REACT 2014
San Francisco, 18th - 21st November

Error Handling in Reactive Systems

Dean Wampler

Typesafe

©Typesafe 2014 – All Rights Reserved

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 – All Rights Reserved

Typesafe Reactive Big Data

typesafe.com/reactive-big-data

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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

©Typesafe 2014 – All Rights Reserved

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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 – All Rights Reserved

Friday, November 21, 14

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 – All Rights Reserved

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

©Typesafe 2014 – All Rights Reserved

Communicating Sequential Processes

Message passing via channels

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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 – All Rights Reserved

Friday, November 21, 14

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

©Typesafe 2014 – All Rights Reserved

©Typesafe 2014 – All Rights Reserved

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!

©Typesafe 2014 – All Rights Reserved

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!

©Typesafe 2014 – All Rights Reserved

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!

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

©Typesafe 2014 – All Rights Reserved

What should you do when it's full?

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

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 – All Rights Reserved

Friday, November 21, 14

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 – All Rights Reserved

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 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

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

Deadlock are possible unless timeouts are used.

©Typesafe 2014 – All Rights Reserved

Reactive Extensions

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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

©Typesafe 2014 – All Rights Reserved

Uses classic exception handling patterns, with extensions.

©Typesafe 2014 – All Rights Reserved

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 – All Rights Reserved

Adapted from http://www.introtorx.com/content/v1.0.10621.0/11_AdvancedErrorHandling.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 ©Typesafe 2014 – All Rights Reserved

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

There is also a Finally method. Analogous to

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

clauses.

©Typesafe 2014 – All Rights Reserved

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

©Typesafe 2014 – All Rights Reserved

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();
}
```


©Typesafe 2014 – All Rights Reserved

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

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

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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 – All Rights Reserved

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

Reactive Streams

©Typesafe 2014 – All Rights Reserved

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

Reactive Streams

©Typesafe 2014 – All Rights Reserved

Reactive Streams

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

©Typesafe 2014 – All Rights Reserved

#3
Leverage types to prevent errors.

©Typesafe 2014 – All Rights Reserved

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

Express what's really happening with types.

©Typesafe 2014 – All Rights Reserved

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 – All Rights Reserved

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 – All Rights Reserved

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 ©Typesafe 2014 – All Rights Reserved

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 ©Typesafe 2014 – All Rights Reserved

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

Use types to enforce correctness.

©Typesafe 2014 – All Rights Reserved

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

Functional Reactive Programming

©Typesafe 2014 – All Rights Reserved

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 – All Rights Reserved

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 – All Rights Reserved

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 – All Rights Reserved

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

Actor Model

©Typesafe 2014 – All Rights Reserved

Superficially similar to channels.

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

Messages are:

- Handled asynchronously.
- Usually untyped.

©Typesafe 2014 – All Rights Reserved

CSP and Actors are dual

©Typesafe 2014 – All Rights Reserved

CSP Processes are anonymous

... while actors have identities.

©Typesafe 2014 – All Rights Reserved

CSP messaging is synchronous

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

©Typesafe 2014 – All Rights Reserved

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

... but CSP and Actors can implement each other

©Typesafe 2014 – All Rights Reserved

An actor mailbox looks a lot like a channel.

©Typesafe 2014 – All Rights Reserved

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".

©Typesafe 2014 – All Rights Reserved

CSP Processes are anonymous

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

©Typesafe 2014 – All Rights Reserved

CSP messaging is synchronous

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

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

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

Erlang and Akka

©Typesafe 2014 – All Rights Reserved

Distributed Actors

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

©Typesafe 2014 – All Rights Reserved

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

Failure-handling in Actor Systems

©Typesafe 2014 – All Rights Reserved

Let it Crash!

©Typesafe 2014 – All Rights Reserved

Erlang introduced supervisors

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

©Typesafe 2014 – All Rights Reserved

Erlang introduced supervisors

Generalizes nicely to distributed actor systems.

©Typesafe 2014 – All Rights Reserved

Typesafe
©Typesafe 2014 – All Rights Reserved

©Typesafe 2014 – All Rights Reserved

©Typesafe 2014 – All Rights Reserved

Advantages

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

©Typesafe 2014 – All Rights Reserved

Criticisms of Actors

©Typesafe 2014 – All Rights Reserved

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 – All Rights Reserved

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 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

Answers

©Typesafe 2014 – All Rights Reserved

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 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

Must send message to get state

- Also an advantage.
- Protocol for coordinating reads and writes.

©Typesafe 2014 – All Rights Reserved

Couples a queue, mutable state, and a process

- Production systems provide as much decoupling as you need.

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

Actors are untyped

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

©Typesafe 2014 – All Rights Reserved

How should we handle failures?

©Typesafe 2014 – All Rights Reserved

A separate system is needed to supervise and manage processes.

©Typesafe 2014 – All Rights Reserved

Use Actors or...

©Typesafe 2014 – All Rights Reserved

https://github.com/Netflix/Hystrix

©Typesafe 2014 – All Rights Reserved

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

©Typesafe 2014 – All Rights Reserved

Conclusions

©Typesafe 2014 – All Rights Reserved

Actors

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

©Typesafe 2014 – All Rights Reserved

Actors

+Industry proven scalability and resiliency.

+Native asynchrony.

Best-in-class strategy for failure handling.

©Typesafe 2014 – All Rights Reserved

CSP, Rx, etc.

- Limited failure handling facilities.
- Distributed channels?

©Typesafe 2014 – All Rights Reserved

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 2014 – All Rights Reserved

Since we're in a comedy club, let me finish by saying...

©Typesafe 2014 – All Rights Reserved

You've been a great audience!

©Typesafe 2014 – All Rights Reserved

I'm here all day!

©Typesafe 2014 – All Rights Reserved

Don't forget to tip your waitress!

©Typesafe 2014 – All Rights Reserved

Try the veal!

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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 – All Rights Reserved

Communicating Sequential Processes

Message passing via channels

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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 to follow.

Communicating Sequential Processes

C. A. R. Hoare

June 21, 2004

©Typesafe 2014 – All Rights Reserved

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 – All Rights Reserved

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 – All Rights Reserved

Prefix

$$a \longrightarrow P$$

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

©Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

Nondeterministic Choice

$$a \longrightarrow P \sqcap b \longrightarrow \mathrm{Q}$$

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

©Typesafe 2014 – All Rights Reserved

Interleaving

$$P \mid\mid\mid Q$$

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

©Typesafe 2014 – All Rights Reserved

Interface Parallel

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

Represents synchronization on event a between P and Q.

©Typesafe 2014 – All Rights Reserved

Hiding

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

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

©Typesafe 2014 – All Rights Reserved

References

©Typesafe 2014 – All Rights Reserved

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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 – All Rights Reserved

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 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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.

©Typesafe 2014 – All Rights Reserved

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

©Typesafe 2014 – All Rights Reserved

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

©Typesafe 2014 – All Rights Reserved

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

©Typesafe 2014 – All Rights Reserved

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

Friday, November 21, 14

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.

Typesafe

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

© Typesafe 2014 – All Rights Reserved

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.

©Typesafe 2014 – All Rights Reserved

Friday, November 21, 14

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/

©Typesafe 2014 – All Rights Reserved