Building Scalable, Highly Concurrent & Fault-Tolerant Systems: Lessons Learned

Jonas Bonér

CTO Typesafe

Twitter: @jboner

Typesafe

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again

I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again

Lessons Learned through...

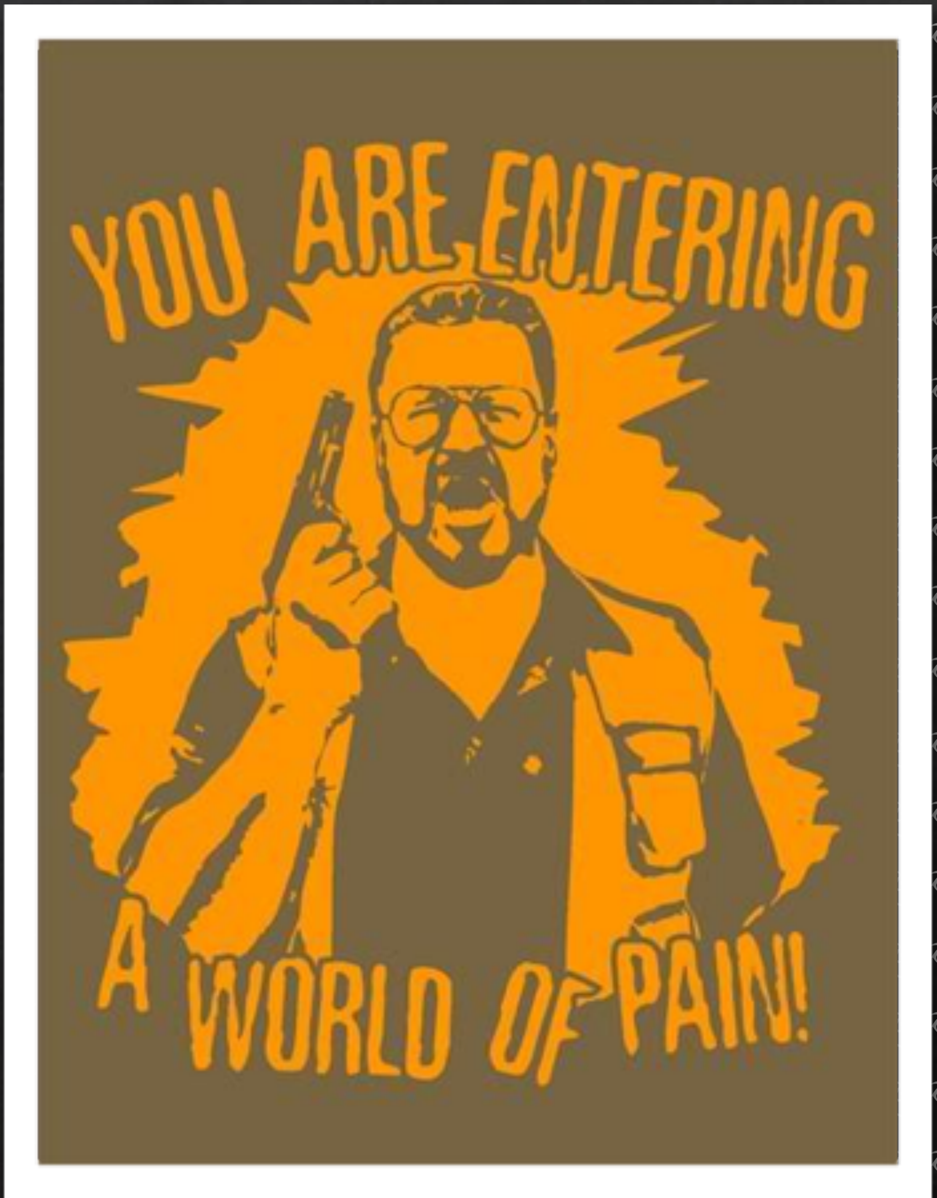
Lessons Learned through... Agony

I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again

I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again

I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again
I will never use distributed transactions again

I will never use distributed transactions again I will never use distributed transactions again
I will never use distributed transactions again I will never use distributed transactions again
I will never use distributed transactions again I will never use distributed transactions again
I will never use distributed transactions again I will never use distributed transactions again
I will never use distributed transactions again I will never use distributed transactions again
I will never use distributed transactions again I will never use distributed transactions again



I will never use distributed transactions again I will never use distributed transactions again
I will never use distributed transactions again I will never use distributed transactions again
I will never use distributed transactions again I will never use distributed transactions again
I will never use distributed transactions again I will never use distributed transactions again
I will never use distributed transactions again I will never use distributed transactions again

Lessons
Learned
through...
Agony
and Pain
lots of
Pain

Agenda

- It's All Trade-offs
- Go Concurrent
- Go Reactive
- Go Fault-Tolerant
- Go Distributed
- Go Big


There is no
Free Lunch.

