Scaling software with akka

Jonas Bonér

CTO

@jboner

Typesafe

Scaling software with akka

Selection of Akka Production Users

TDC
Telefónica
CSC
htc quietly brilliant
hp
BLIZZARD ENTERTAINMENT

UBS
SIEMENS
amazon.com
KLOUT

W3C®
HSBC
CISCO
HUAWEI
JUNIPer NETWORKS

svt
Autodesk
CREDIT SUISSE
IGN

AtoS
O2
vmware
dialog Smart Stream Platform
SEVEN Networks

xerox
DRW TRADING GROUP

OOYALA
novus
abiquo
foursquare

T8 Webware
CARTOMAPIC
moshi monsters
ROVIO
BBC
navirec

PRECOG
Maritime Poker
Answers.com The world's leading Q&A site
azavea
banksimple
zeebox The best thing to happen to TV since TV

Typesafe

akka

Manage System Overload

Automatic Replication & Distribution

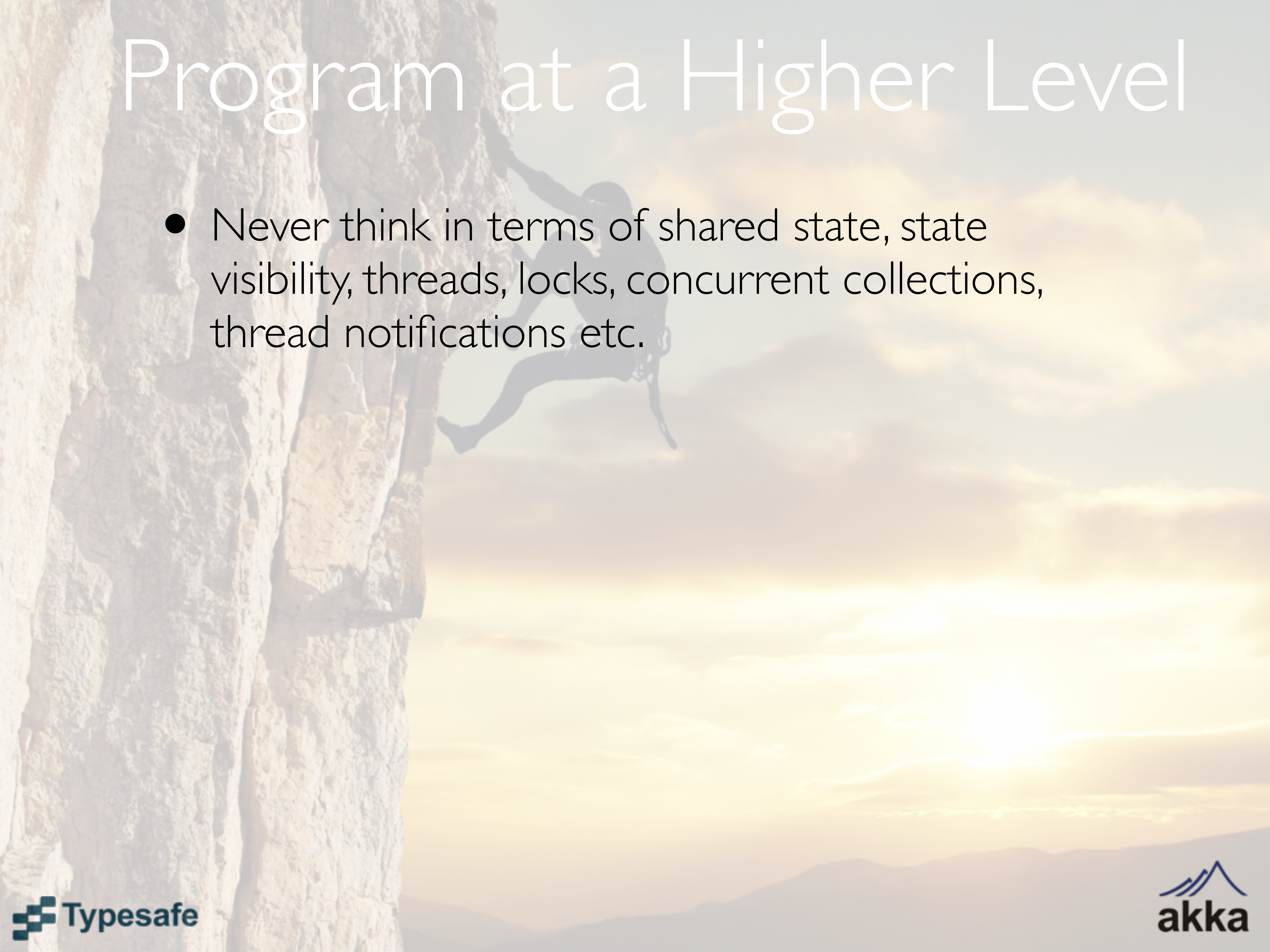
for Fault-tolerance & Scalability

Typesafe

akka

Program at a Higher Level

Program at a Higher Level

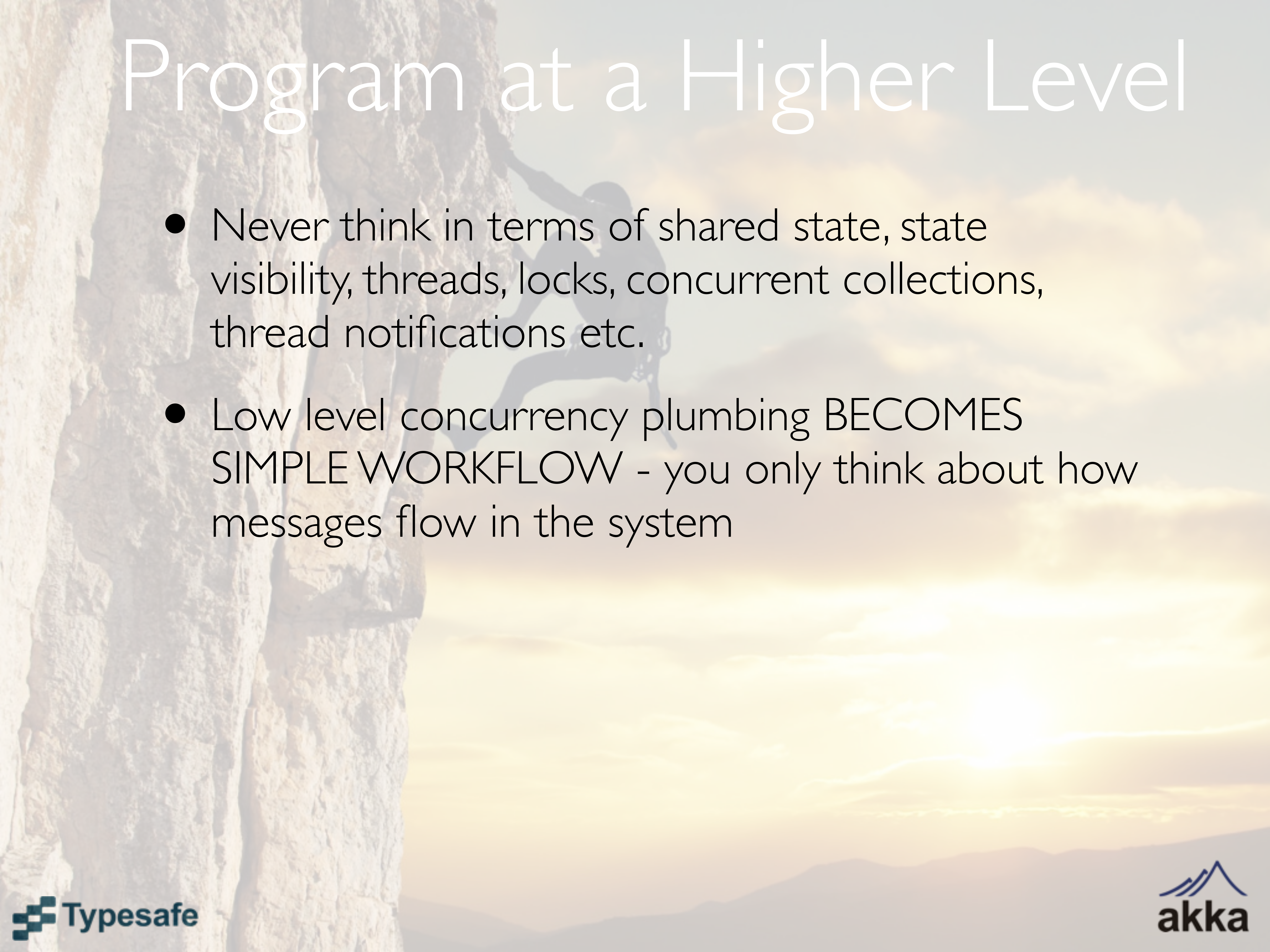


