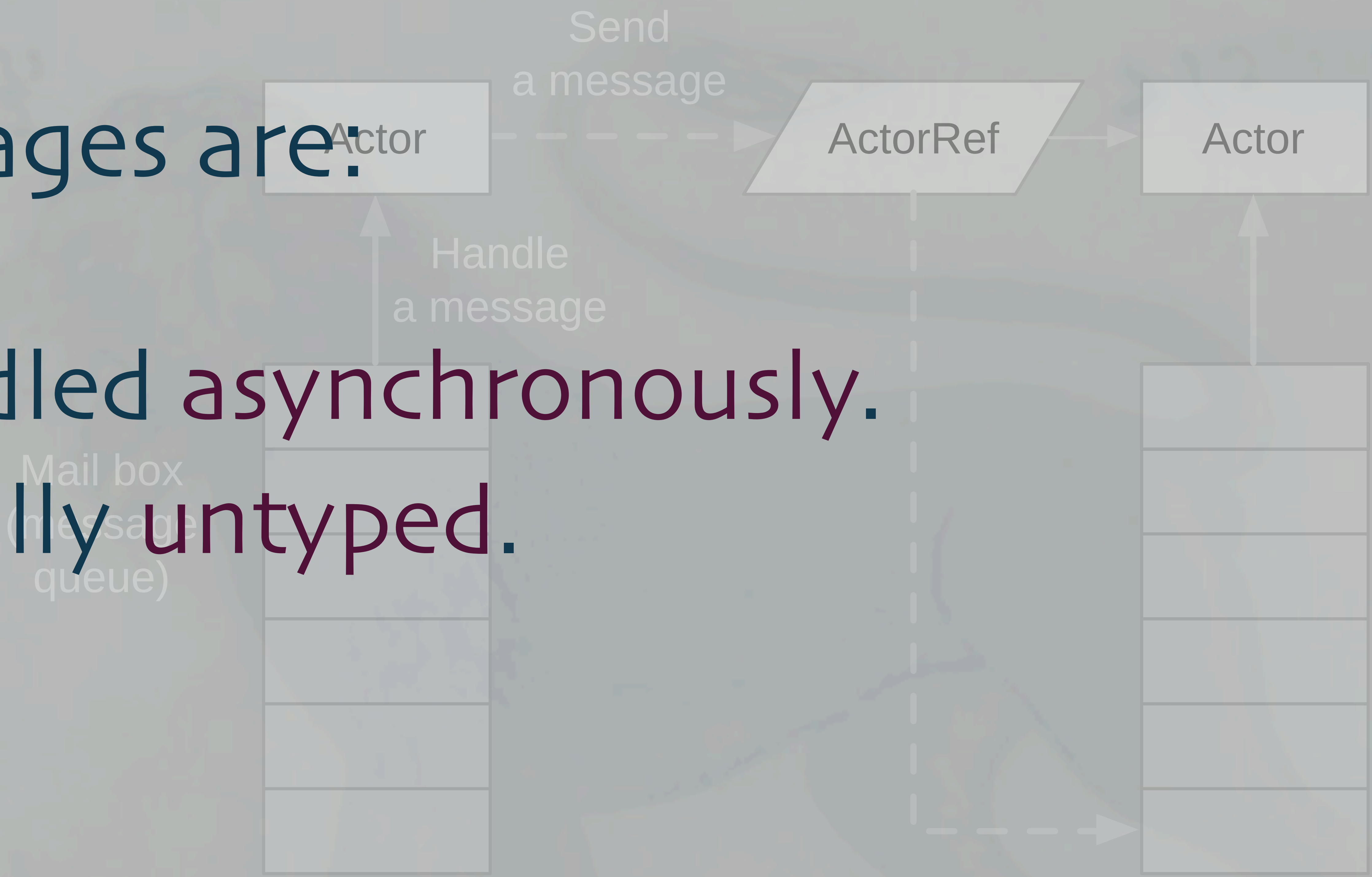
This is how they look in Akka, where there is a layer of indirection, the ActorRef, between actors. This helps with the drawback that actors know each other's identities, but mostly it's there to make the system more resilient, where a failed actor can be restarted while keeping the same ActorRef that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

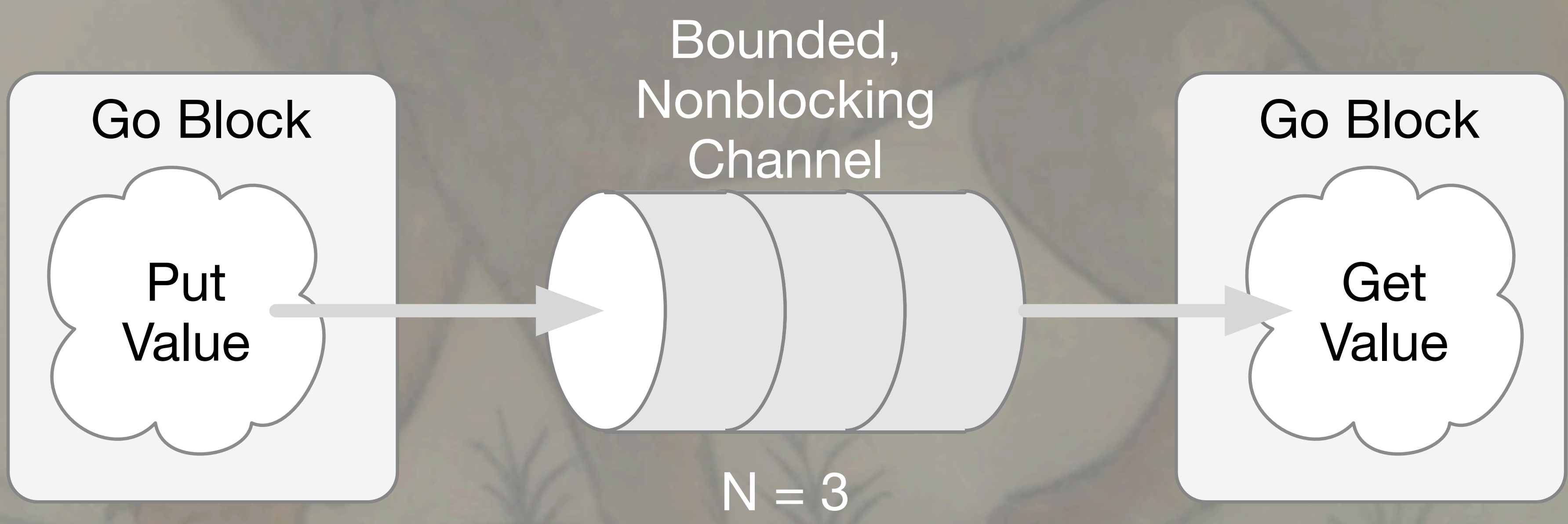
- Handled asynchronously.
- Usually untyped.



# CSP and Actors are dual

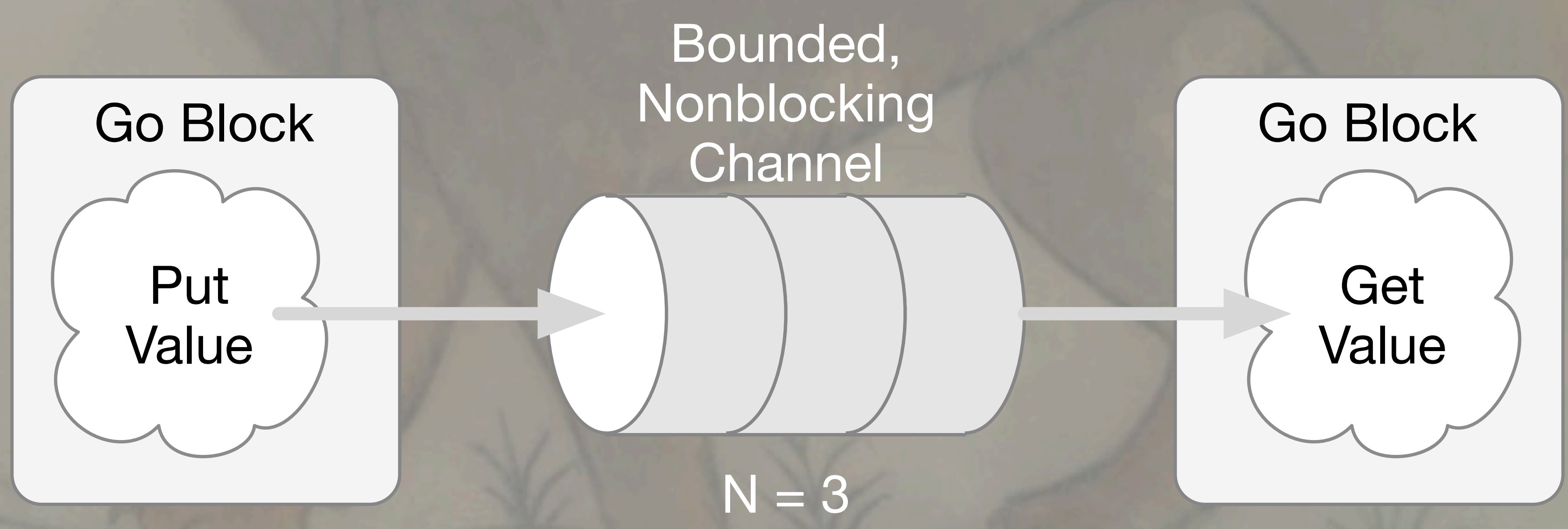
# CSP Processes are anonymous

# … while actors have identities.



# CSP messaging is synchronous

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



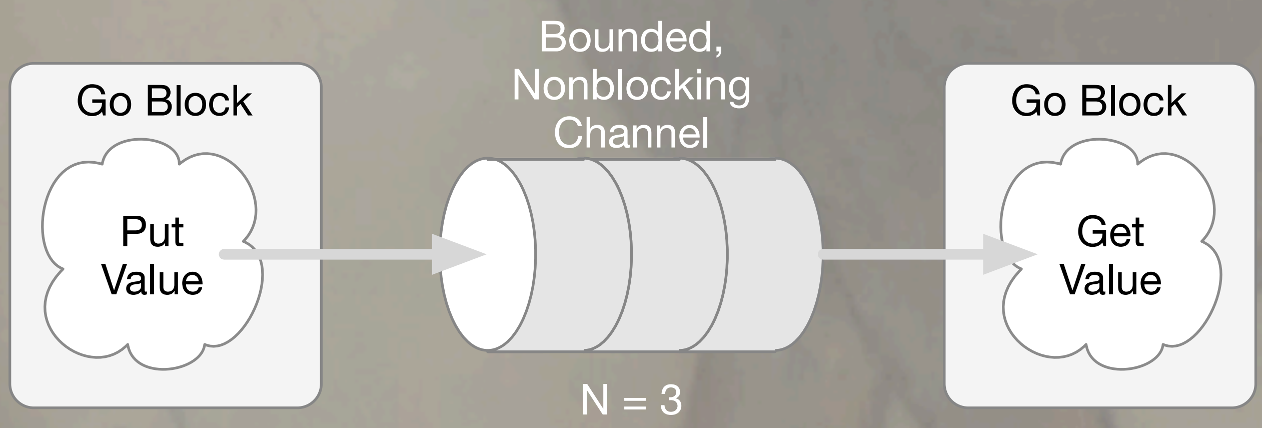


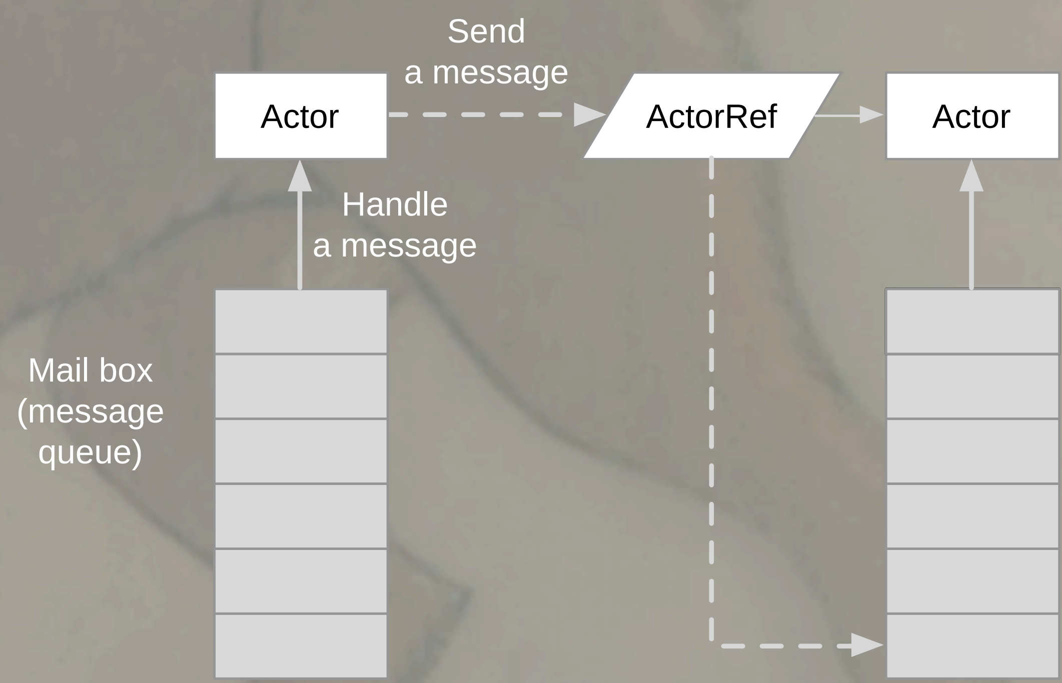

Wednesday, March 18, 15

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


# … but CSP and Actors can implement each other

# An actor mailbox looks a lot like a channel.






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.

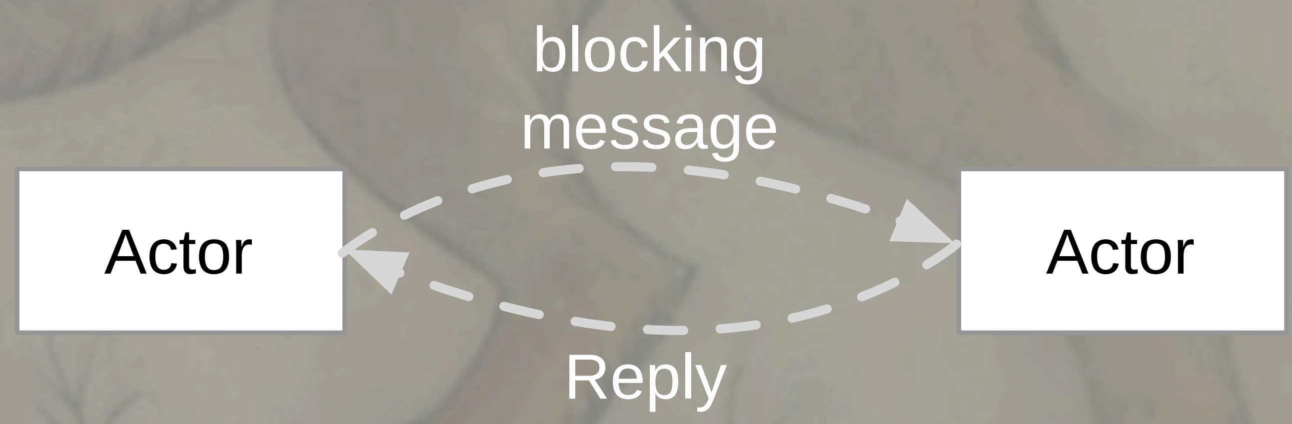
An actor can be used as a channel , i.e., a "message broker".