It's all

Trade-offs

Performance

vs

Scalability

Latency

vs

Throughput

Availability

vs

Consistency

Go Concurrent

Shared mutable state

Shared mutable state

Together with threads...

Shared mutable state

Together with threads...

...leads to

Shared mutable state

Together with threads...

...leads to

...code that is totally INDETERMINISTIC

Shared mutable state

Together with threads...

...leads to

...code that is totally INDETERMINISTIC

...and the root of all EVIL

Shared mutable state

Together with threads...

...leads to

...code that is totally INDETERMINISTIC

...and the root of all EVIL

Please, avoid it at all cost

Typesafe

Shared mutabl e

Together

...leads to

ERMINISTIC

VIL

Pl oid it at all cost

Typesafe

The problem with locks

- Locks do not compose
- Locks breaks encapsulation
- Taking too few locks
- Taking too many locks
- Taking the wrong locks
- Taking locks in the wrong order
- Error recovery is hard

You deserve better tools

- Dataflow Concurrency
- Actors
- Software Transactional Memory (STM)
- Agents

Dataflow Concurrency

- Deterministic
- Declarative
- Data-driven
  - Threads are suspended until data is available
  - Lazy & On-demand
- No difference between:
  - Concurrent code
  - Sequential code
- Examples: Akka & GPars

Typesafe

Actors

- Share NOTHING
- Isolated lightweight event-based processes
- Each actor has a mailbox (message queue)
- Communicates through asynchronous and non-blocking message passing
- Location transparent (distributable)
- Examples: Akka & Erlang

STM

- See the memory as a transactional dataset
- Similar to a DB: *begin*, *commit*, *rollback* (ACI)
- Transactions are retried upon collision
- Rolls back the memory on abort
- Transactions can nest and compose
- Use STM instead of abusing your database with temporary storage of "stratch" data
- Examples: Haskell, Clojure & Scala

Agents

- Reactive memory cells (STM Ref)
- Send a update function to the Agent, which
  1. adds it to an (ordered) queue, to be
  2. applied to the Agent asynchronously
- Reads are "free", just dereferences the Ref
- Cooperates with STM
- Examples: Clojure & Akka

If we could start all over...

If we could start all over...

1. Start with a *Deterministic*, *Declarative* & *Immutable* core

If we could start all over...

1. Start with a *Deterministic*, *Declarative* & *Immutable* core
    - Logic & Functional Programming

If we could start all over...

1. Start with a *Deterministic*, *Declarative* & *Immutable* core
   - Logic & Functional Programming
   - Dataflow

If we could start all over...

1. Start with a *Deterministic*, *Declarative* & *Immutable* core
    - Logic & Functional Programming
    - Dataflow
2. Add *Indeterminism* selectively - *only where needed*

If we could start all over...

1. Start with a *Deterministic*, *Declarative* & *Immutable* core
    - Logic & Functional Programming
    - Dataflow
2. Add *Indeterminism* selectively - *only where needed*
    - Actor/Agent-based Programming

If we could start all over...

1. Start with a *Deterministic*, *Declarative* & *Immutable* core
    - Logic & Functional Programming
    - Dataflow
2. Add *Indeterminism* selectively - *only where needed*
    - Actor/Agent-based Programming
3. Add *Mutability* selectively - *only where needed*

If we could start all over...

1. Start with a *Deterministic*, *Declarative* & *Immutable* core
   - Logic & Functional Programming
   - Dataflow
2. Add *Indeterminism* selectively - *only where needed*
   - Actor/Agent-based Programming
3. Add *Mutability* selectively - *only where needed*
   - Protected by Transactions (STM)

If we could start all over...

1. Start with a *Deterministic*, *Declarative* & *Immutable* core
    - Logic & Functional Programming
    - Dataflow
2. Add *Indeterminism* selectively - *only where needed*
    - Actor/Agent-based Programming
3. Add *Mutability* selectively - *only where needed*
    - Protected by Transactions (STM)
4. Finally - *only if really needed*

If we could start all over...