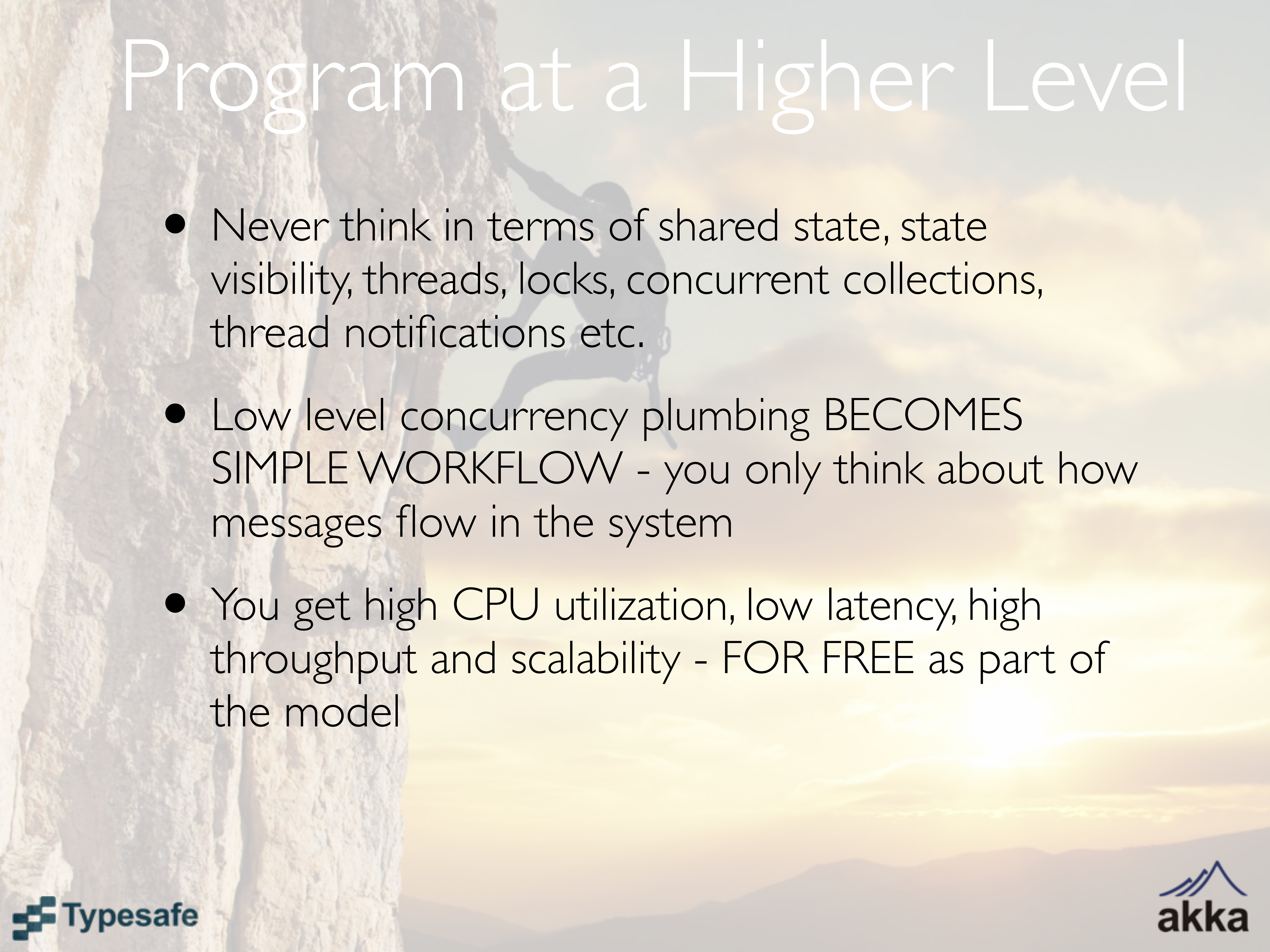
Program at a Higher Level

- Never think in terms of shared state, state visibility, threads, locks, concurrent collections, thread notifications etc.

Program at a Higher Level

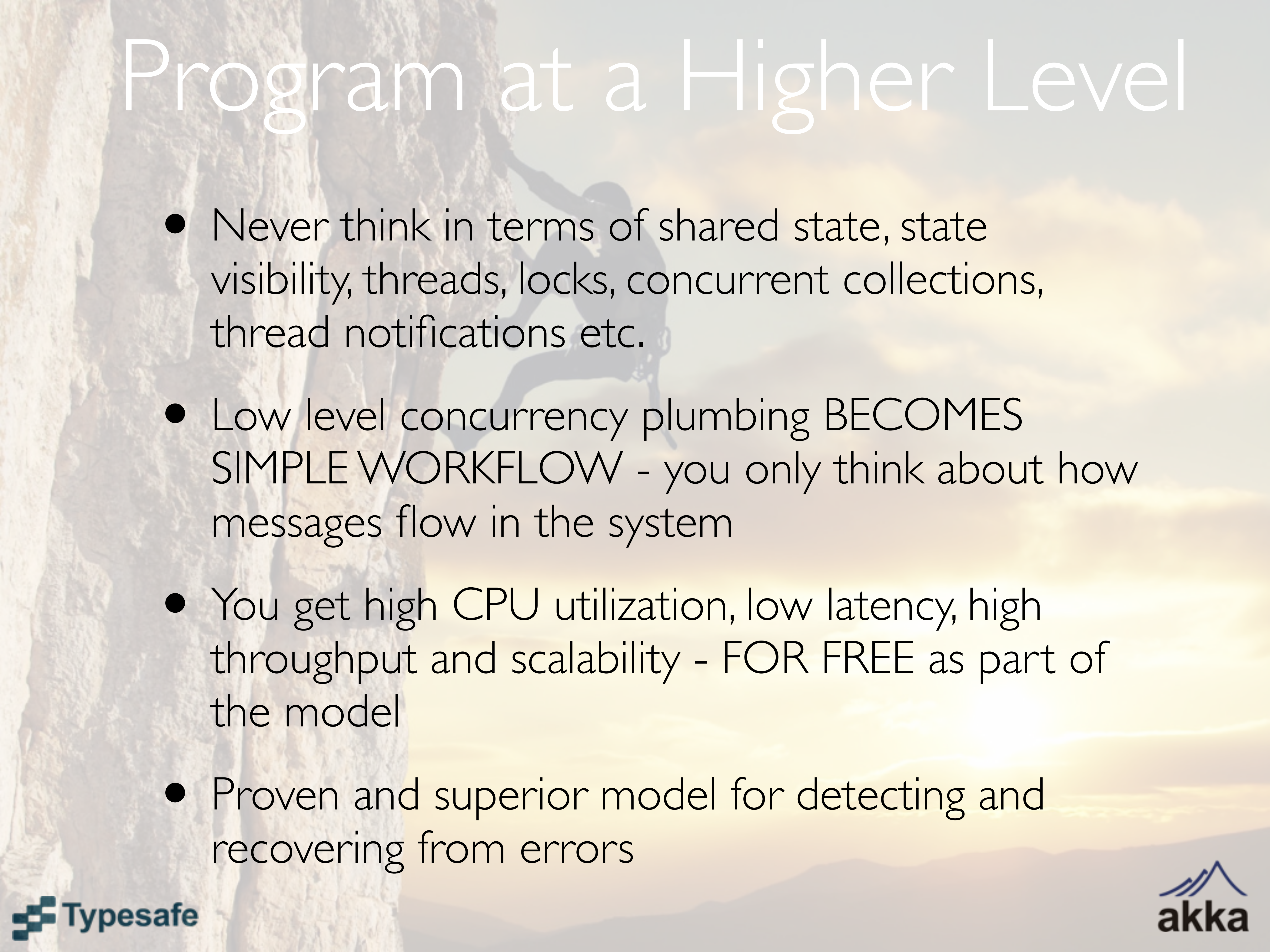
- Never think in terms of shared state, state visibility, threads, locks, concurrent collections, thread notifications etc.
- Low level concurrency plumbing BECOMES SIMPLE WORKFLOW - you only think about how messages flow in the system





Program at a Higher Level

- Never think in terms of shared state, state visibility, threads, locks, concurrent collections, thread notifications etc.
- Low level concurrency plumbing BECOMES SIMPLE WORKFLOW - you only think about how messages flow in the system
- You get high CPU utilization, low latency, high throughput and scalability - FOR FREE as part of the model