# 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.



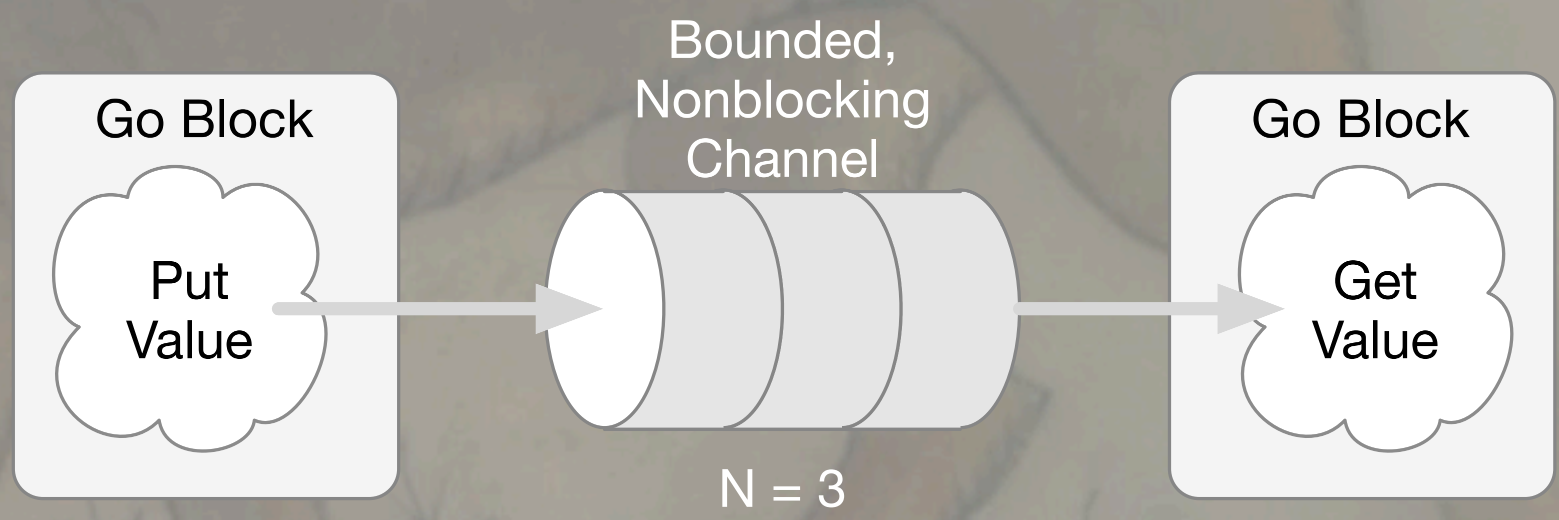





Most actor systems provide a blocking message send primitive where the “thread” blocks until an answer message is received.

# CSP messaging is synchronous

## Buffered channels behave asynchronously.

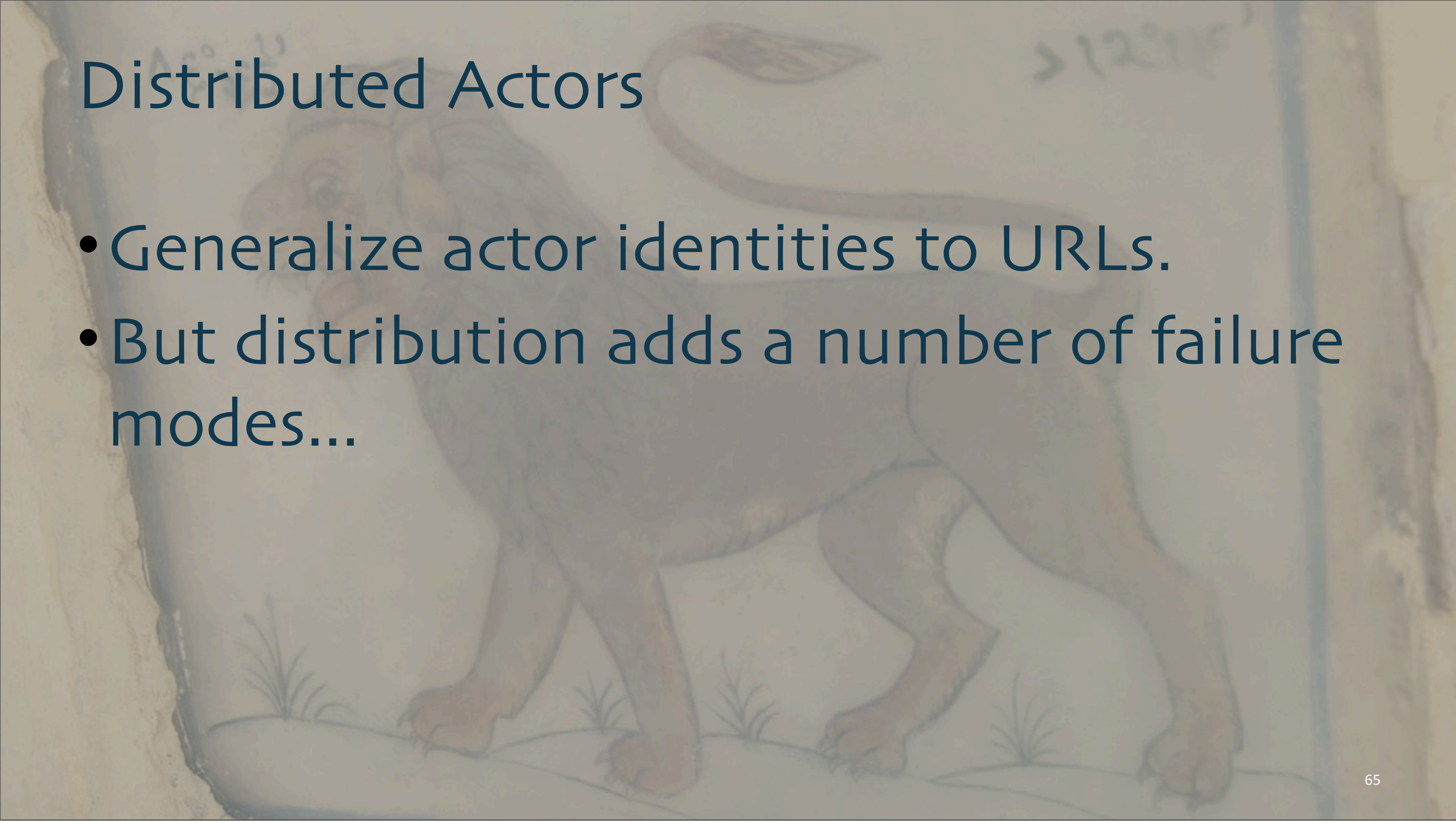





Wednesday, March 18, 15

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


Erlang
and
Akka



# Distributed Actors

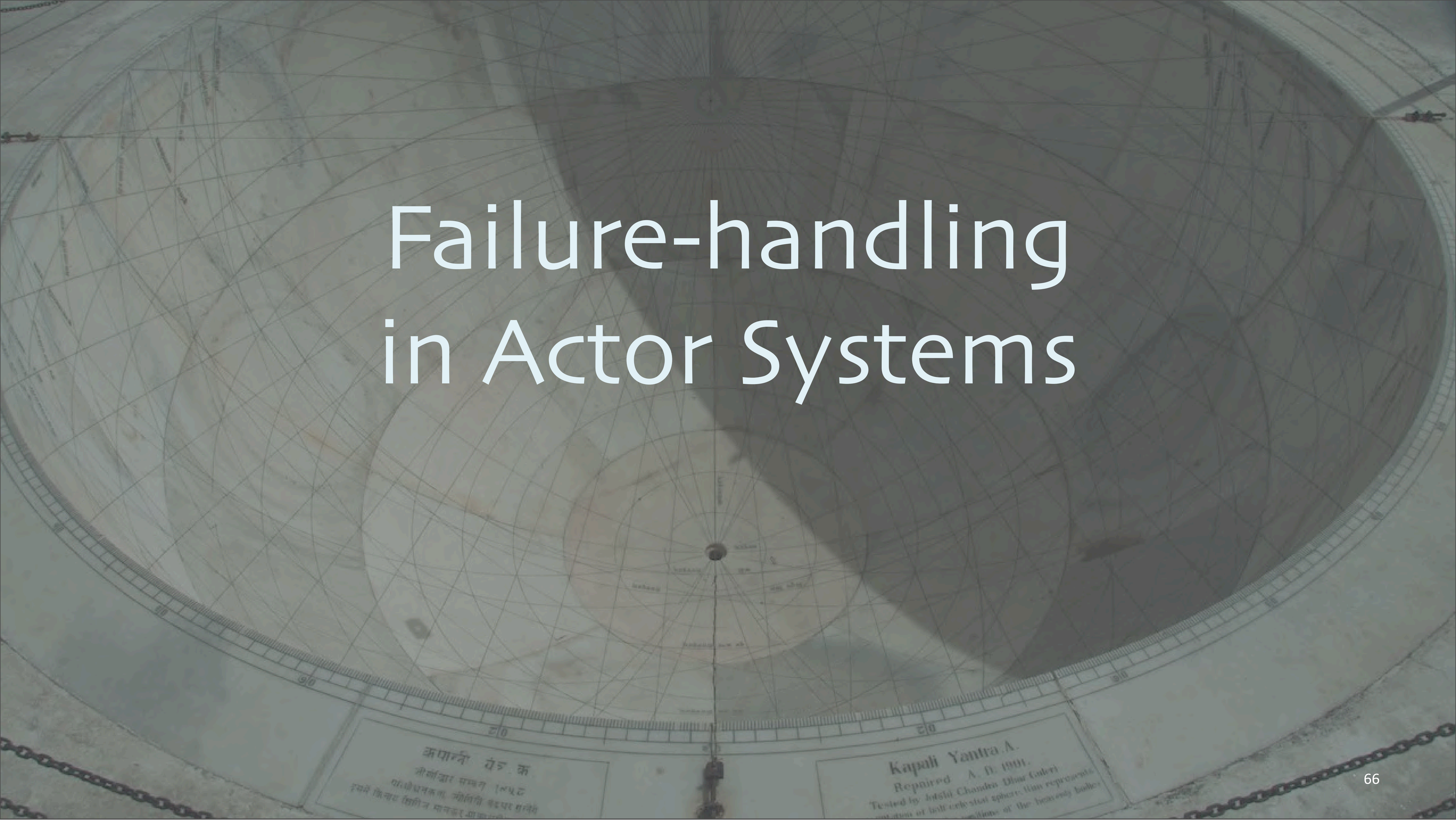
- Generalize actor identities to URLs.
- But distribution adds a number of failure modes...