1. Start with a *Deterministic*, *Declarative* & *Immutable* core
    - Logic & Functional Programming
    - Dataflow
2. Add *Indeterminism* selectively - *only where needed*
    - Actor/Agent-based Programming
3. Add *Mutability* selectively - *only where needed*
    - Protected by Transactions (STM)
4. Finally - *only if really needed*
    - Add *Monitors (Locks)* and explicit *Threads*

Typesafe

Go Reactive

Never block

- ...unless you *really* have to
- Blocking kills scalability (and performance)
- Never sit on resources you don't use
- Use non-blocking IO
- Be reactive
- How?

Go Async

Design for reactive event-driven systems

1. Use asynchronous message passing
2. Use Iteratee-based IO
3. Use push *not* pull (or poll)

- Examples:
    - Akka or Erlang actors
    - Play's reactive Iteratee IO
    - Node.js or JavaScript Promises
    - Server-Sent Events or WebSockets
    - Scala's Futures library

Go Fault-Tolerant

Failure Recovery in Java/C/C# etc.

Failure Recovery in Java/C/C# etc.

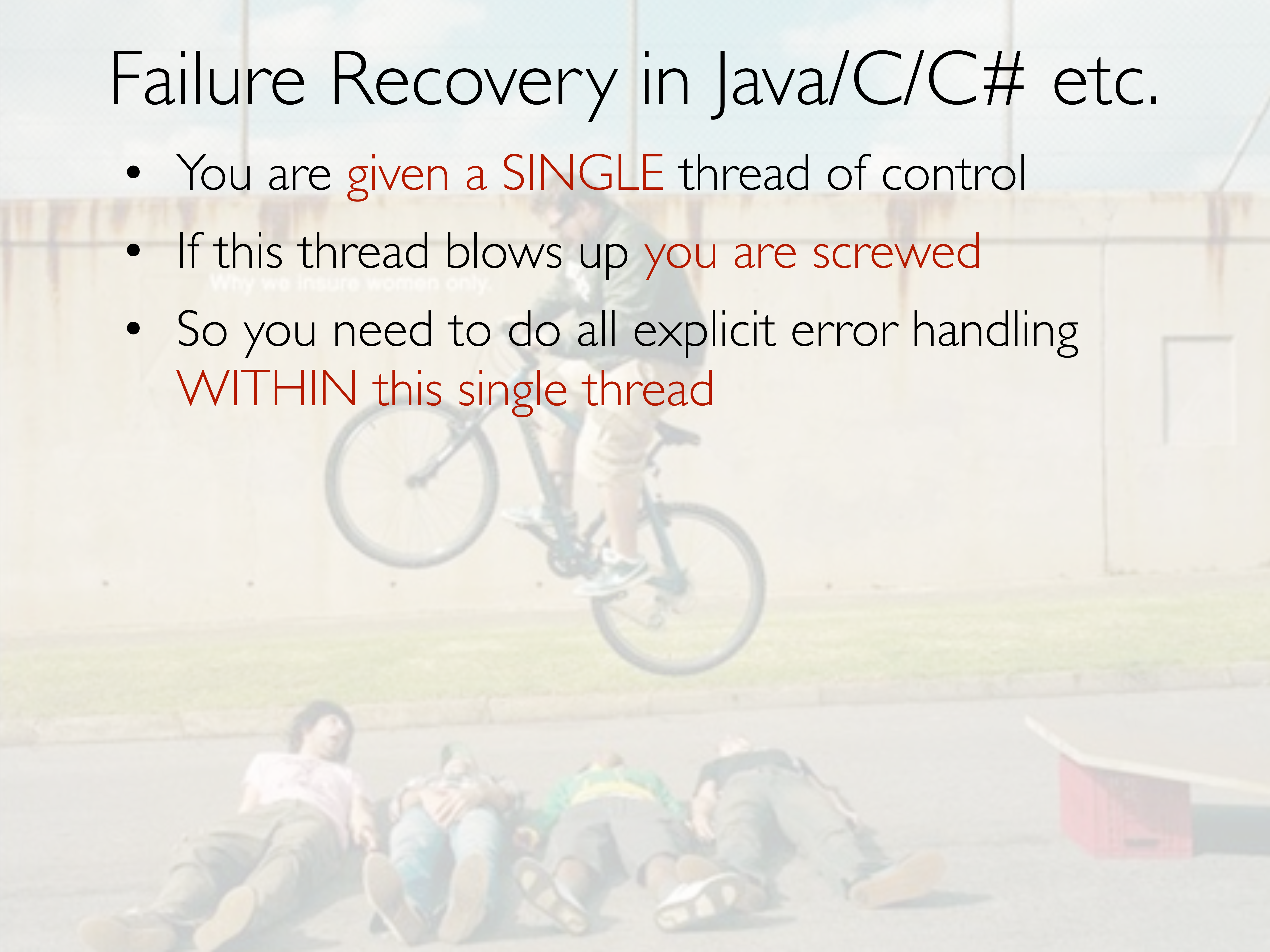
- You are given a SINGLE thread of control