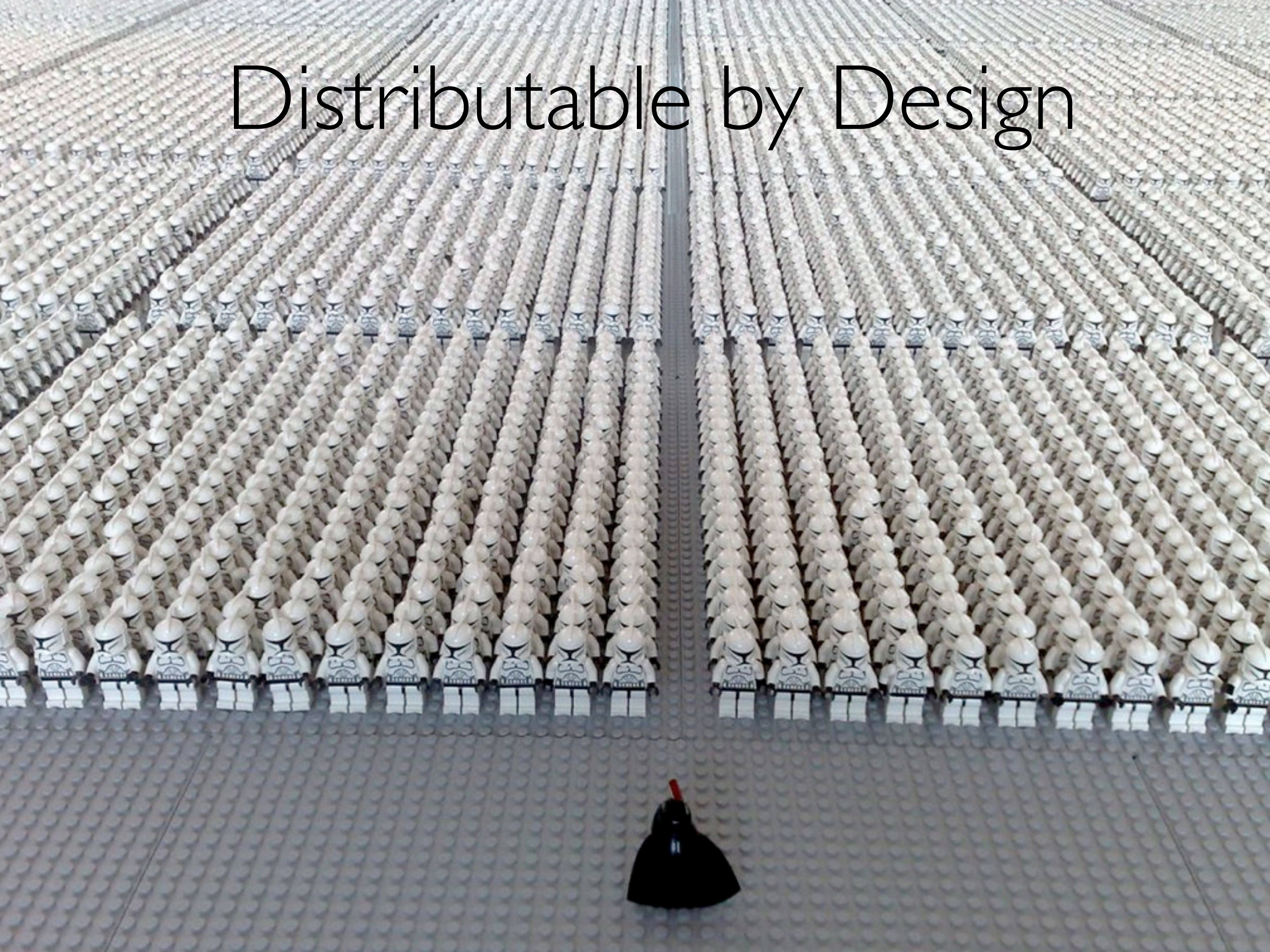


Program at a Higher Level

- Never think in terms of shared state, state visibility, threads, locks, concurrent collections, thread notifications etc.
- Low level concurrency plumbing BECOMES SIMPLE WORKFLOW - you only think about how messages flow in the system
- You get high CPU utilization, low latency, high throughput and scalability - FOR FREE as part of the model
- Proven and superior model for detecting and recovering from errors