Wednesday, March 18, 15

URL vs. URI?? See http://danielmiessler.com/study/url_vs_uri/

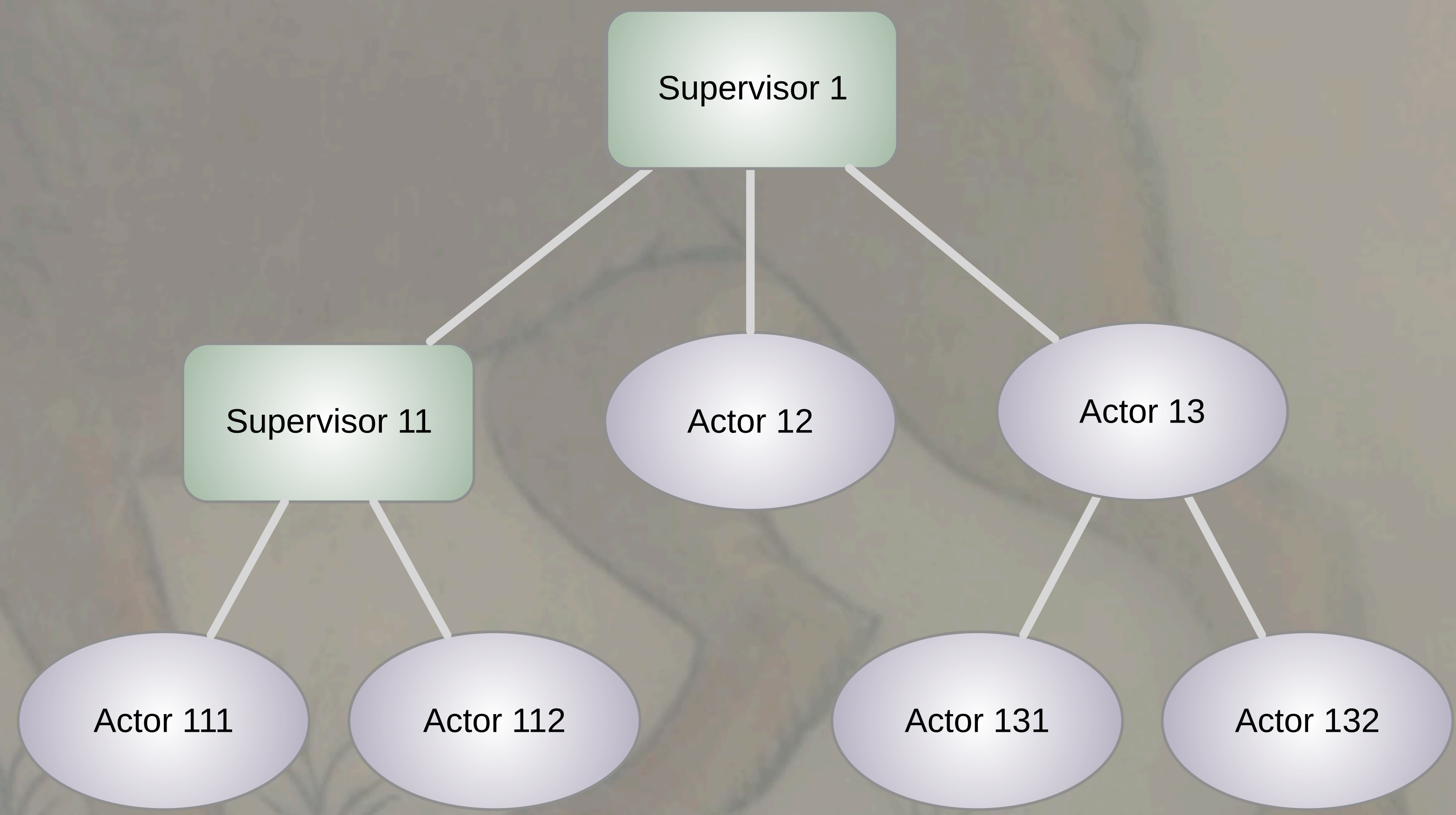
# Failure-handling in Actor Systems



# Let it Crash!

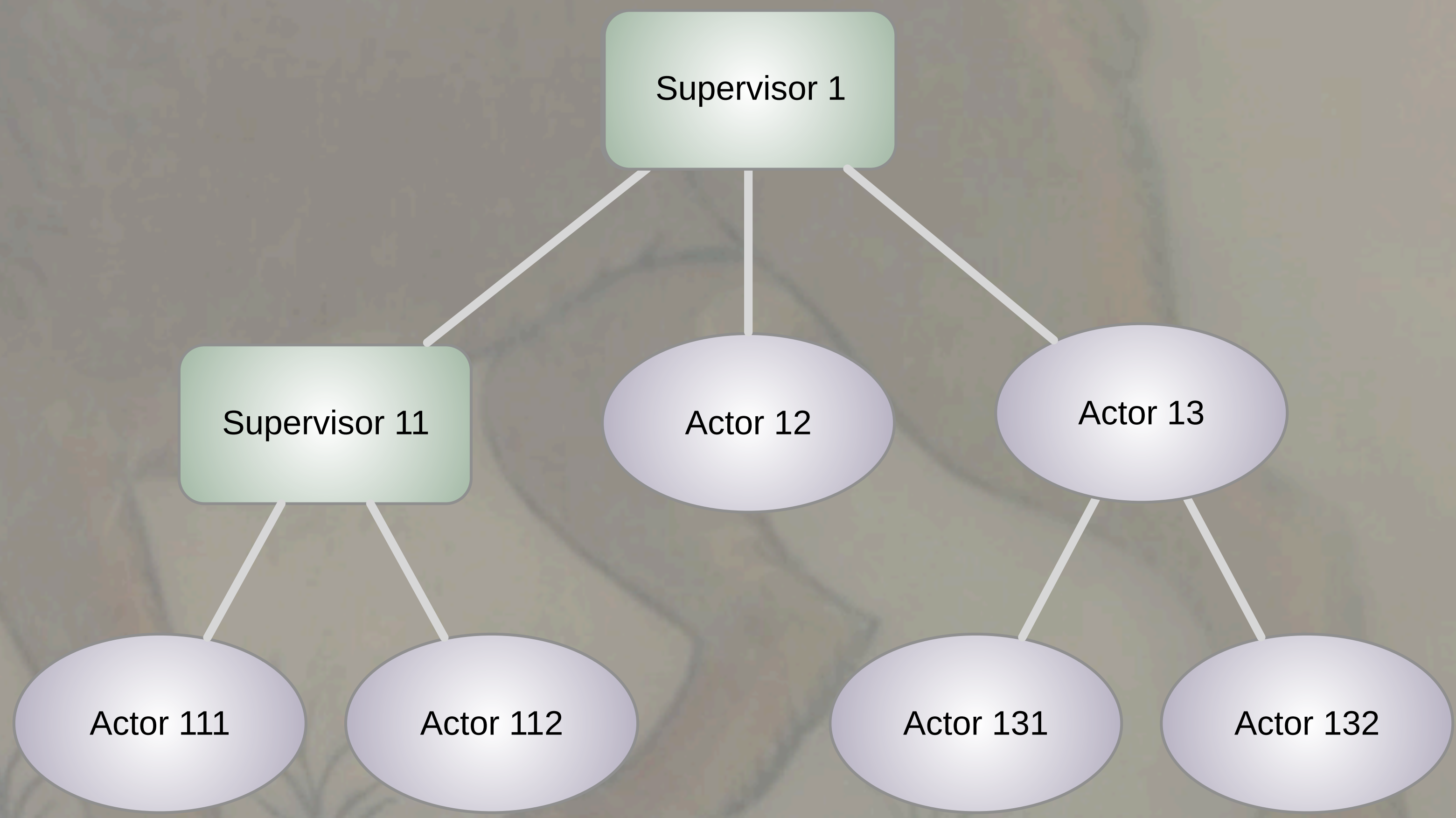
# Erlang introduced supervisors

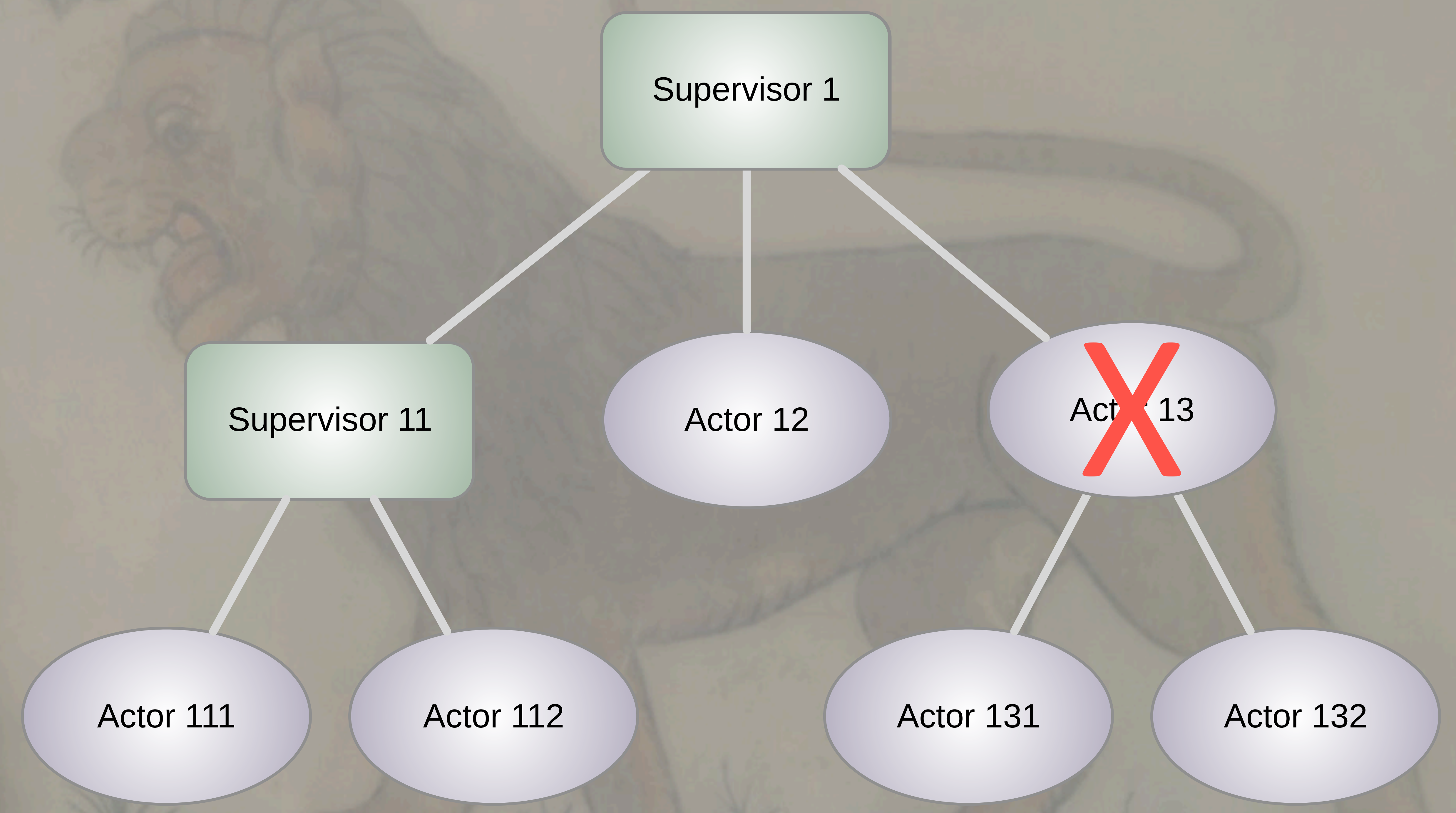
A hierarchy of actors that manage each "worker" actor's lifecycle.

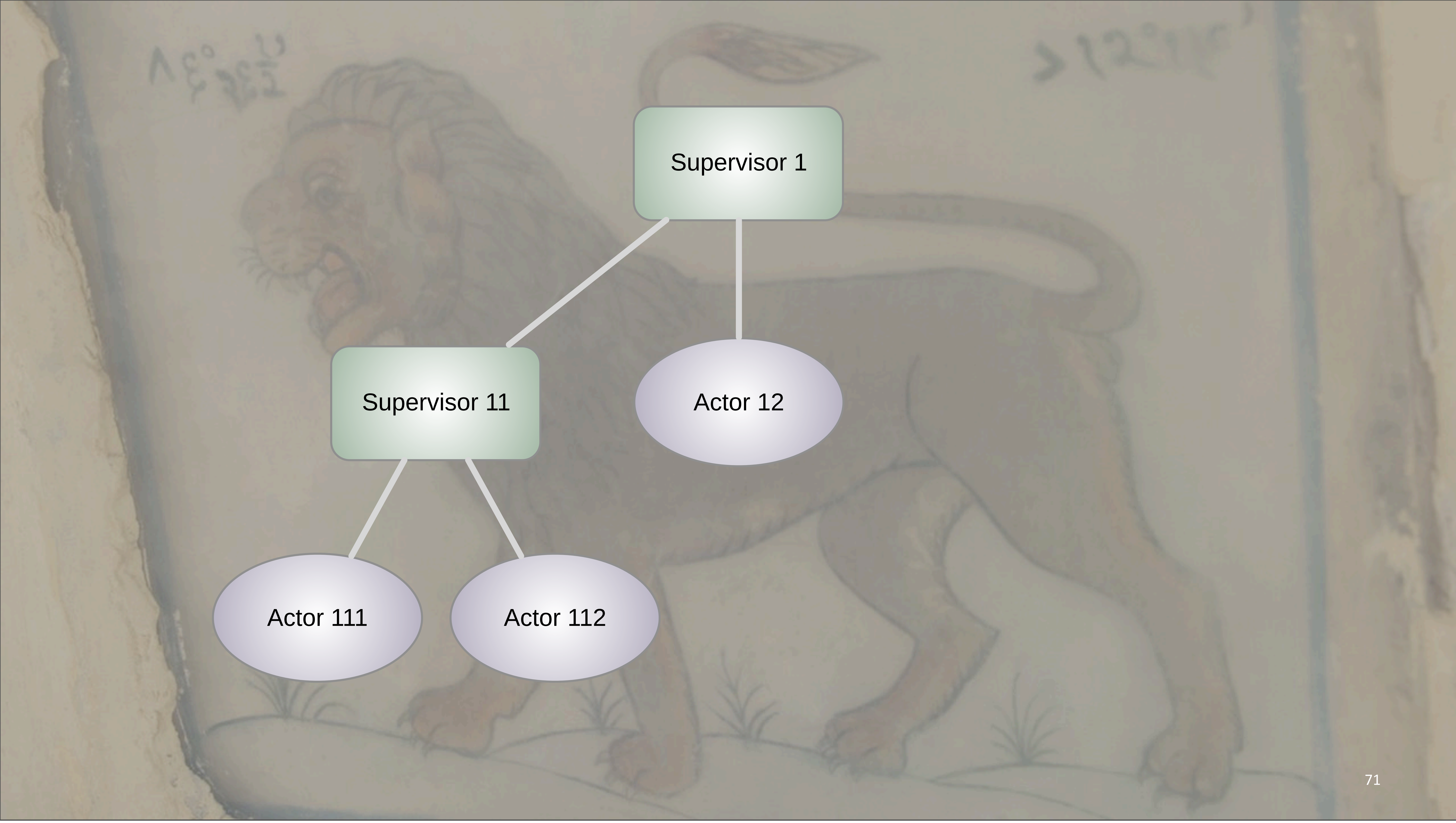


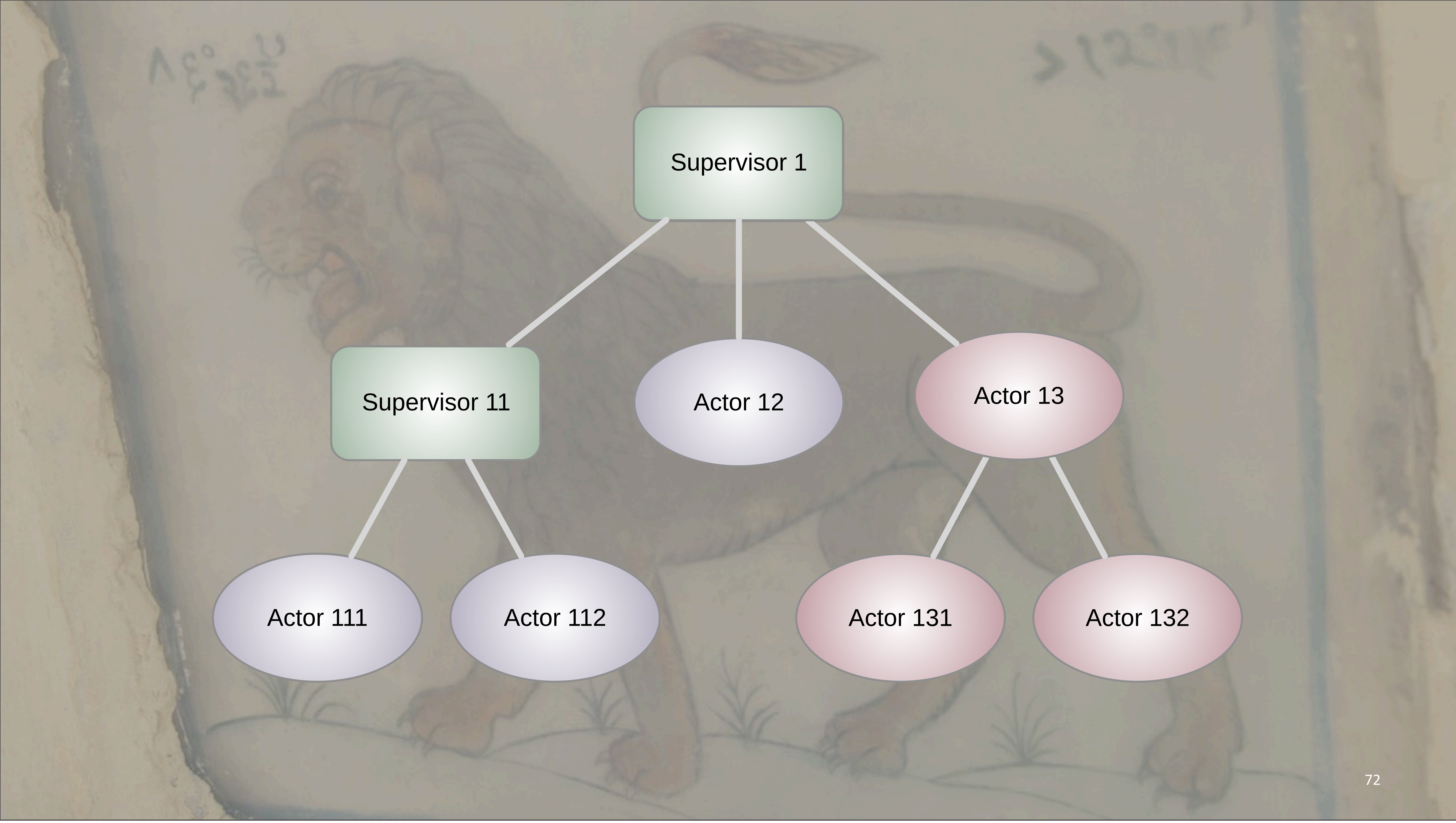
# Erlang introduced supervisors

Generalizes nicely to distributed actor systems.