Failure Recovery in Java/C/C# etc.

- You are given a SINGLE thread of control
- If this thread blows up you are screwed

Failure Recovery in Java/C/C# etc.

- You are given a SINGLE thread of control
- If this thread blows up you are screwed
- So you need to do all explicit error handling WITHIN this single thread

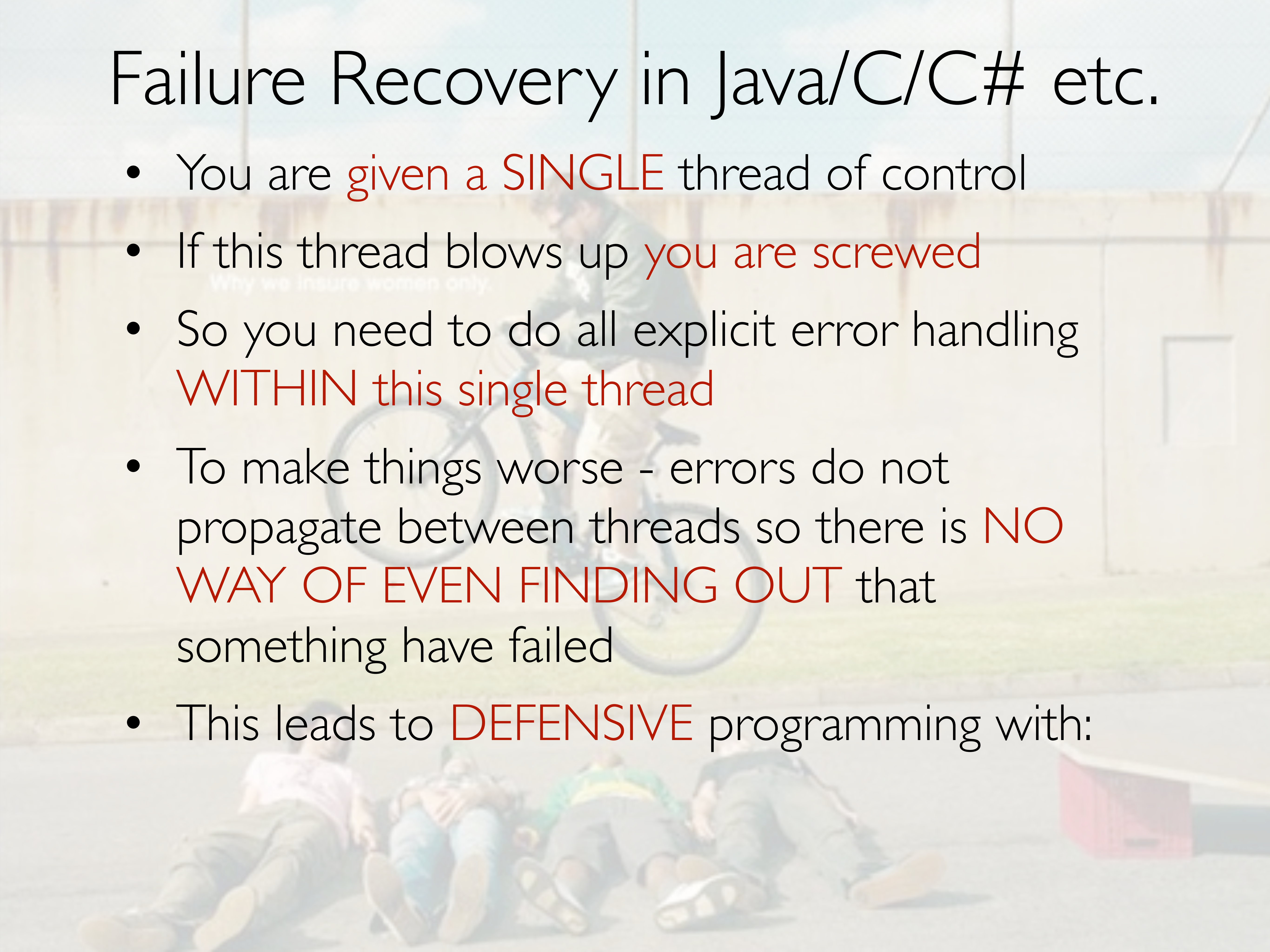


Failure Recovery in Java/C/C# etc.

- You are given a SINGLE thread of control
- If this thread blows up you are screwed
- So you need to do all explicit error handling WITHIN this single thread
- To make things worse - errors do not propagate between threads so there is NO WAY OF EVEN FINDING OUT that something have failed

Failure Recovery in Java/C/C# etc.

- You are given a SINGLE thread of control
- If this thread blows up you are screwed
- So you need to do all explicit error handling WITHIN this single thread
- To make things worse - errors do not propagate between threads so there is NO WAY OF EVEN FINDING OUT that something have failed
- This leads to DEFENSIVE programming with:



Failure Recovery in Java/C/C# etc.

- You are given a SINGLE thread of control
- If this thread blows up you are screwed
- So you need to do all explicit error handling WITHIN this single thread
- To make things worse - errors do not propagate between threads so there is NO WAY OF EVEN FINDING OUT that something have failed
- This leads to DEFENSIVE programming with:
    - Error handling TANGLED with business logic

Failure Recovery in Java/C/C# etc.

- You are given a SINGLE thread of control
- If this thread blows up you are screwed
- So you need to do all explicit error handling WITHIN this single thread
- To make things worse - errors do not propagate between threads so there is NO WAY OF EVEN FINDING OUT that something have failed
- This leads to DEFENSIVE programming with:
    - Error handling TANGLED with business logic
    - SCATTERED all over the code base

Failure Recovery in Jav /C# etc.