Typesafe

akka

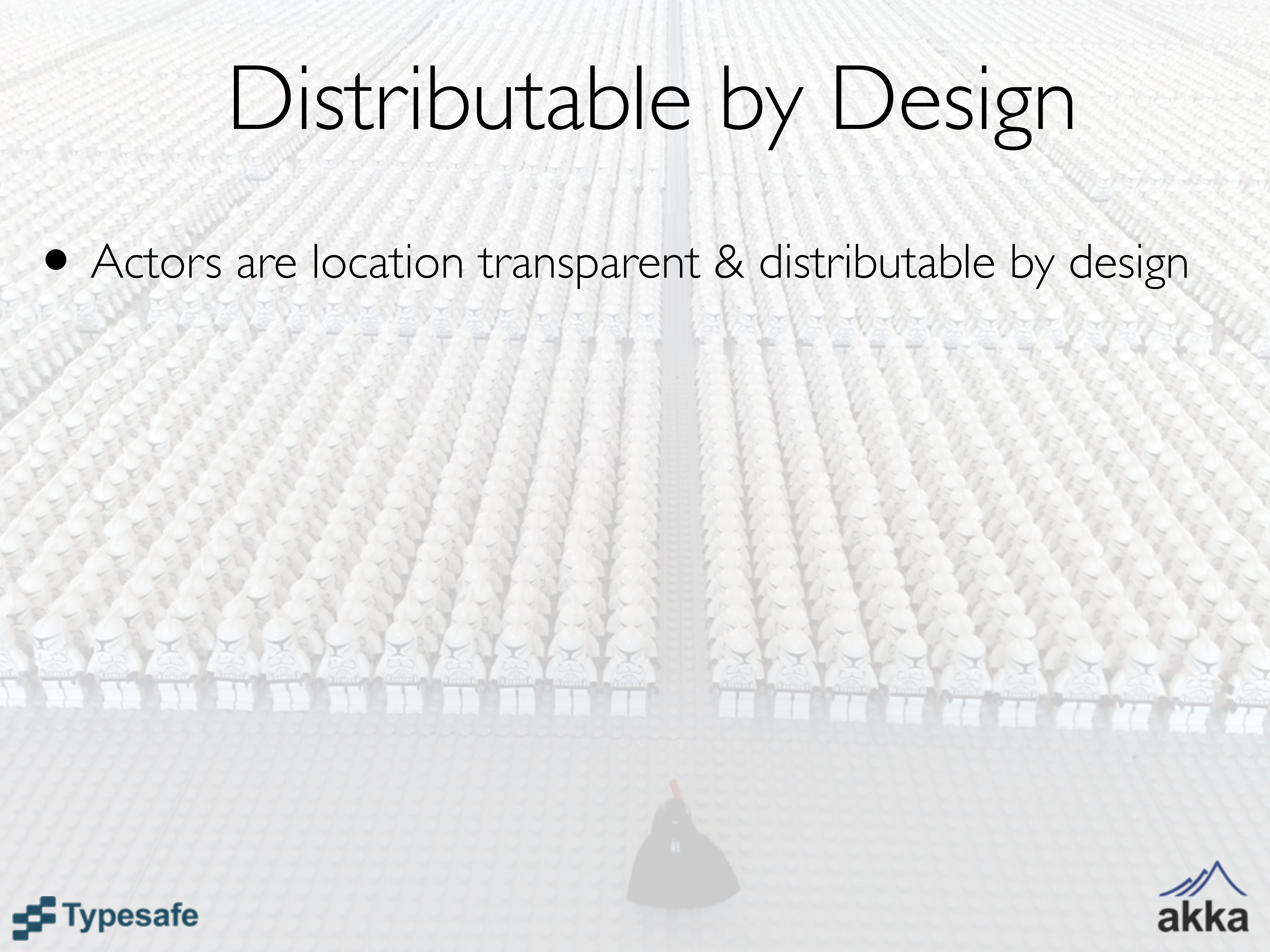


Distributable by Design

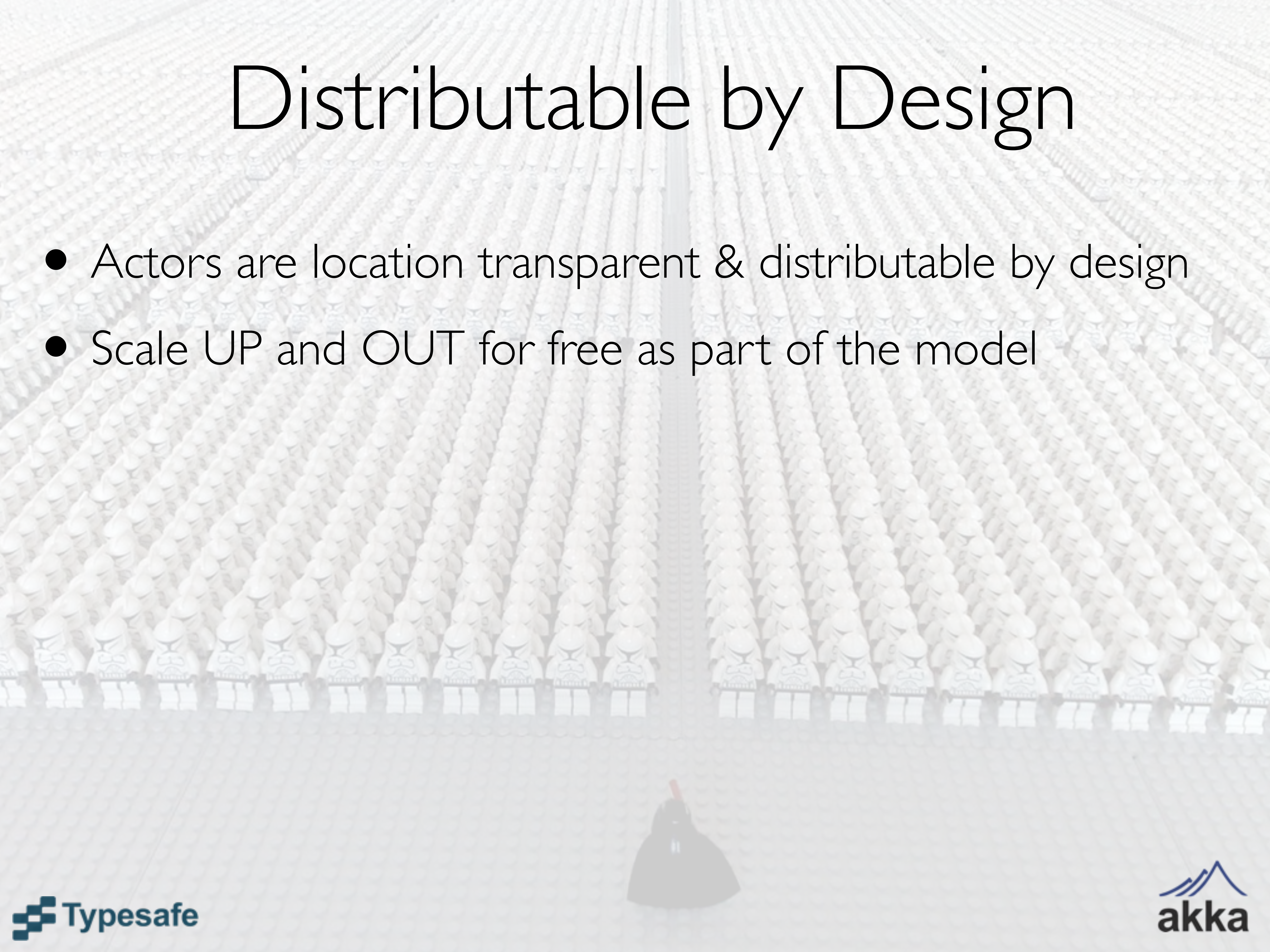
Distributable by Design

Distributable by Design

- Actors are location transparent & distributable by design



Distributable by Design



- Actors are location transparent & distributable by design
- Scale UP and OUT for free as part of the model

Distributable by Design

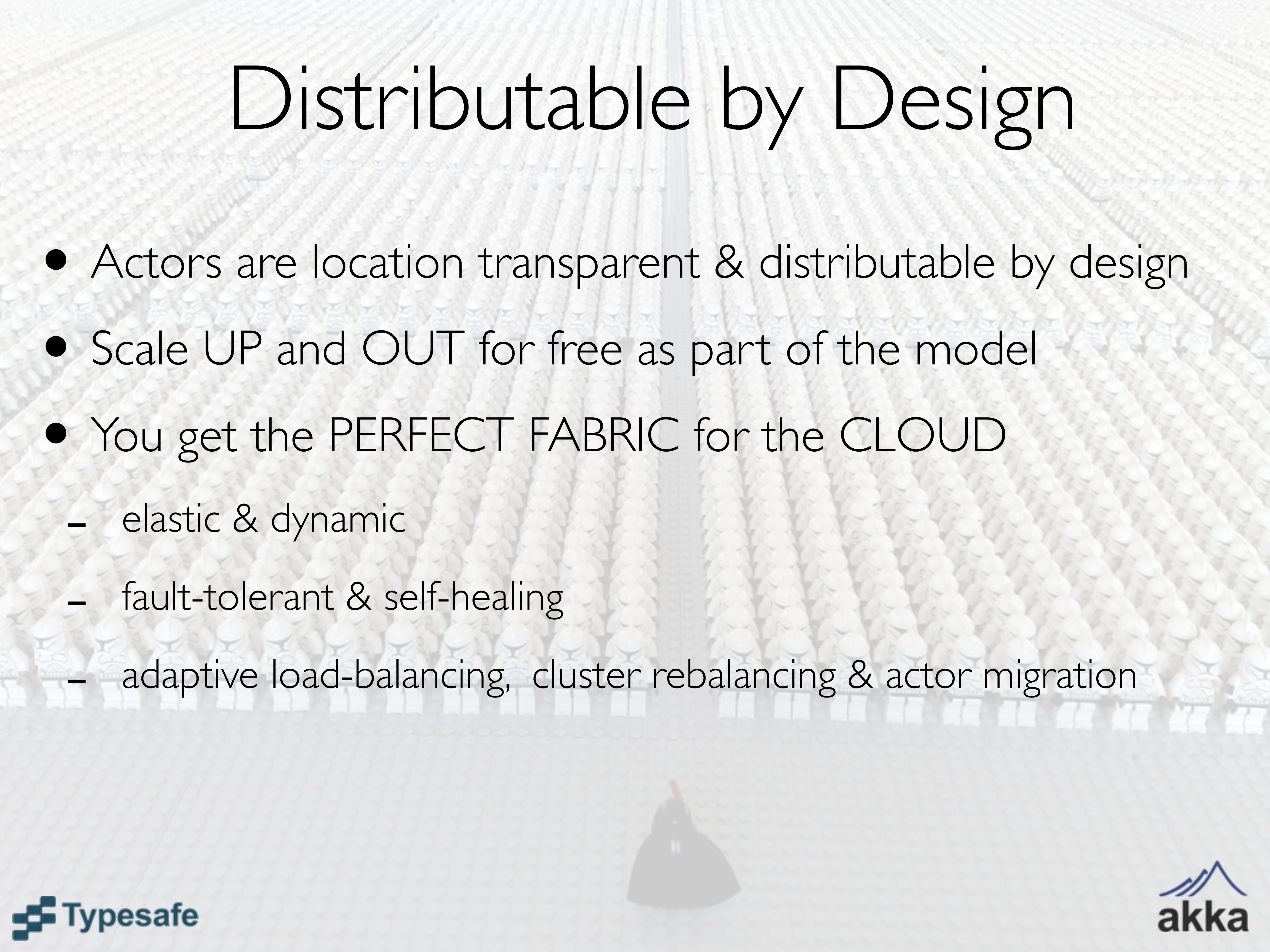
- Actors are location transparent & distributable by design
- Scale UP and OUT for free as part of the model
- You get the PERFECT FABRIC for the CLOUD

Distributable by Design

- Actors are location transparent & distributable by design
- Scale UP and OUT for free as part of the model
- You get the PERFECT FABRIC for the CLOUD
  - elastic & dynamic

Distributable by Design

- Actors are location transparent & distributable by design
- Scale UP and OUT for free as part of the model
- You get the PERFECT FABRIC for the CLOUD
  - elastic & dynamic
  - fault-tolerant & self-healing



Distributable by Design

- Actors are location transparent & distributable by design
- Scale UP and OUT for free as part of the model
- You get the PERFECT FABRIC for the CLOUD
  - elastic & dynamic
  - fault-tolerant & self-healing
  - adaptive load-balancing, cluster rebalancing & actor migration

Typesafe

akka

Distributable by Design

- Actors are location transparent & distributable by design
- Scale UP and OUT for free as part of the model
- You get the PERFECT FABRIC for the CLOUD
  - elastic & dynamic
  - fault-tolerant & self-healing
  - adaptive load-balancing, cluster rebalancing & actor migration
  - build extremely loosely coupled and dynamic systems that can change and adapt at runtime

How

can we achieve this?