Supervisor 1
Supervisor 11
Actor 12
Act 13
Actor 111
Actor 112
Actor 131
Actor 132


Supervisor 1
Supervisor 11
Actor 12
Actor 111
Actor 112



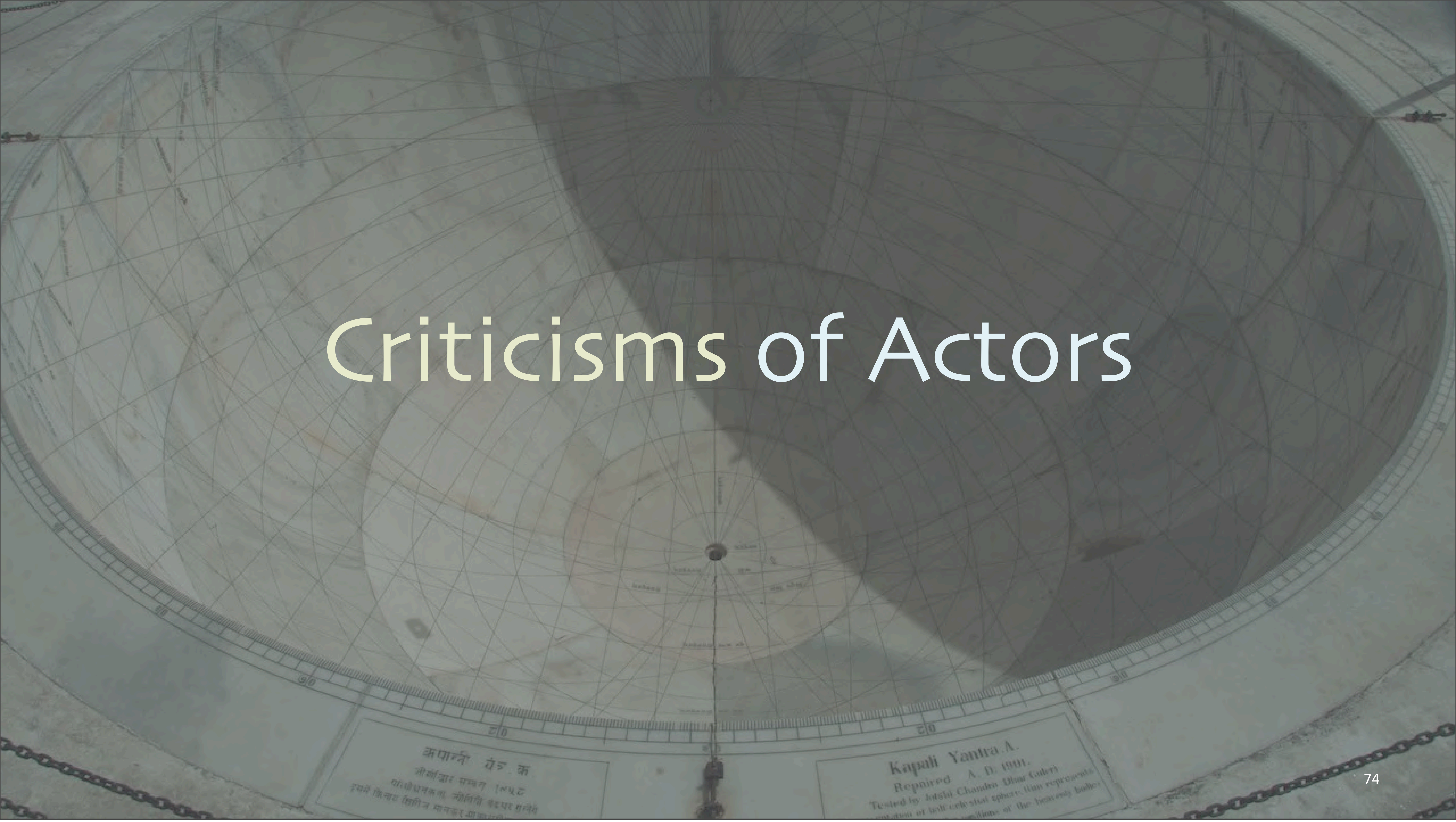



Wednesday, March 18, 15

# Advantages

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

# Criticisms of Actors



# 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.





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

# 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”.



Wednesday, March 18, 15

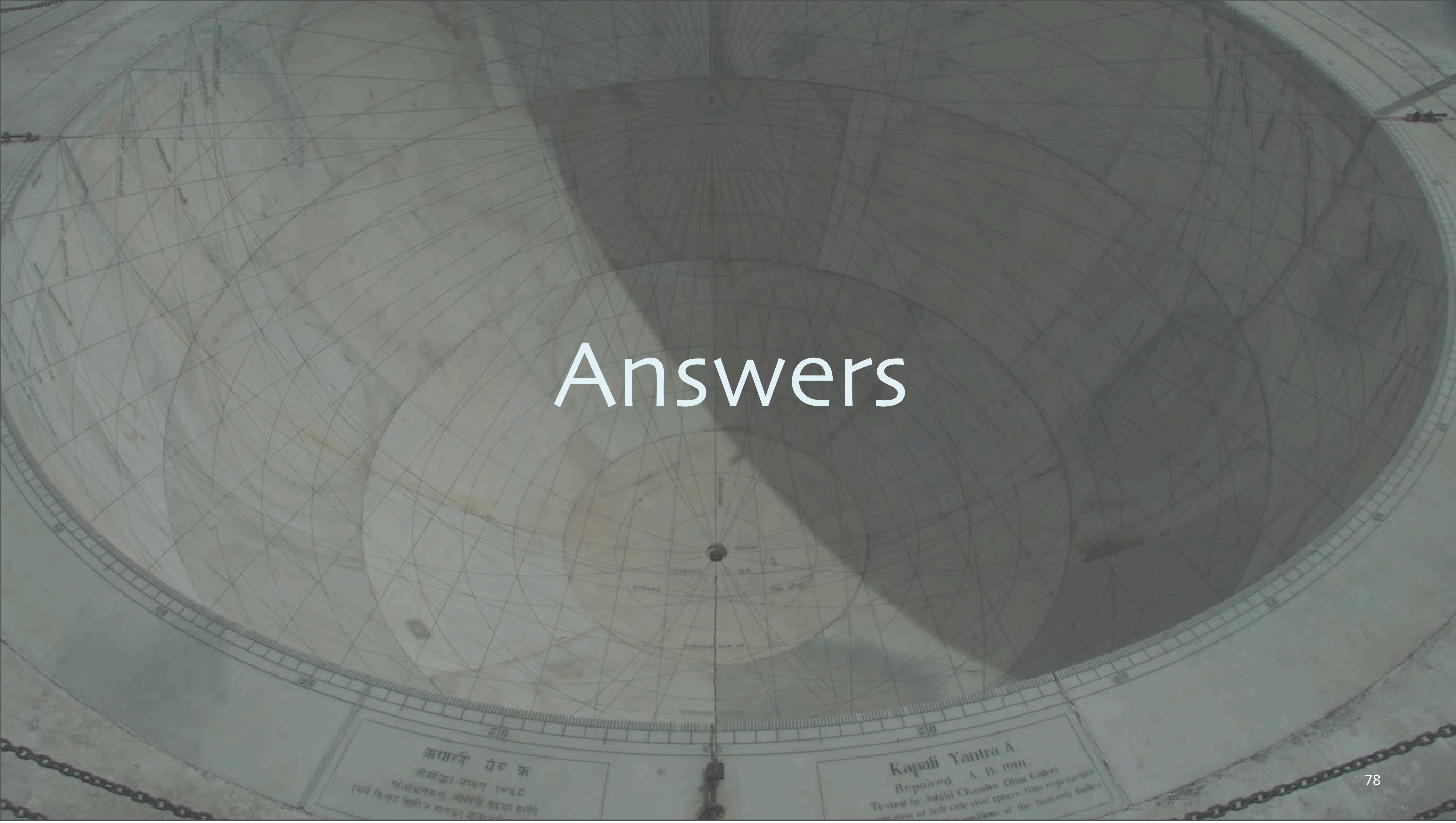
From https://github.com/halgari/clojure-conj-2013-core.async-examples/blob/master/src/clojure_conj_talk/core.clj

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






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.

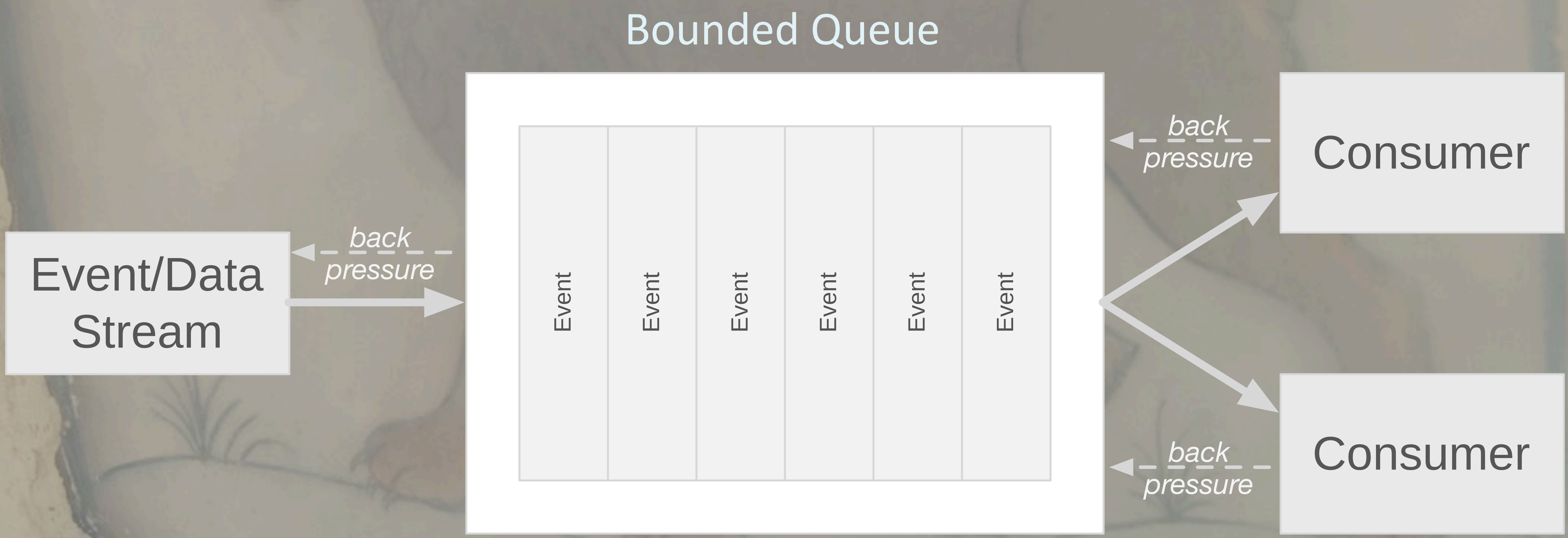




In other words, ignore toy examples.

# 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 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.

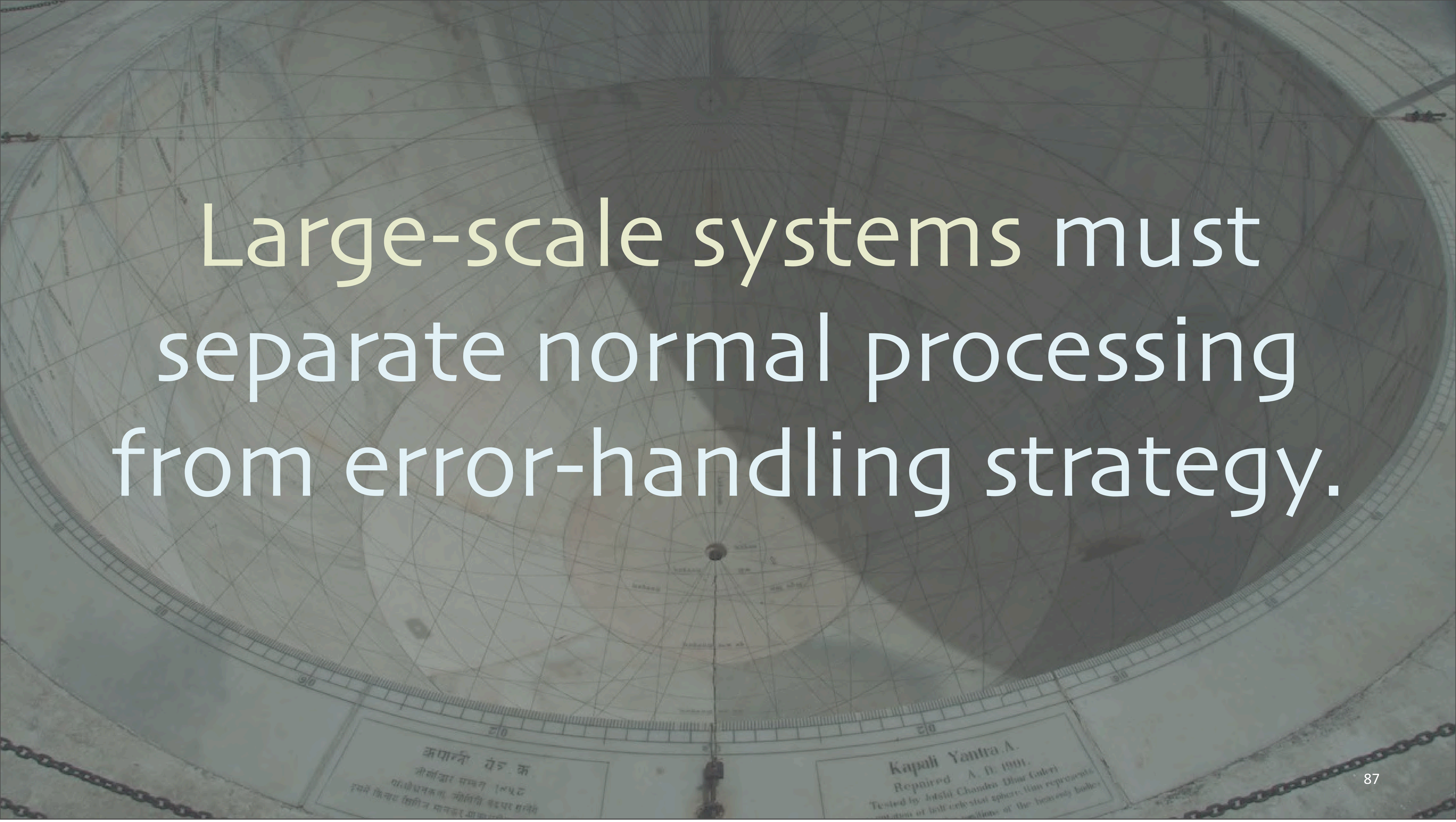
# Actors are untyped

- 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

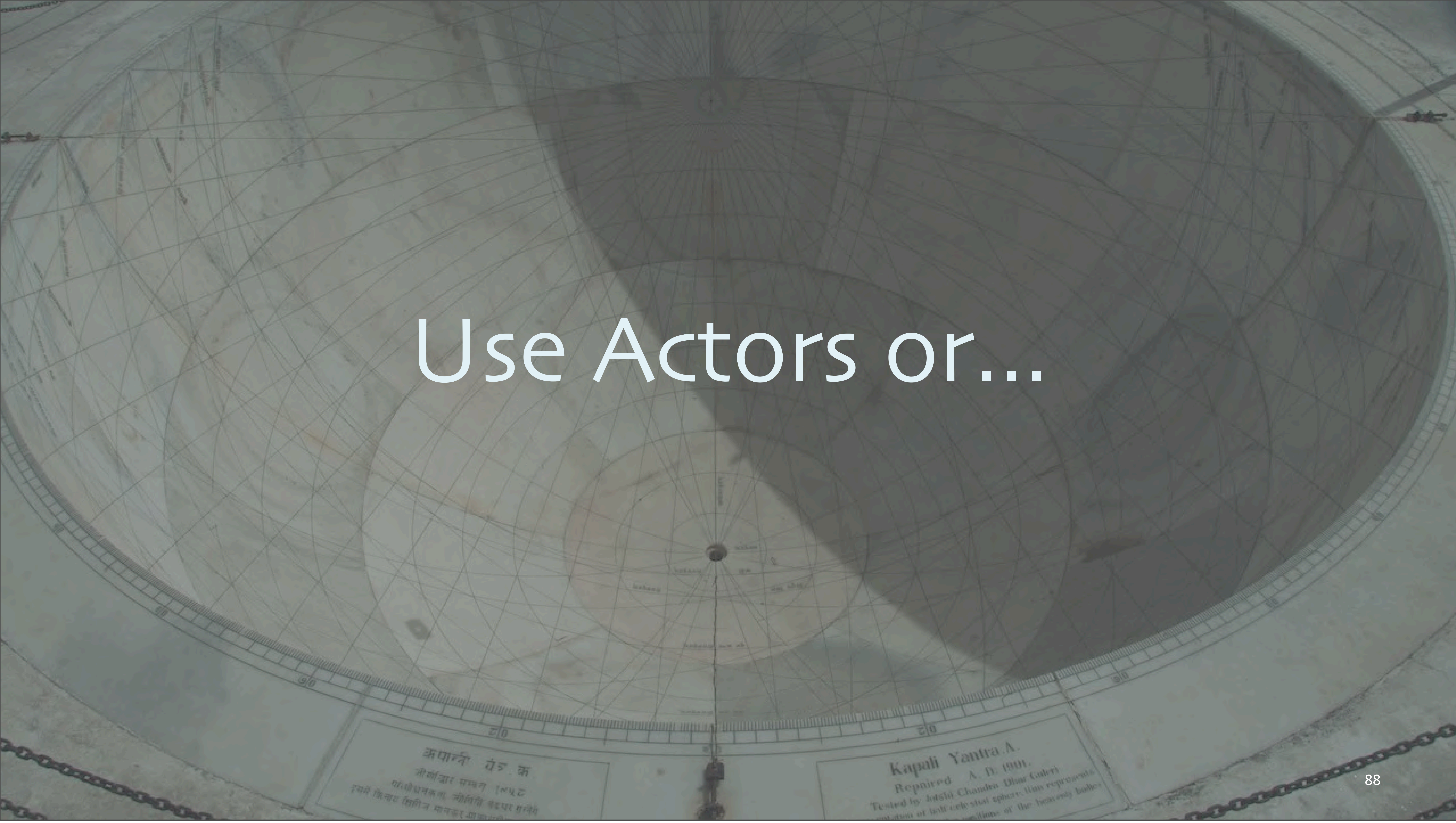
- ... but actually, Akka is adding typed ActorRefs.