- You are given a SINGLE t rol
- If this thread blows
- So you need to WITHIN th
- To ma ot pr here is NO OUT that
- NSIVE programming with:
  - ng TANGLED with business logic
  - SC ERED all over the code base

Just

Let It Crash

The right way

1. Isolated lightweight processes
2. Supervised processes
   - Each running process has a supervising process
   - Errors are sent to the supervisor (asynchronously)
   - Supervisor manages the failure

- Same semantics *local* as *remote*
- For example the Actor Model solves it nicely

Go Distributed

Performance

vs

Scalability

How do I know if I have a performance problem?

How do I know if I have a performance problem?

If your system is
slow for a single user

How do I know if I have a scalability problem?

How do I know if I have a scalability problem?

If your system is
fast for a single user
but slow under heavy load

(Three) Misconceptions about Reliable Distributed Computing
- Werner Vogels

1. Transparency is the ultimate goal
2. Automatic object replication is desirable
3. All replicas are equal and deterministic

Classic paper: *A Note On Distributed Computing* - Waldo et. al.

Fallacy 1

Transparent Distributed Computing

- Emulating Consistency and Shared Memory in a distributed environment
- Distributed Objects
  - *"Sucks like an inverted hurricane"* - Martin Fowler
- Distributed Transactions
  - ...don't get me started...

Fallacy 2
RPC

- Emulating synchronous blocking method dispatch - across the network
- Ignores:
  - Latency
  - Partial failures
  - General scalability concerns, caching etc.
- *"Convenience over Correctness"* - Steve Vinoski

Instead

Instead

Embrace the Network

Use Asynchronous Message Passing

and be done with it

Guaranteed Delivery

Delivery Semantics

- No guarantees
- At most once
- At least once
- Once and only once

It’s all lies.

It’s all lies.

The network is inherently unreliable and there is *no such thing* as 100% *guaranteed delivery*

It's all lies.