Let’s use Actors

What is an Actor?

What is an Actor?

- Akka's unit of code organization is called an Actor

What is an Actor?

- Akka's unit of code organization is called an Actor
- Actors helps you create concurrent, scalable and fault-tolerant applications

What is an Actor?

- Akka's unit of code organization is called an Actor
- Actors helps you create concurrent, scalable and fault-tolerant applications
- Like Java EE servlets and session beans, Actors is a model for organizing your code that keeps many "policy decisions" separate from the business logic

What is an Actor?

- Akka's unit of code organization is called an Actor
- Actors helps you create concurrent, scalable and fault-tolerant applications
- Like Java EE servlets and session beans, Actors is a model for organizing your code that keeps many "policy decisions" separate from the business logic
- Actors may be new to many in the Java community, but they are a tried-and-true concept (Hewitt 1973) used for many years in telecom systems with 9 nines uptime

What can I use Actors for?

What can I use Actors for?

In different scenarios, an Actor may be an alternative to:

What can I use Actors for?

In different scenarios, an Actor may be an alternative to:

- a thread

What can I use Actors for?

In different scenarios, an Actor may be an alternative to:

- a thread
- an object instance or component

What can I use Actors for?

In different scenarios, an Actor may be an alternative to:

- a thread
- an object instance or component
- a callback or listener

What can I use Actors for?

In different scenarios, an Actor may be an alternative to:

- a thread
- an object instance or component
- a callback or listener
- a singleton or service

What can I use Actors for?

In different scenarios, an Actor may be an alternative to:

- a thread
- an object instance or component
- a callback or listener
- a singleton or service
- a router, load-balancer or pool

What can I use Actors for?

In different scenarios, an Actor may be an alternative to:

- a thread
- an object instance or component
- a callback or listener
- a singleton or service
- a router, load-balancer or pool
- a Java EE Session Bean or Message-Driven Bean

What can I use Actors for?

In different scenarios, an Actor may be an alternative to:

- a thread
- an object instance or component
- a callback or listener
- a singleton or service
- a router, load-balancer or pool
- a Java EE Session Bean or Message-Driven Bean
- an out-of-process service

What can I use Actors for?

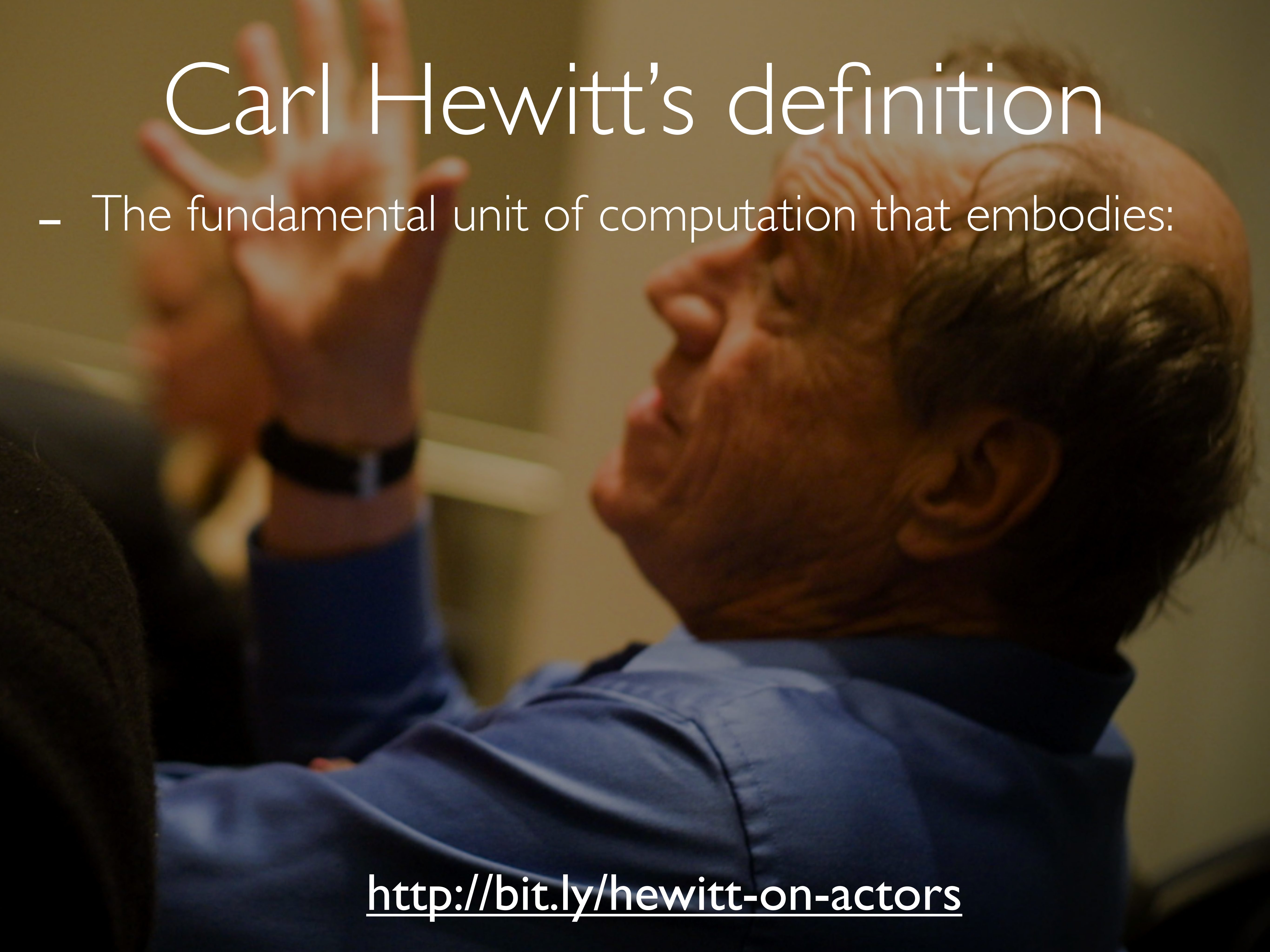
In different scenarios, an Actor may be an alternative to:

- a thread
- an object instance or component
- a callback or listener
- a singleton or service
- a router, load-balancer or pool
- a Java EE Session Bean or Message-Driven Bean
- an out-of-process service
- a Finite State Machine (FSM)

So, what is the Actor Model?