How should we handle failures?




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...
Kapali Yantra A.
Repaired A. D. 1901.

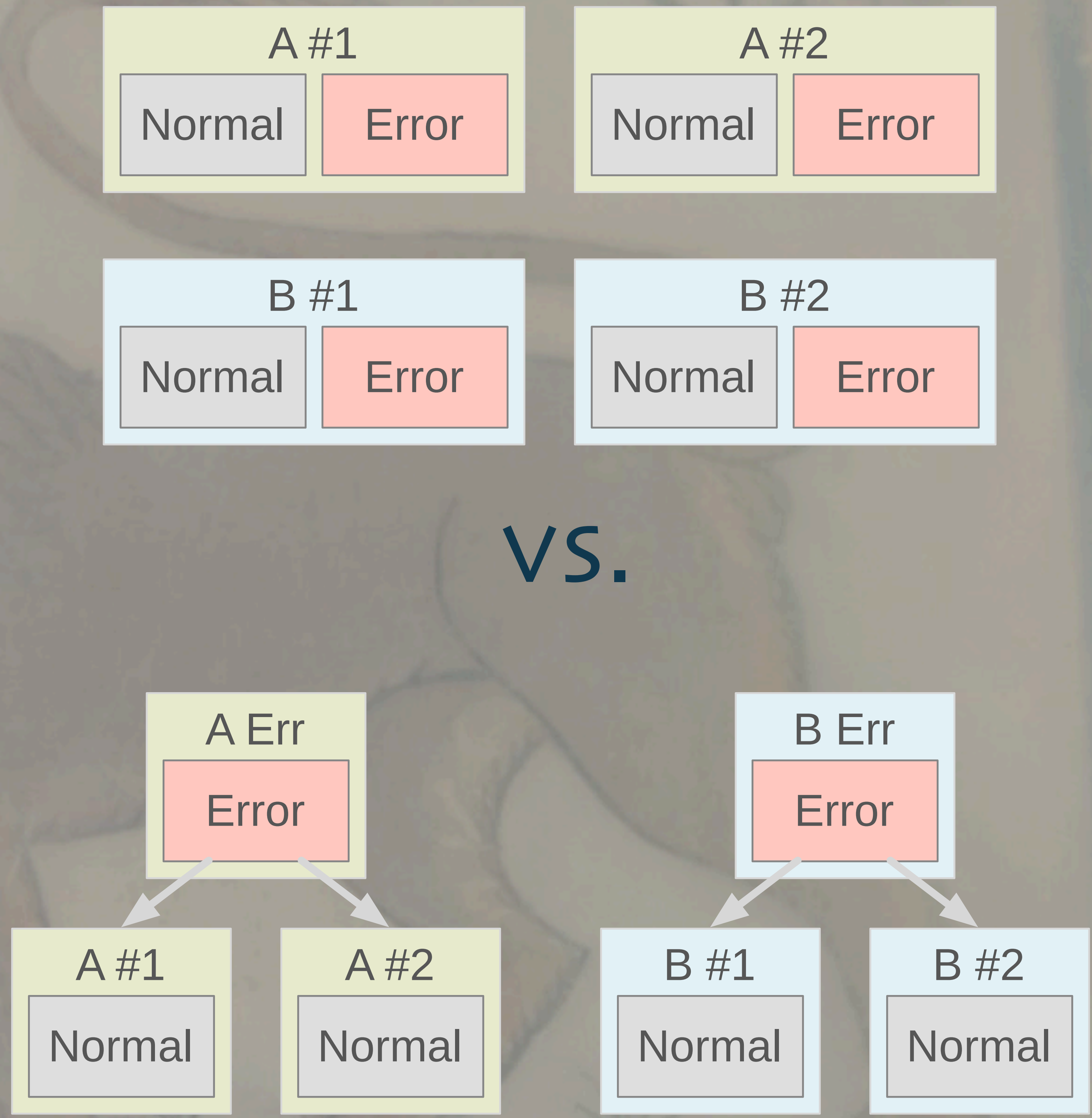
HYSTRIX

DEFEND YOUR APP

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:






Wednesday, March 18, 15

Removed duplicated error-handling logic also makes the normal logic processes smaller, so you can run more of them, etc.

# Conclusions

# 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?



Wednesday, March 18, 15

I would include futures in the list of derivatives.

# CSP, Rx, etc.

+ Emphasize data flows.
+ Typed channels.

Optimal replacement for multithreaded (intra-process) programming.


Wednesday, March 18, 15

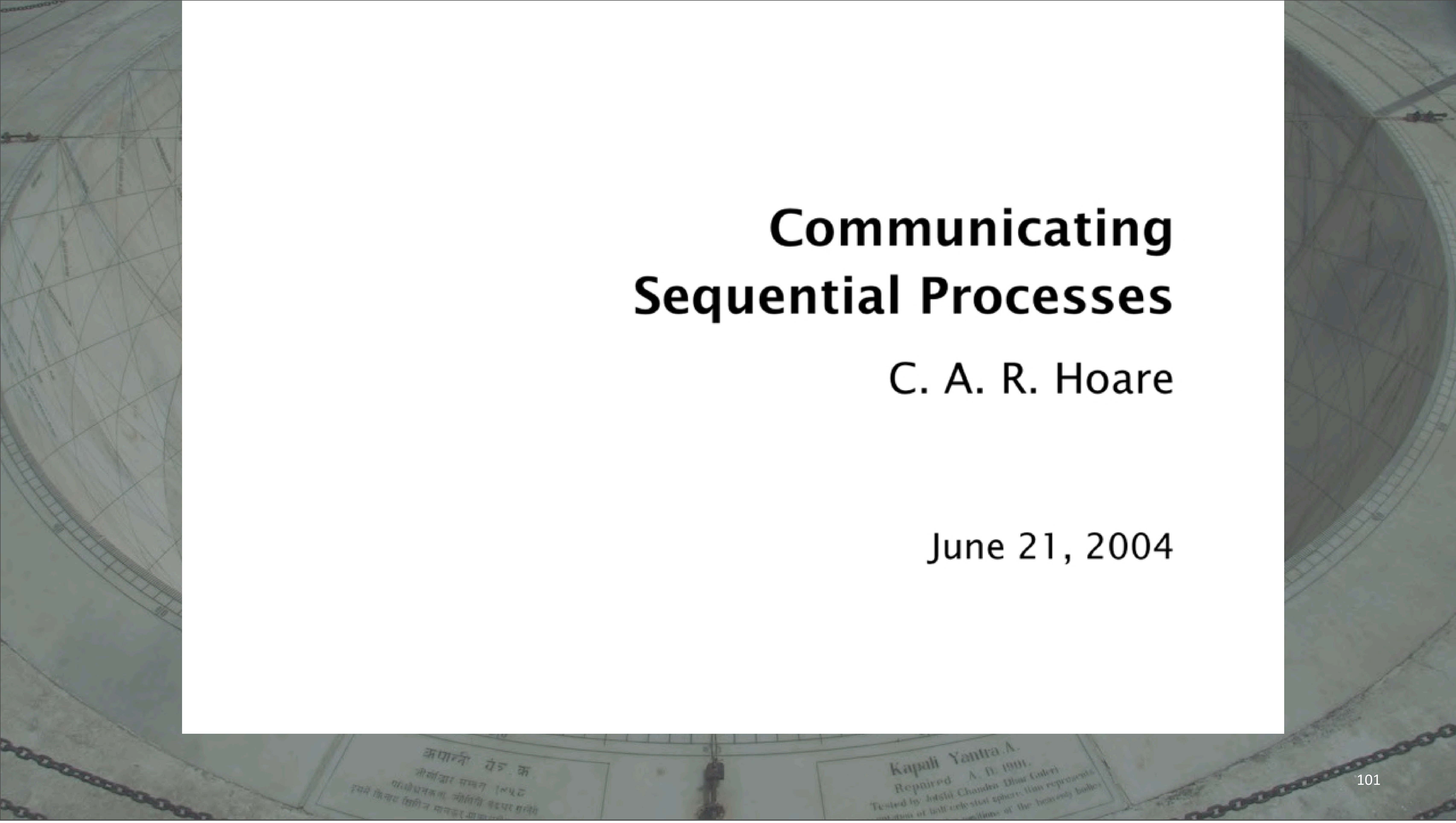
Photos from Jantar Mantar (“instrument”, “calculation”), the astronomical observatory built in Jaipur, India, by Sawai Jai Singh, a Rajput King, in the 1720s-30s. He built four others around India. This is the largest and best preserved.

# Bonus Slides




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.




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.