Guaranteed Delivery

Guaranteed Delivery

The question is what to guarantee

Guaranteed Delivery

The question is what to guarantee

1. The message is - *sent out on the network?*

Guaranteed Delivery

The question is what to guarantee

1. The message is - *sent out on the network?*
2. The message is - *received by the receiver host's NIC?*

Typesafe

Guaranteed Delivery

The question is what to guarantee

1. The message is - *sent out on the network?*
2. The message is - *received by the receiver host's NIC?*
3. The message is - *put on the receiver's queue?*

Guaranteed Delivery

The question is what to guarantee

1. The message is - *sent out on the network?*
2. The message is - *received by the receiver host's NIC?*
3. The message is - *put on the receiver's queue?*
4. The message is - *applied to the receiver?*

Guaranteed Delivery

The question is what to guarantee

1. The message is - *sent out on the network?*
2. The message is - *received by the receiver host's NIC?*
3. The message is - *put on the receiver's queue?*
4. The message is - *applied to the receiver?*
5. The message is - *starting to be processed by the receiver?*

Guaranteed Delivery

The question is what to guarantee

1. The message is - *sent out on the network?*
2. The message is - *received by the receiver host's NIC?*
3. The message is - *put on the receiver's queue?*
4. The message is - *applied to the receiver?*
5. The message is - *starting to be processed by the receiver?*
6. The message is - *has completed processing by the receiver?*

Typesafe

Ok, then what to do?

1. Start with 0 guarantees (0 additional cost)
2. Add the guarantees you need - one by one

Ok, then what to do?

1. Start with 0 guarantees (0 additional cost)
2. Add the guarantees you need - one by one

Different USE-CASES

→ Different GUARANTEES

→ Different COSTS

Ok, then what to do?

1. Start with 0 guarantees (0 additional cost)
2. Add the guarantees you need - one by one

Different USE-CASES

→ Different GUARANTEES

→ Different COSTS

For each additional guarantee you add you will either:

- decrease performance, throughput or scalability
- increase latency

Typesafe

Just

Just

Use ACKing

Just

Use ACKing

and be done with it

Latency
vs
Throughput

You should strive for

maximal throughput

with

acceptable latency

Go Big

Go Big Data

Big Data

Imperative OO programming *doesn't cut it*

- Object-Mathematics Impedance Mismatch
- We need functional processing, transformations etc.
- Examples: Spark, Crunch/Scrunch, Cascading, Cascalog, Scalding, Scala Parallel Collections
- Hadoop have been called the:
  - *"Assembly language of MapReduce programming"*
  - *"EJB of our time"*

Typesafe

Big Data

Batch processing *doesn't cut it*

- Ala Hadoop
- We need *real-time* data processing
- Examples: Spark, Storm, S4 etc.
- Watch "*Why Big Data Needs To Be Functional*" by Dean Wampler

Go Big

DB

When is a RDBMS **not** good enough?

Typesafe

Scaling reads to a RDBMS is hard

Scaling writes
to a RDBMS
is impossible

Do we

really need

a RDBMS?

Do we

really need

a RDBMS?

Sometimes...

Do we really need a RDBMS?

Typesafe

Do we
really need
a RDBMS?

But many times we don’t

Atomic

Consistent

Isolated

Durable

Typesafe

Availability

vs

Consistency

Typesafe

Brewer's

CAP

theorem

Typesafe

You can only pick 2

- ☐ Consistency
- ☐ Availability
- ☐ Partition tolerance

At a given point in time

Centralized system

- In a centralized system (RDBMS etc.) we don't have network partitions, e.g. P in CAP
- So you get both:

☑ Consistency

☑ Availability

Distributed system

- In a distributed (scalable) system we will have network partitions, e.g. P in CAP
- So you get to only pick one:

  ☐ Consistency

  ☐ Availability

Basically Available

Soft state

Eventually consistent

Think about your data

Then think again

- When do you need ACID?
- When is Eventual Consistency a better fit?
- Different kinds of data has different needs
- You need full consistency less than you think

How fast is fast enough?

- Never guess: Measure, measure and measure
- Start by defining a baseline
  - Where are we now?
- Define what is "good enough" - i.e. SLAs
  - Where do we want to go?
  - When are we done?
- Beware of micro-benchmarks

...or, when can we go for a beer?

- Never guess: Measure, measure and measure
- Start by defining a baseline
  - Where are we now?
- Define what is "good enough" - i.e. SLAs
  - Where do we want to go?
  - When are we done?
- Beware of micro-benchmarks

SO

GO

...now home and build yourself

Scalable,

Highly Concurrent &

Fault-Tolerant

Systems

Thank You

Email: jonas@typesafe.com
Web: typesafe.com
Twitter: @jboner

Typesafe