Carl Hewitt's definition

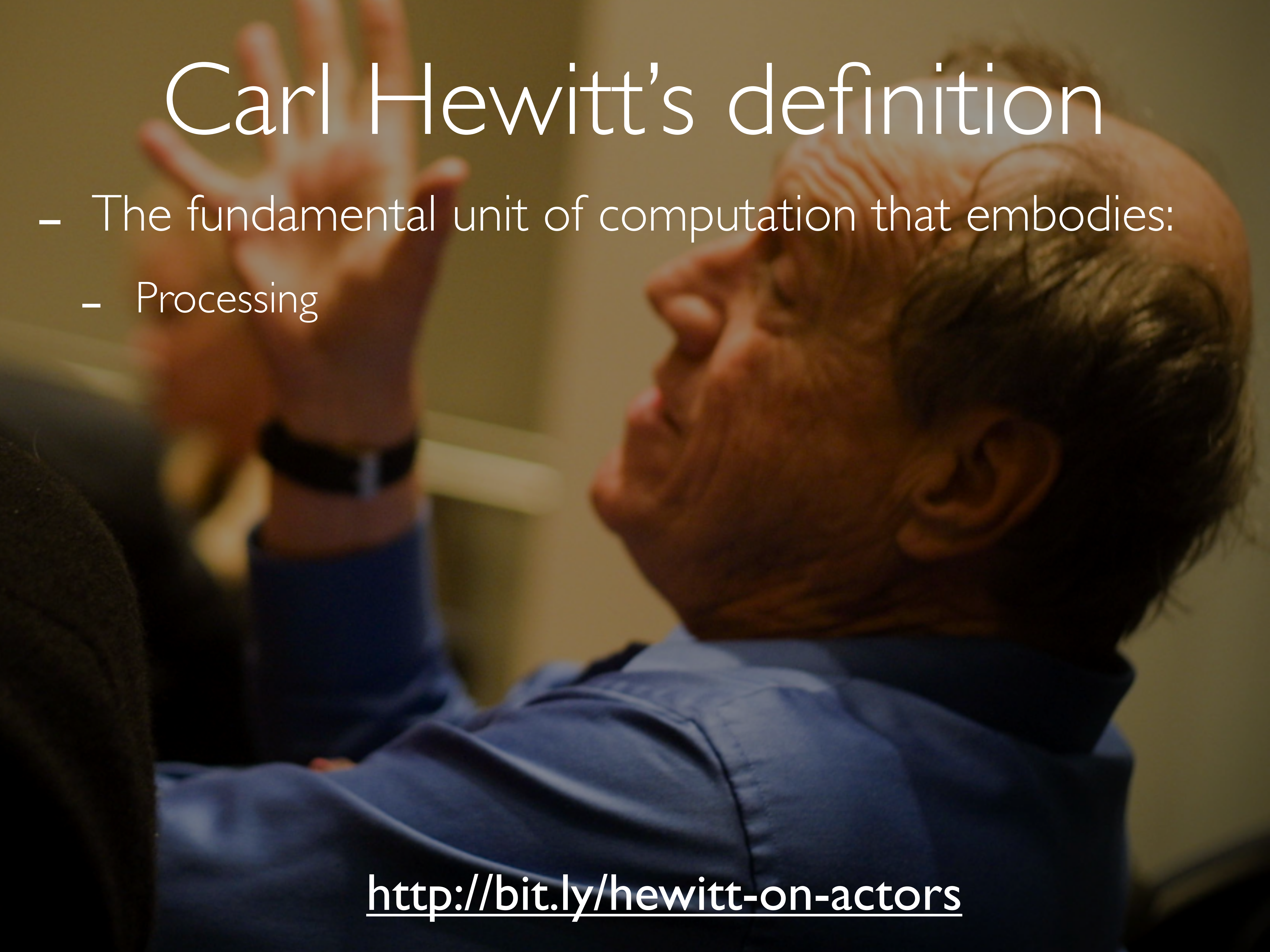
http://bit.ly/hewitt-on-actors



Carl Hewitt's definition

- The fundamental unit of computation that embodies:

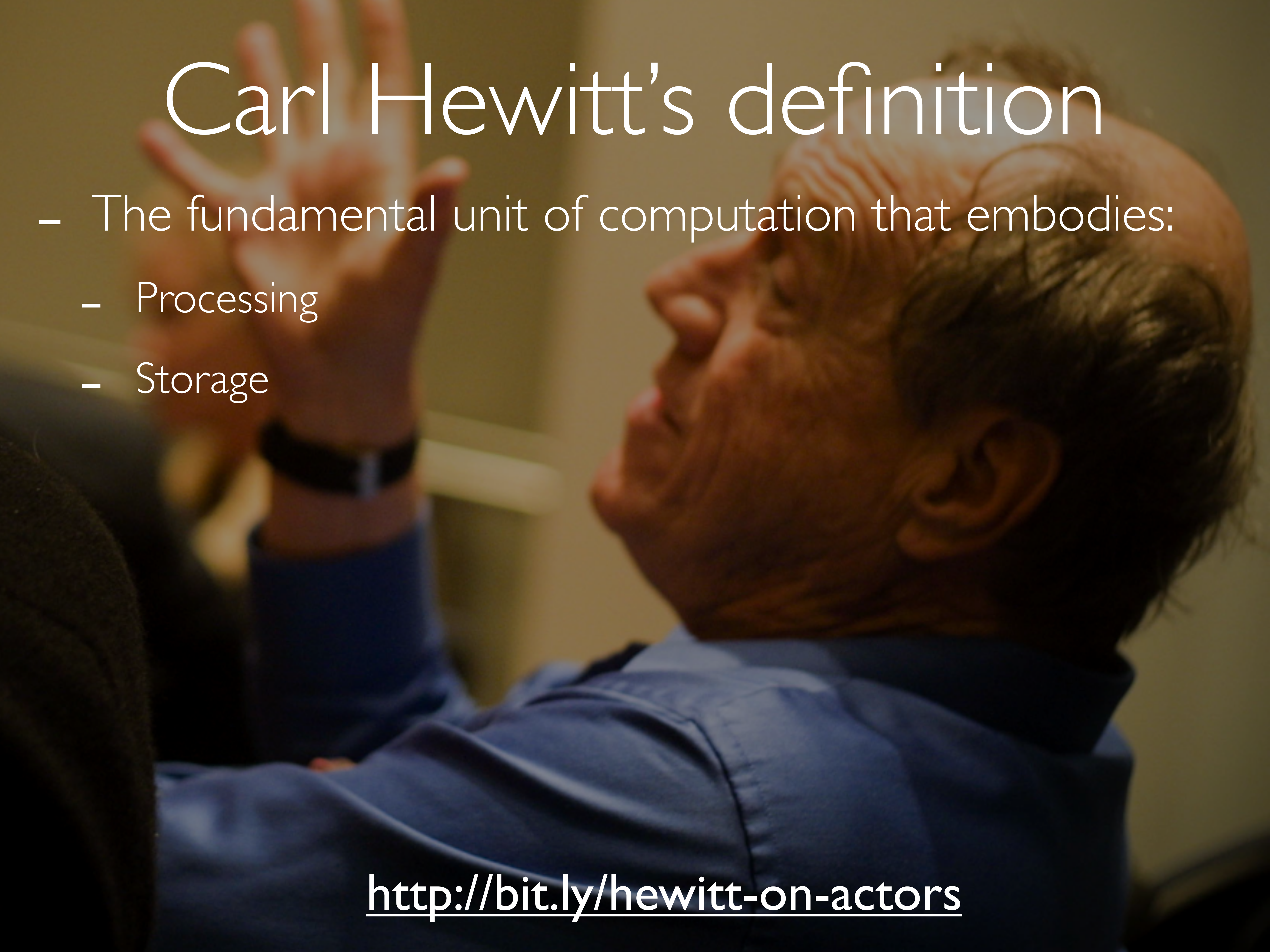
http://bit.ly/hewitt-on-actors



Carl Hewitt's definition

- The fundamental unit of computation that embodies:
  - Processing

http://bit.ly/hewitt-on-actors



Carl Hewitt's definition

- The fundamental unit of computation that embodies:
  - Processing
  - Storage

http://bit.ly/hewitt-on-actors

Carl Hewitt's definition

- The fundamental unit of computation that embodies:
  - Processing
  - Storage
  - Communication

http://bit.ly/hewitt-on-actors

Carl Hewitt’s definition

- The fundamental unit of computation that embodies:
  - Processing
  - Storage
  - Communication
- 3 axioms - When an Actor receives a message it can:

http://bit.ly/hewitt-on-actors

Carl Hewitt's definition

- The fundamental unit of computation that embodies:
  - Processing
  - Storage
  - Communication
- 3 axioms - When an Actor receives a message it can:
  - Create new Actors

http://bit.ly/hewitt-on-actors

Carl Hewitt's definition

- The fundamental unit of computation that embodies:
  - Processing
  - Storage
  - Communication
- 3 axioms - When an Actor receives a message it can:
  - Create new Actors
  - Send messages to Actors it knows

http://bit.ly/hewitt-on-actors

Carl Hewitt's definition

- The fundamental unit of computation that embodies:
  - Processing
  - Storage
  - Communication
- 3 axioms - When an Actor receives a message it can:
  - Create new Actors
  - Send messages to Actors it knows
  - Designate how it should handle the next message it receives

http://bit.ly/hewitt-on-actors

4 core Actor operations

0. DEFINE
1. CREATE
2. SEND
3. BECOME
4. SUPERVISE

0. DEFINE

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}
```

0. DEFINE

Define the message(s) the Actor should be able to respond to

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}
```

0. DEFINE

Define the message(s) the Actor should be able to respond to

Define the Actor class

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}
```

0. DEFINE

Define the message(s) the Actor should be able to respond to

Define the Actor class

```scala
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}
```

Define the Actor's behavior

1. CREATE

- CREATE - creates a new instance of an Actor
- Extremely lightweight (2.7 Million per Gb RAM)
- Very strong encapsulation - encapsulates:
  - state
  - behavior
  - message queue
- State & behavior is indistinguishable from each other
- Only way to observe state is by sending an actor a message and see how it reacts

CREATE Actor

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
```

Typesafe

akka

CREATE Actor

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who                                " + who)
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
```

Create an Actor system

CREATE Actor

```scala
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
```

Create an Actor system

Actor configuration

CREATE Actor

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
```