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

# Prefix

$$a \longrightarrow P$$

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





A process communicates

# Deterministic Choice

$$a \longrightarrow P \;\Box\; b \longrightarrow Q$$

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

# Nondeterministic Choice

$$a \longrightarrow P \sqcap b \longrightarrow Q$$

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

# Interleaving

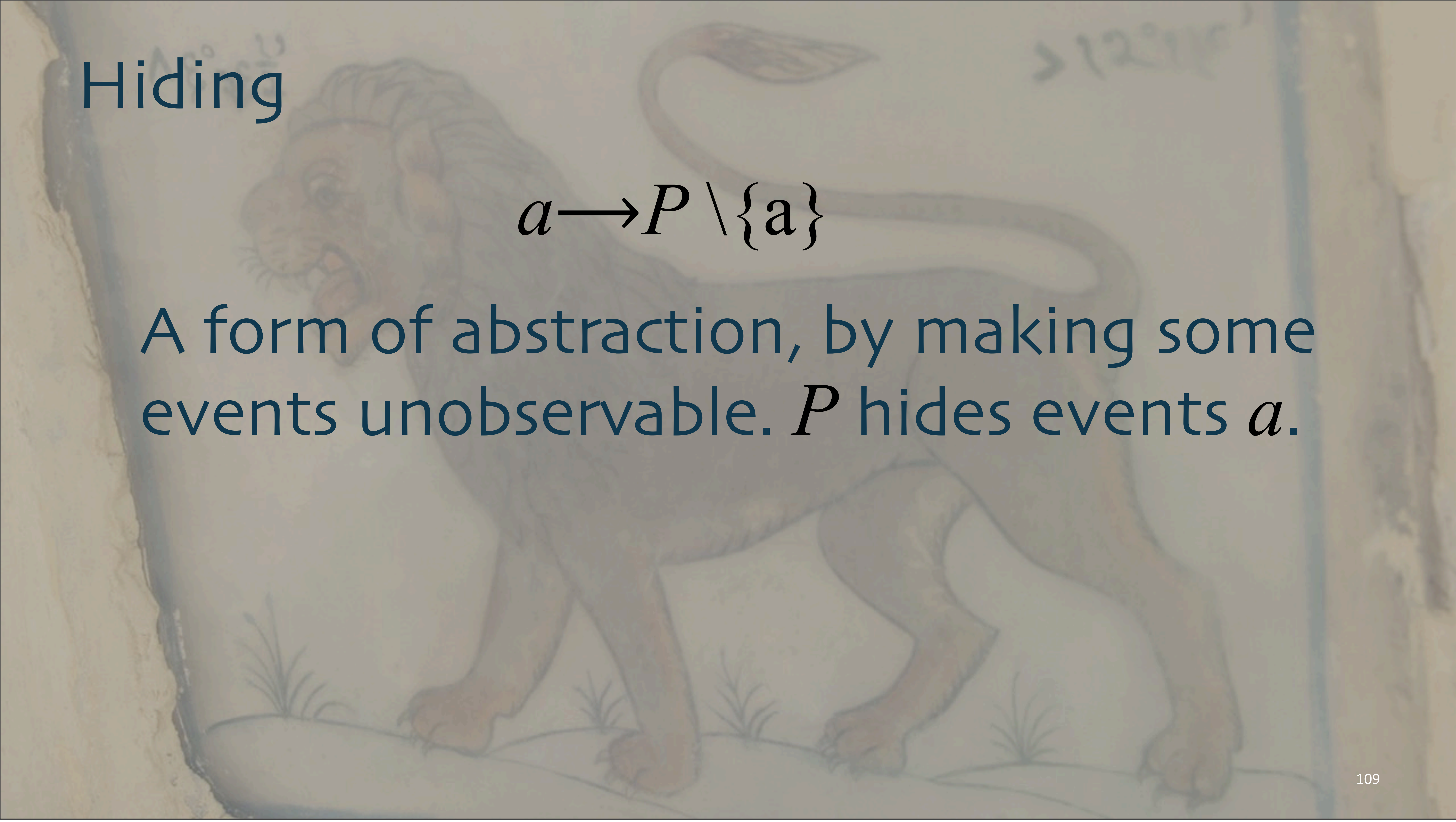
$$P \;|||\; Q$$

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

# Interface Parallel

$$P \,|[\{a\}]|\, Q$$

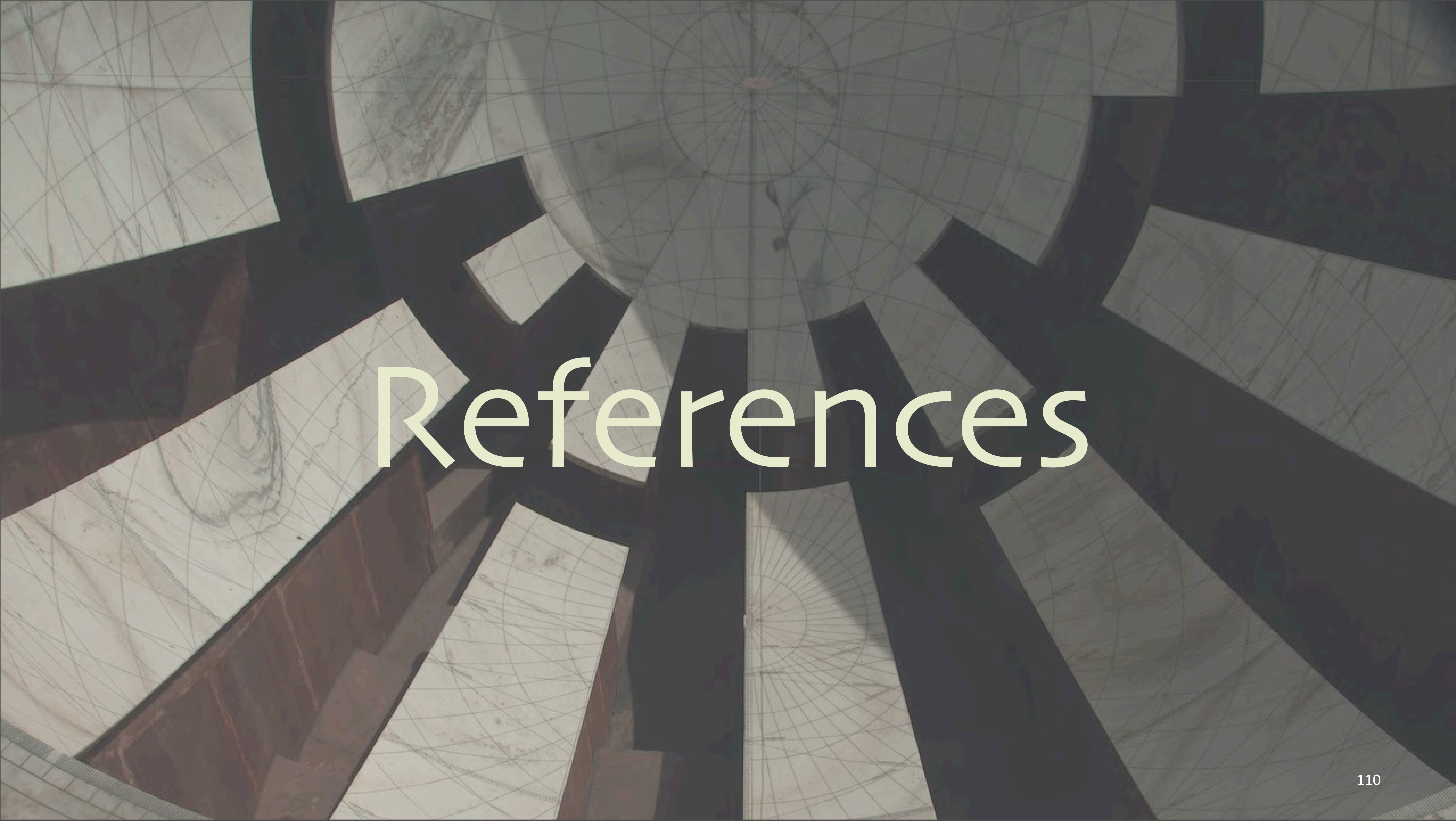
Represents synchronization on event $a$ between $P$ and $Q$.

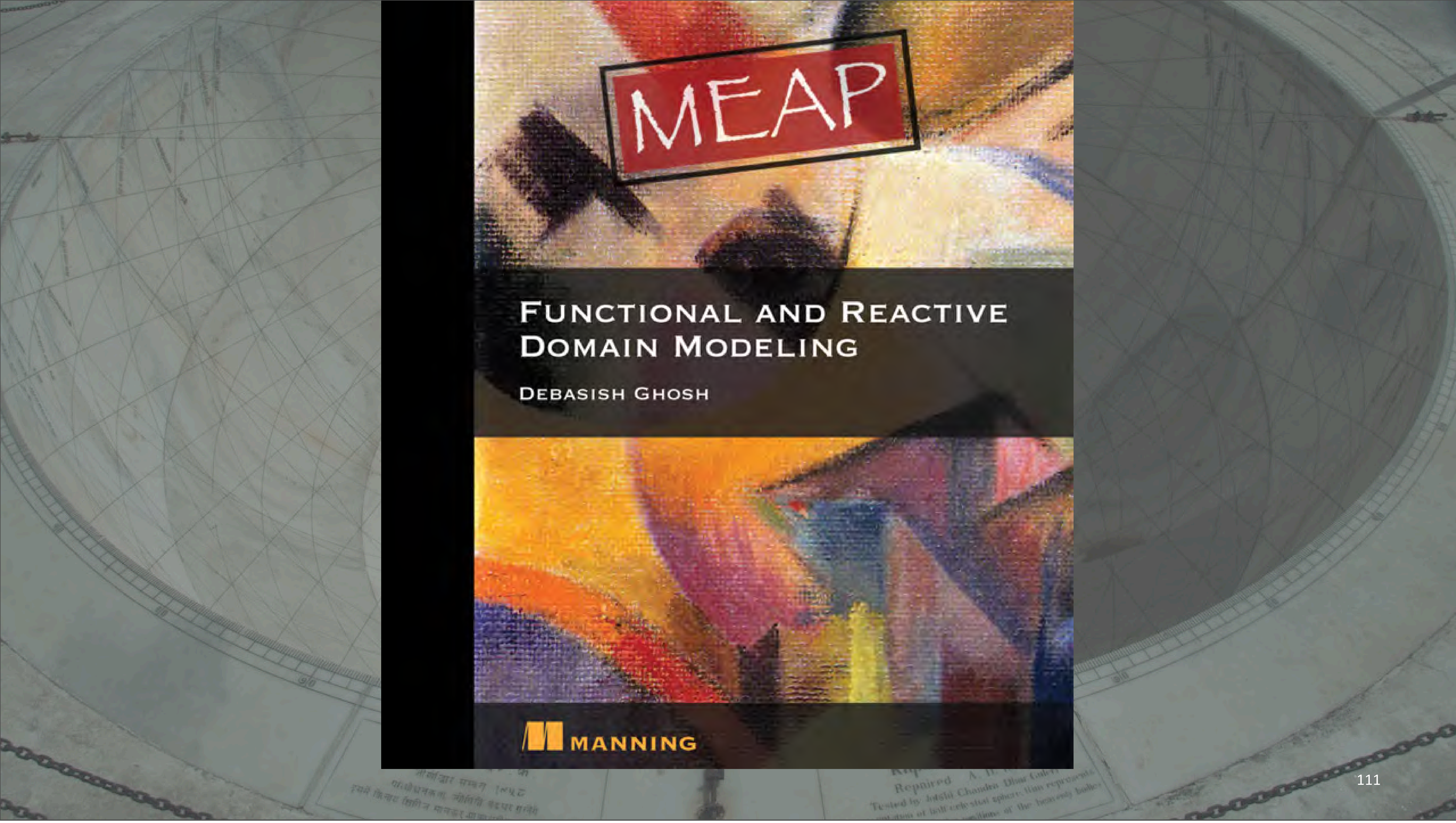


# Hiding

$$a \longrightarrow P \setminus \{\mathrm{a}\}$$

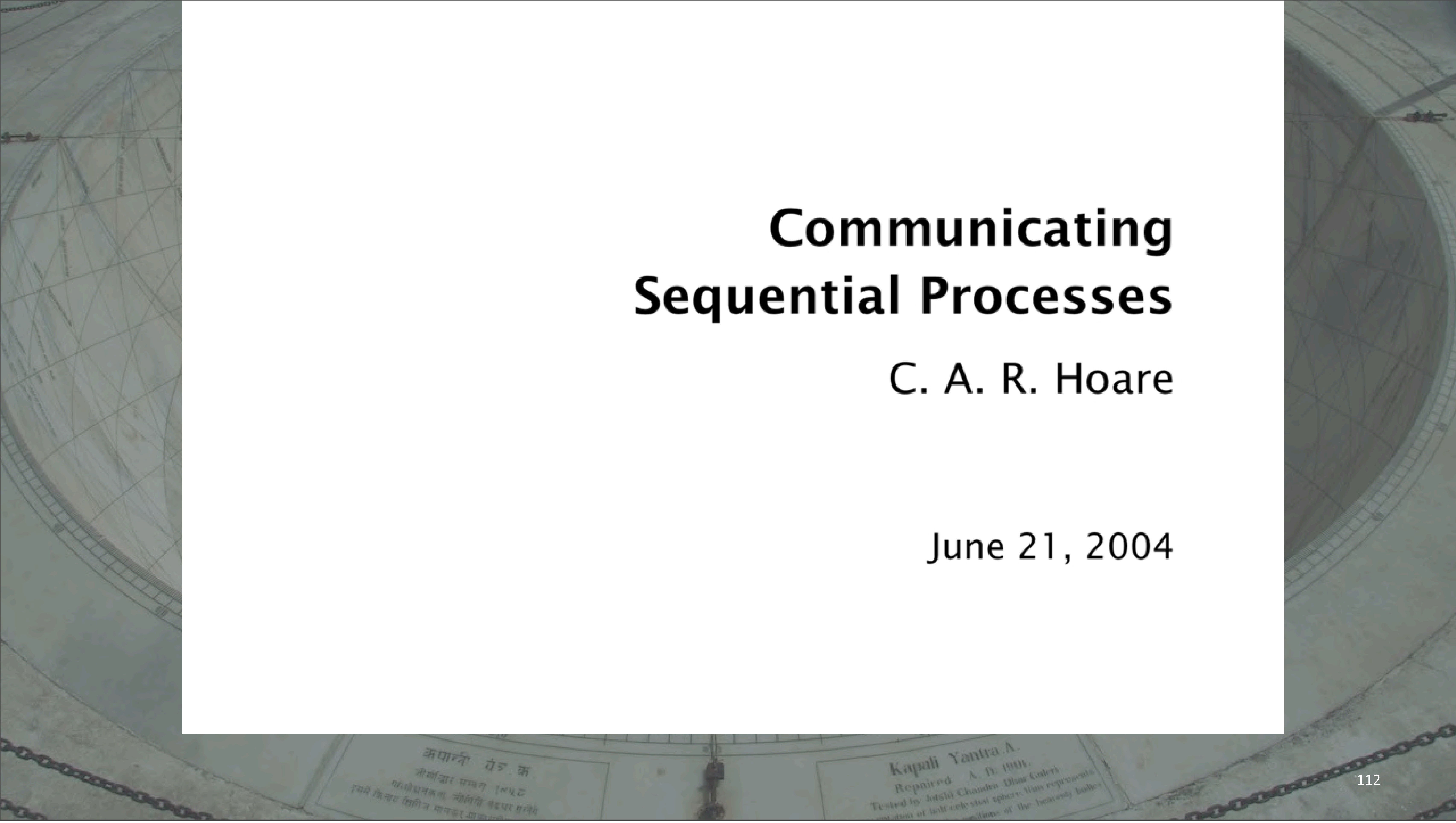
A form of abstraction, by making some events unobservable. $P$ hides events $a$.


References
110






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






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.

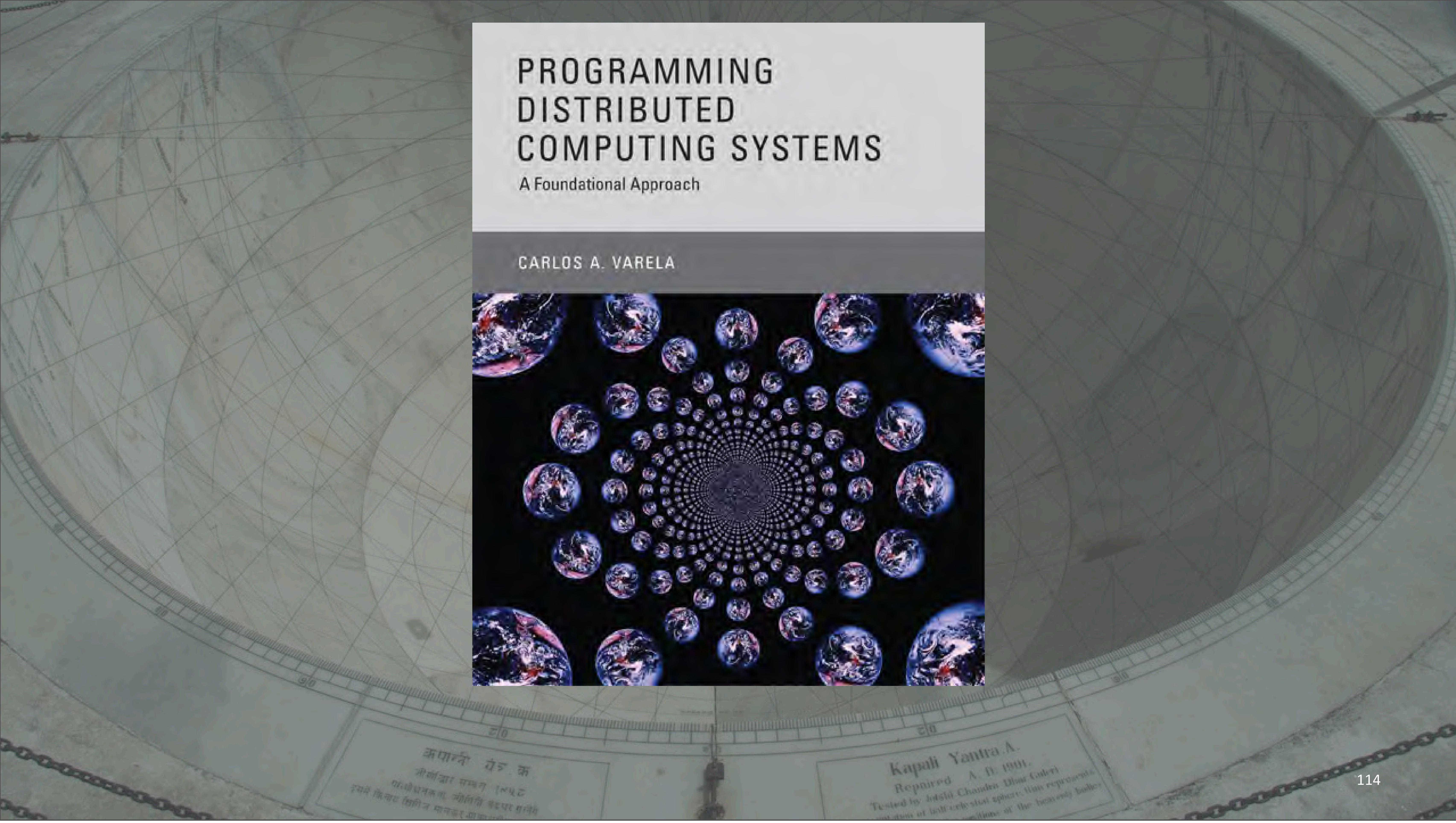
The original version is in print in April 2005 with Prentice-Hall (Pearson).