Create an Actor system

Actor configuration

Give it a name

CREATE Actor

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
```

Create an Actor system

Actor configuration

Create the Actor

Give it a name

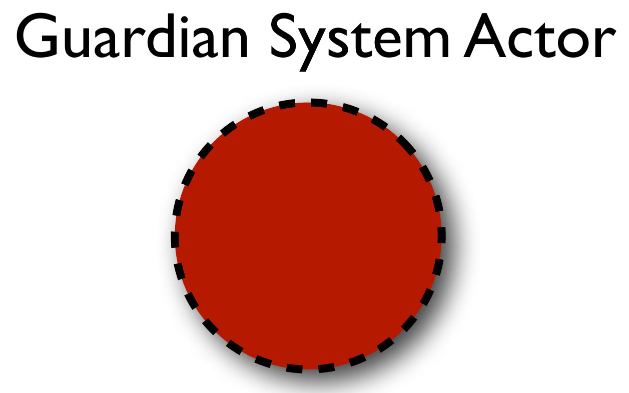
CREATE Actor

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
```

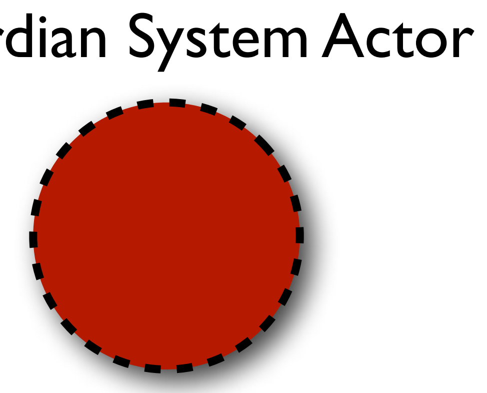
Typesafe

akka

Actors can form hierarchies

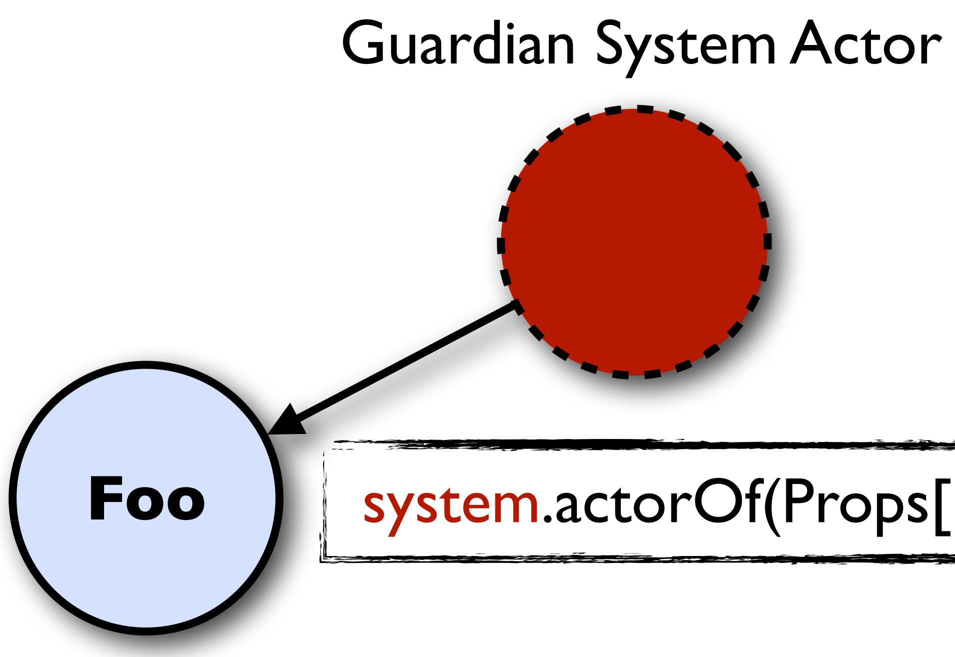


Actors can form hierarchies



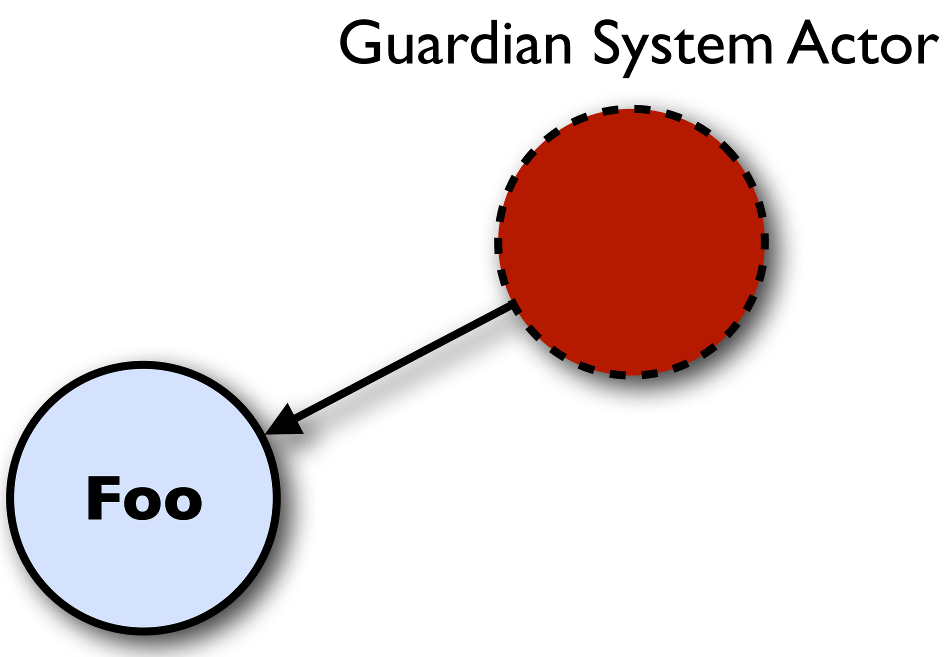
```
system.actorOf(Props[Foo], "Foo")
```

Actors can form hierarchies



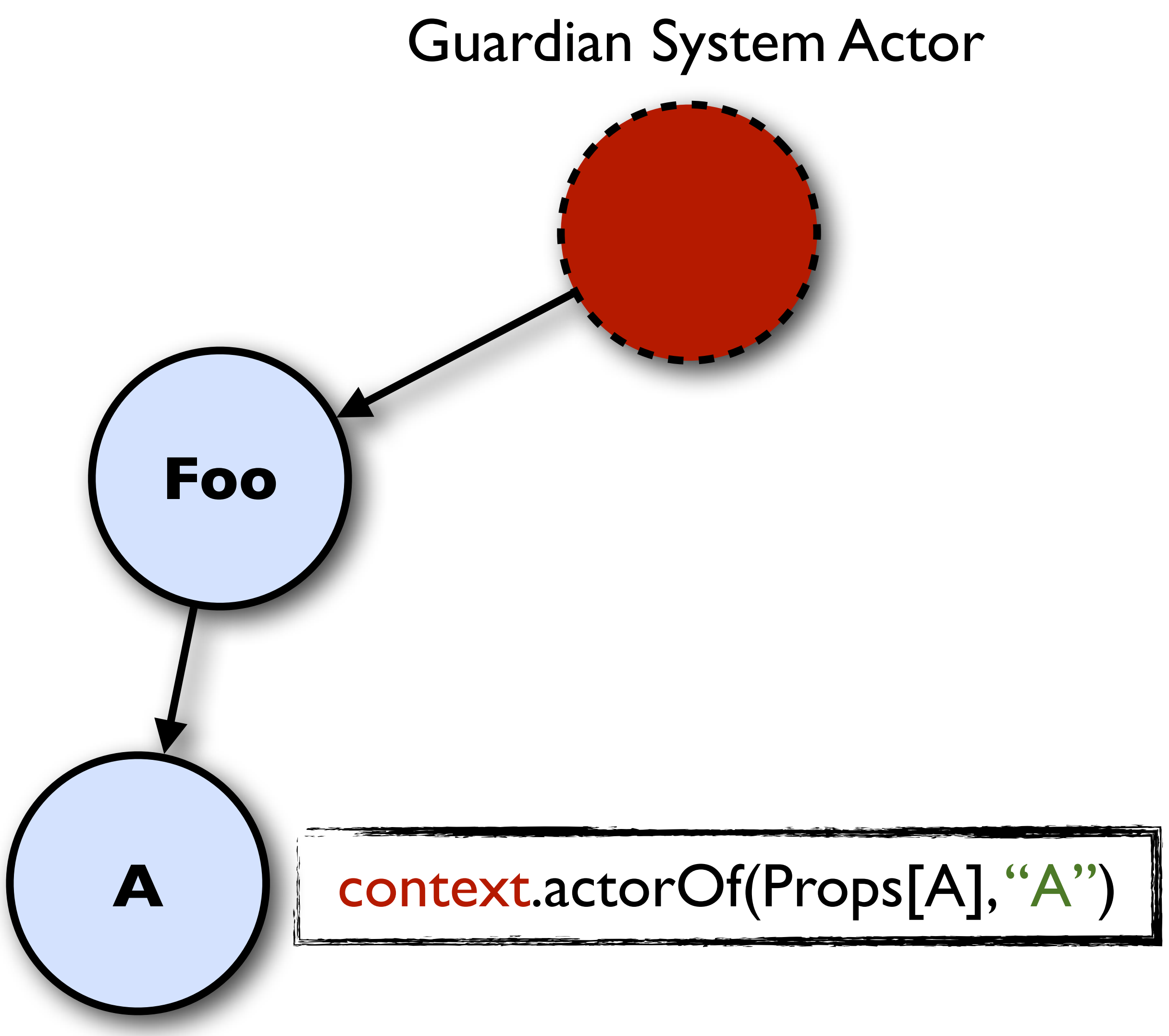
```
system.actorOf(Props[Foo], "Foo")
```