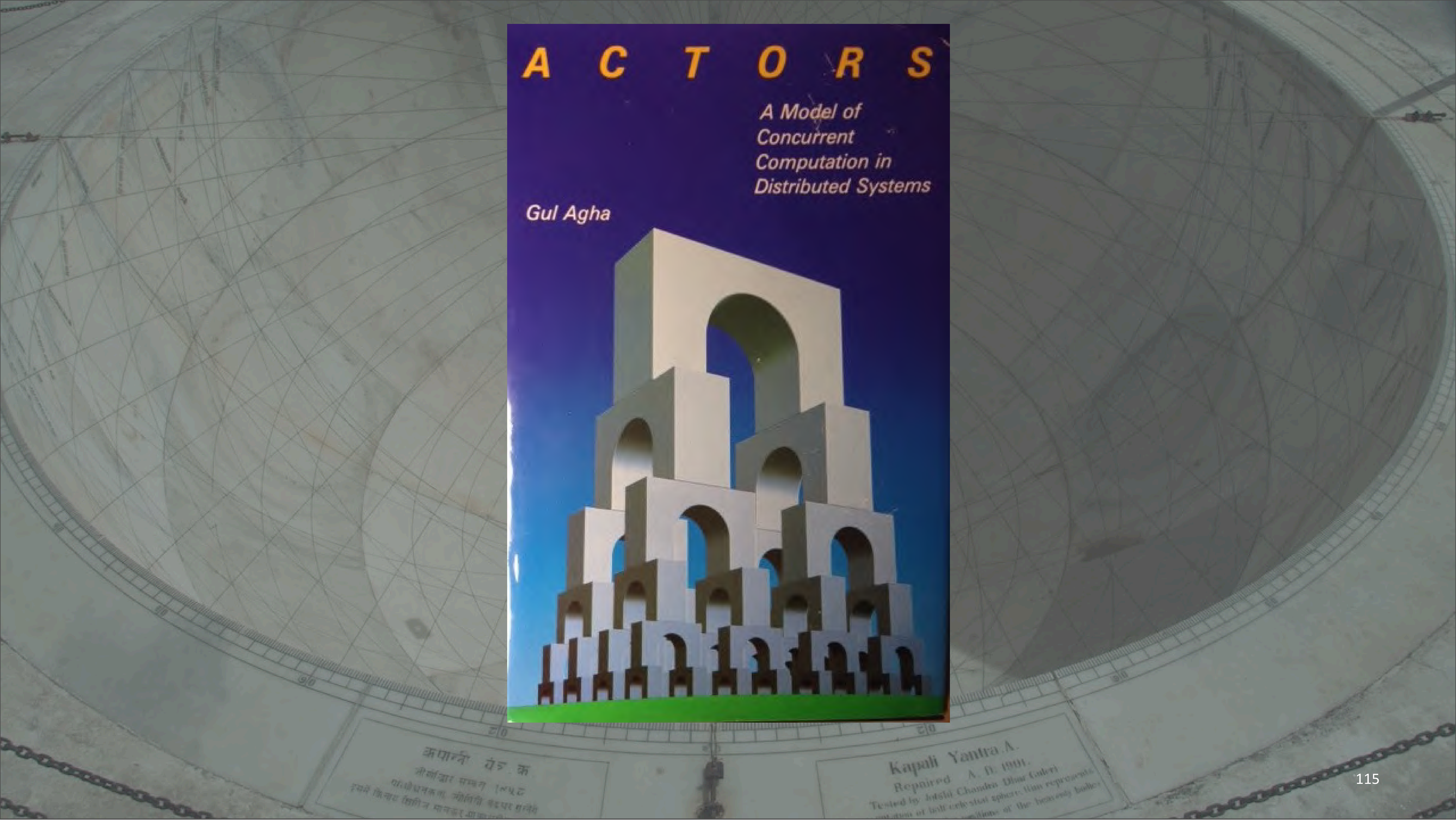
Modern treatment of CSP. Roscoe helped transform the original CSP language into its more rigorous, process algebra form, which was influenced by Milner’s Calculus of Communicating Systems work. This book’s PDF is available free. The treatment is more accessible than Hoare’s book.








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.

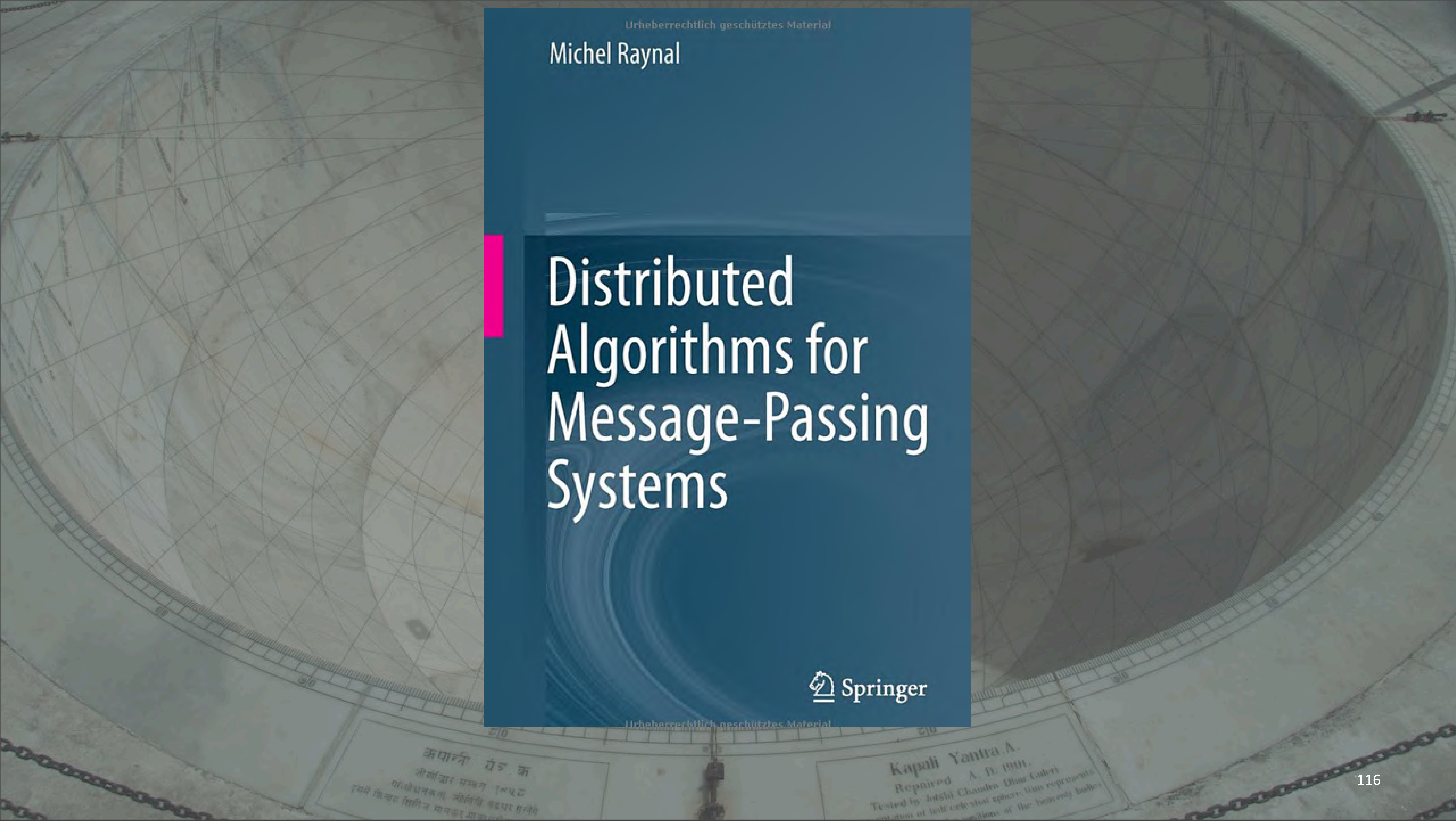





Gul Agha was a grad student at MIT during the 80s and worked on the actor model with Hewitt and others. This book is based on his dissertation.
It doesn't discuss error handling, actor supervision, etc. as these concepts .

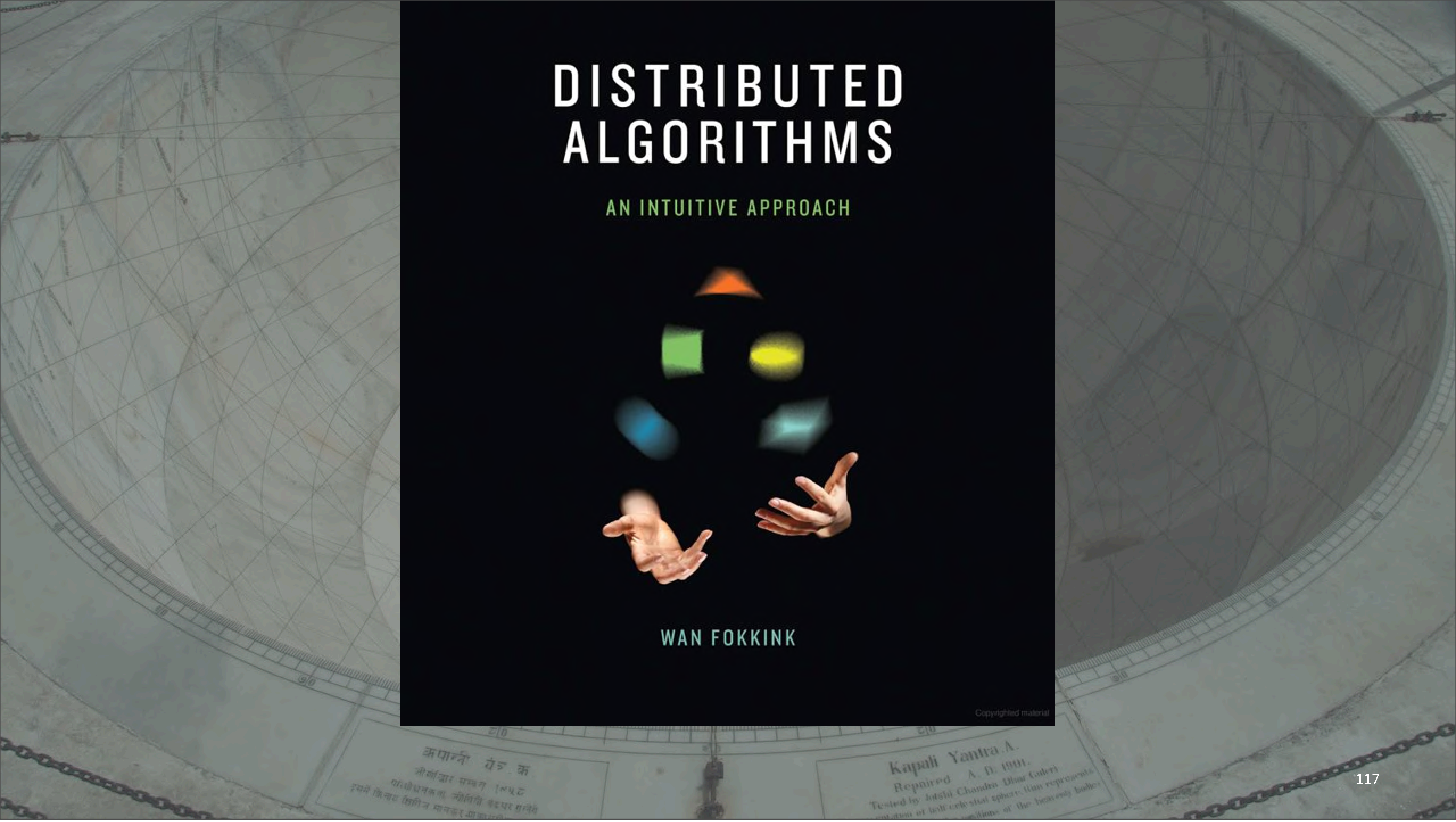
His thesis, http://dspace.mit.edu/handle/1721.1/6952, the basis for his book,http://mitpress.mit.edu/books/actors

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




Wednesday, March 18, 15

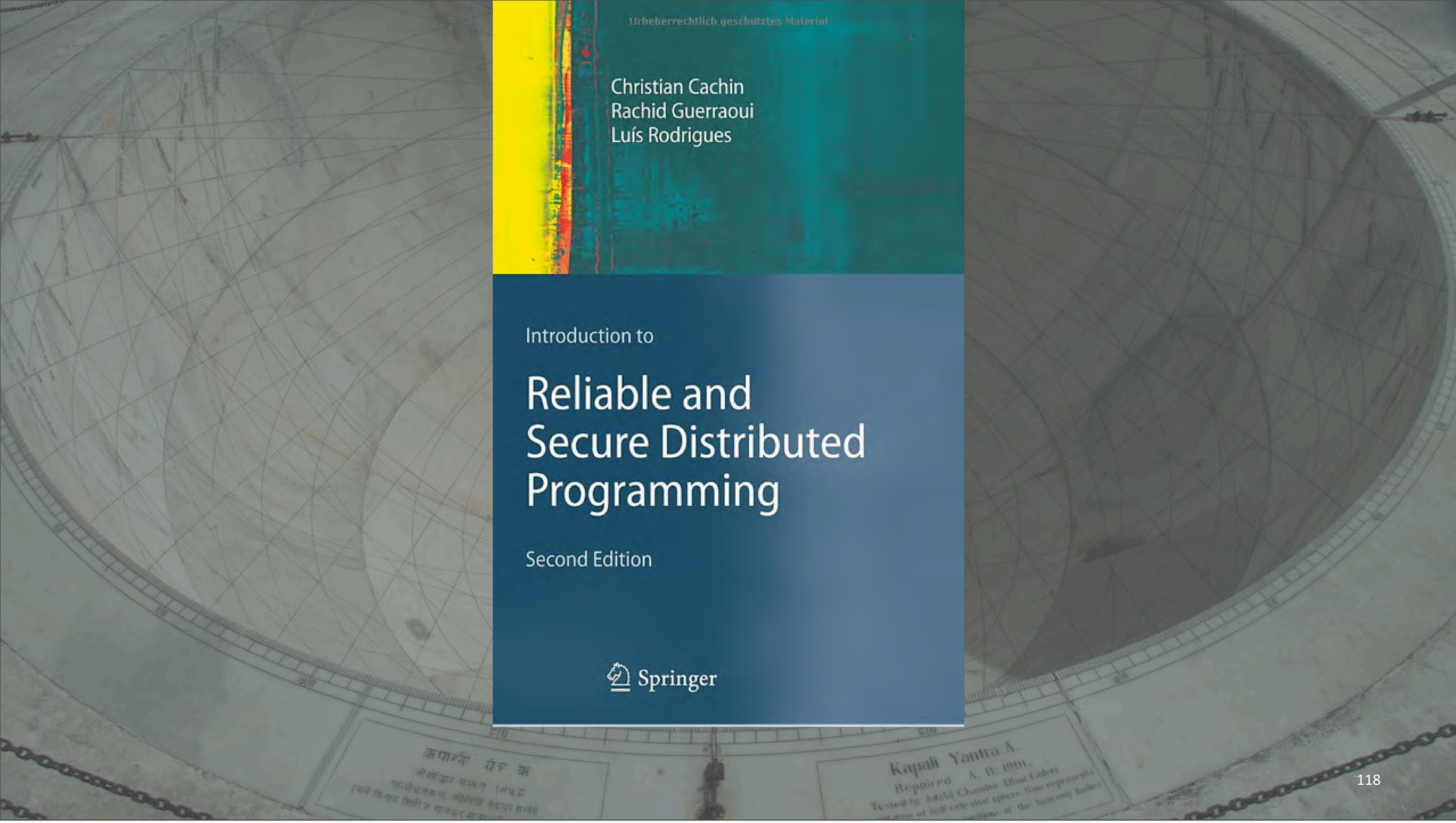
Survey of the classic graph traversal algorithms, algorithms for detecting failures in a cluster, leader election, etc.






Wednesday, March 18, 15

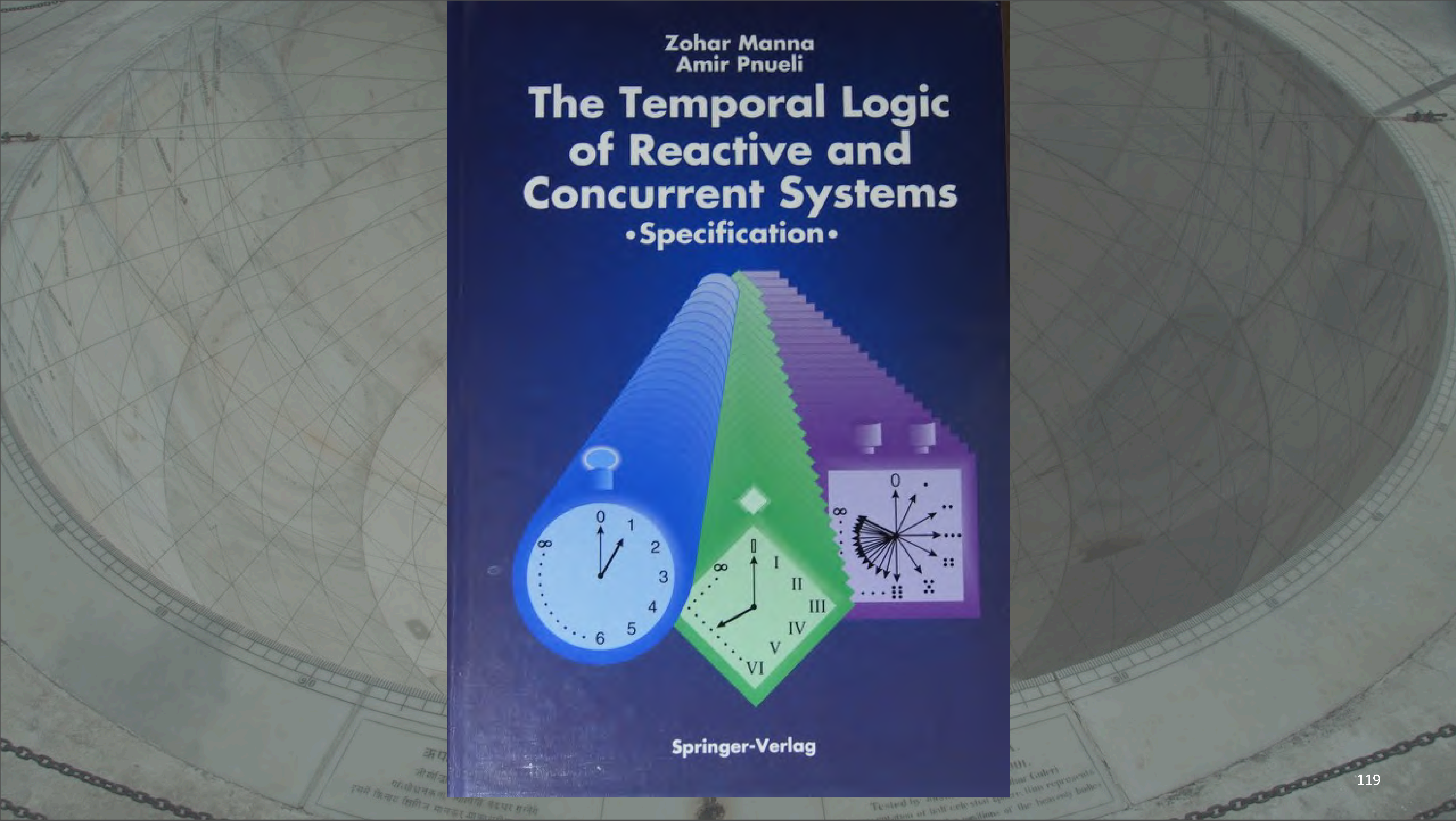
A less comprehensive and formal, but more intuitive approach to fundamental algorithms.







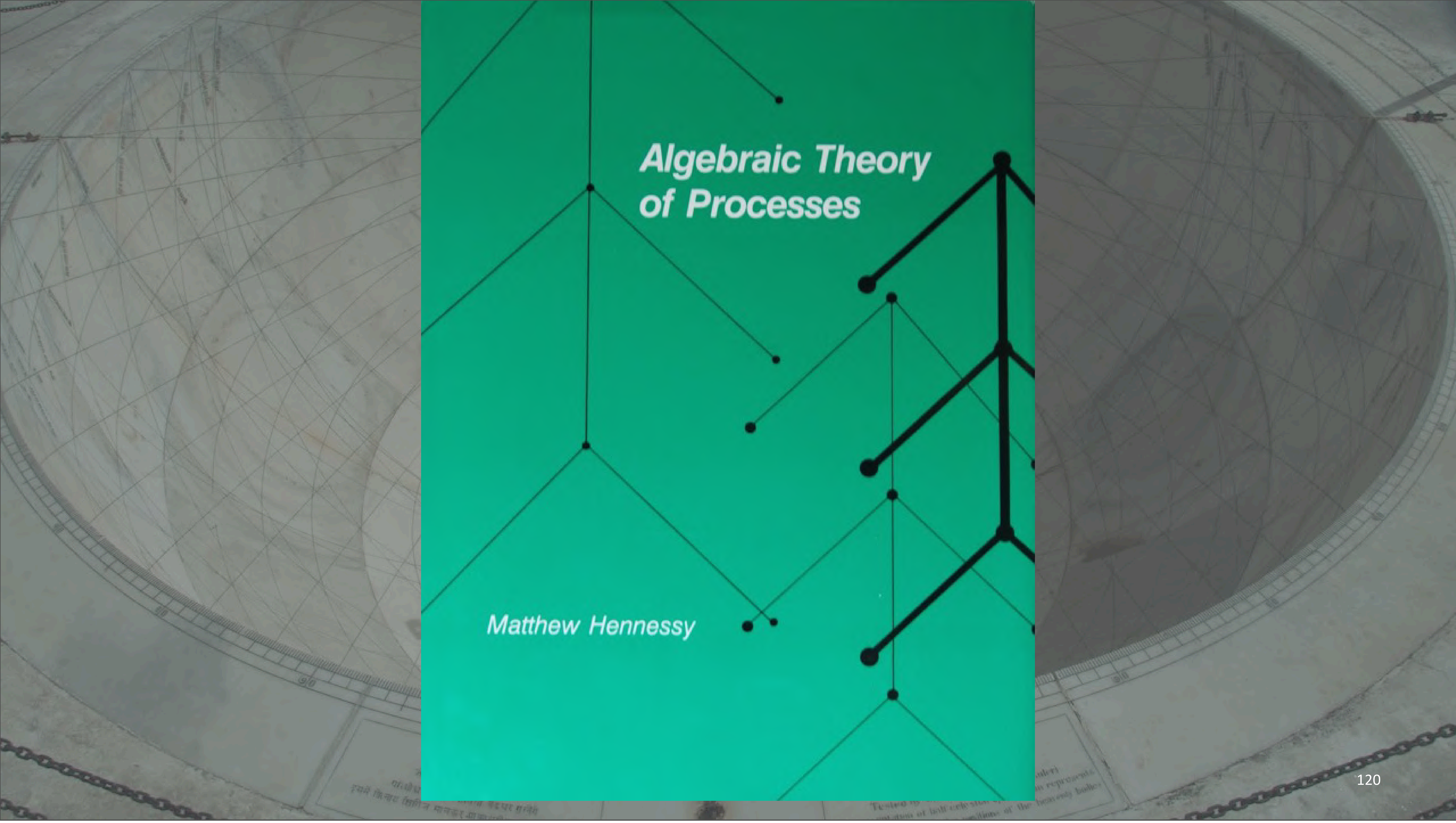

Comprehensive and somewhat formal like Raynal’s book, but more focused on modeling common failures in real systems.







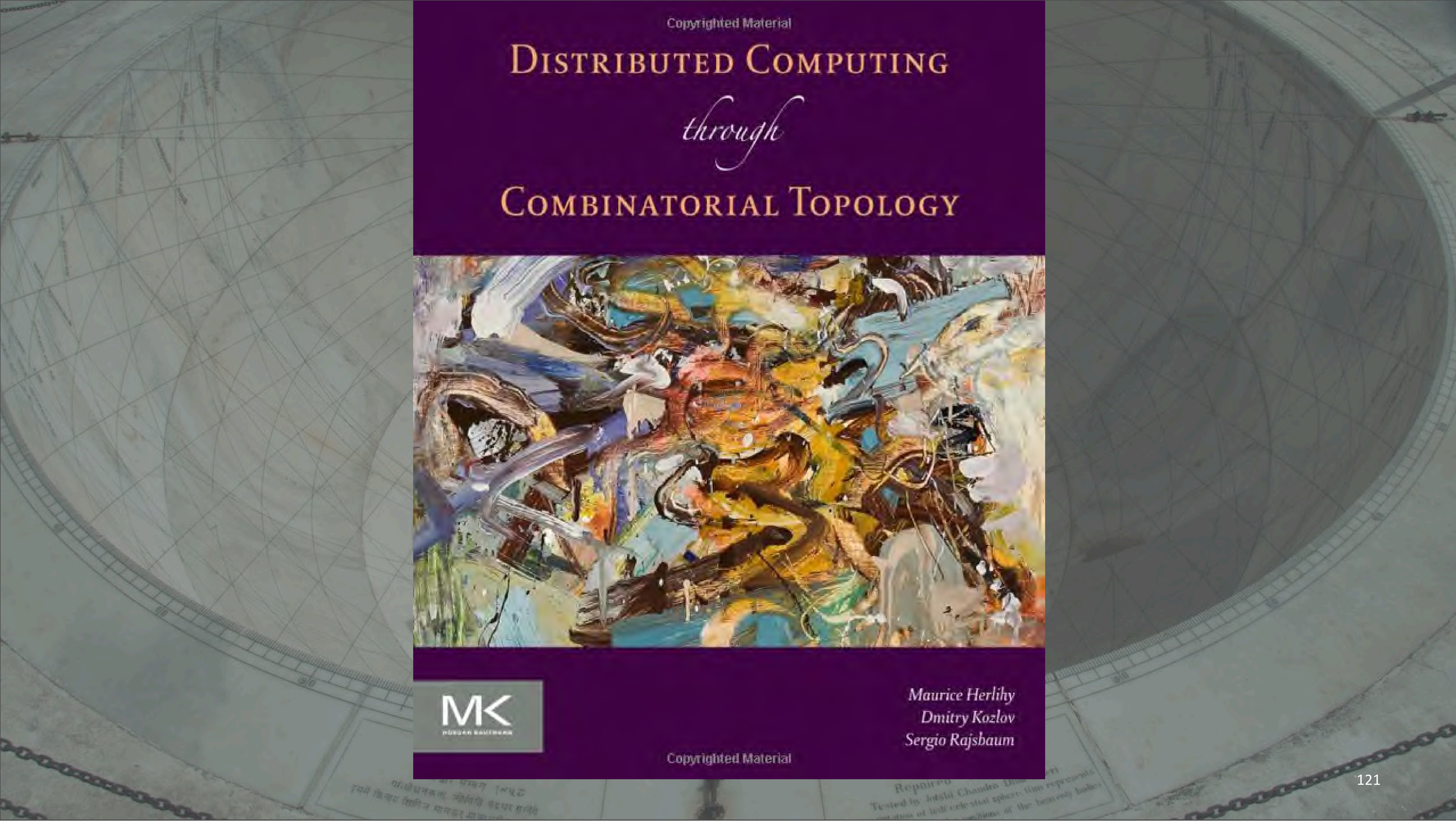

1992: Yes, “Reactive” isn’t new ;) This book is lays out a theoretical model for specifying and proving “reactive” concurrent systems based on temporal logic. While its goal is to prevent logic errors, It doesn’t discuss handling failures from environmental or other external causes in great depth.







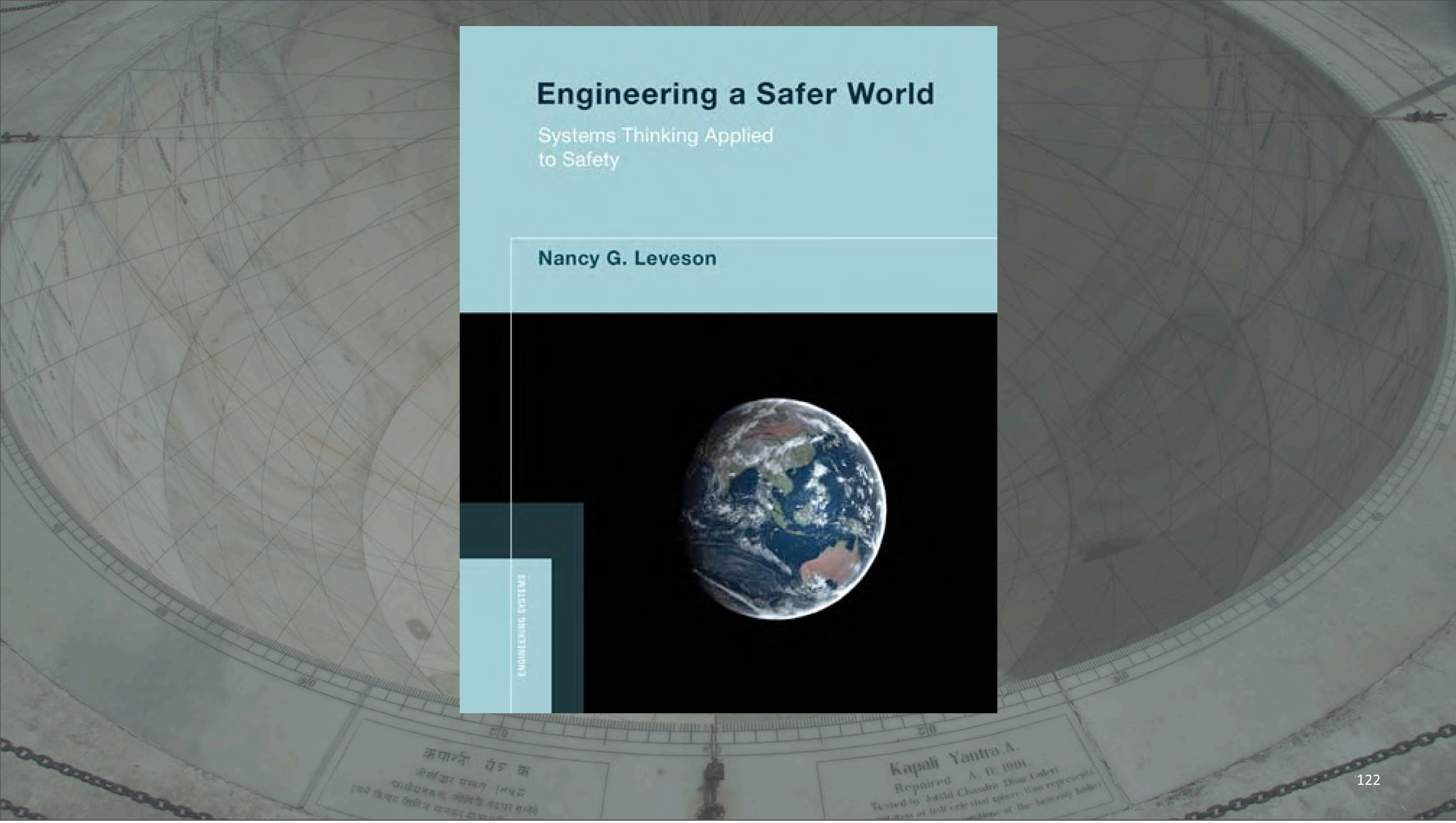

1988: Another treatment of concurrency using algebra. It’s not based on CSP, but it has similar constructs.






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.




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=f6kdp27TYZs&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/