Actors can form hierarchies



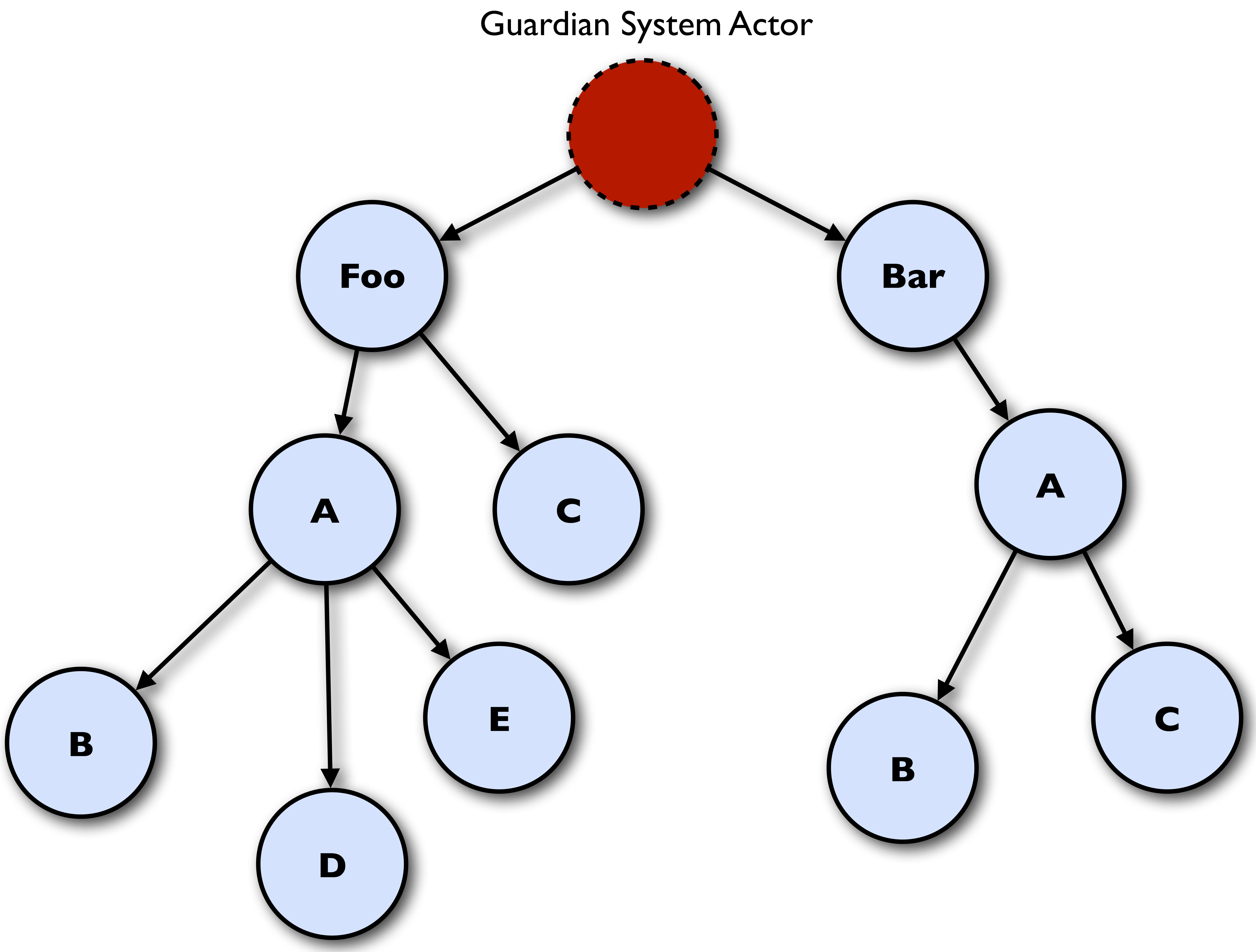
```
context.actorOf(Props[A], "A")
```

Actors can form hierarchies

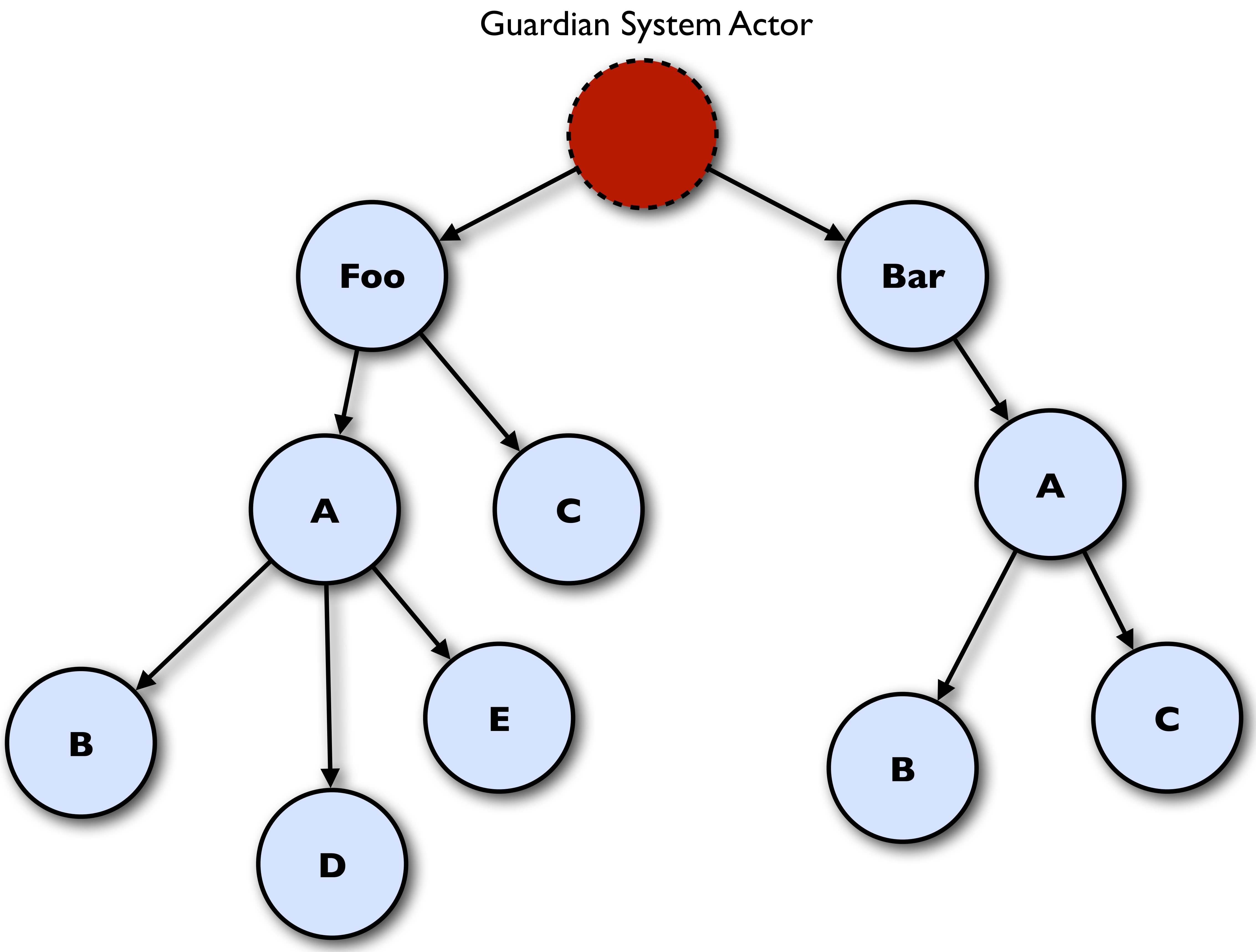


```
context.actorOf(Props[A], "A")
```

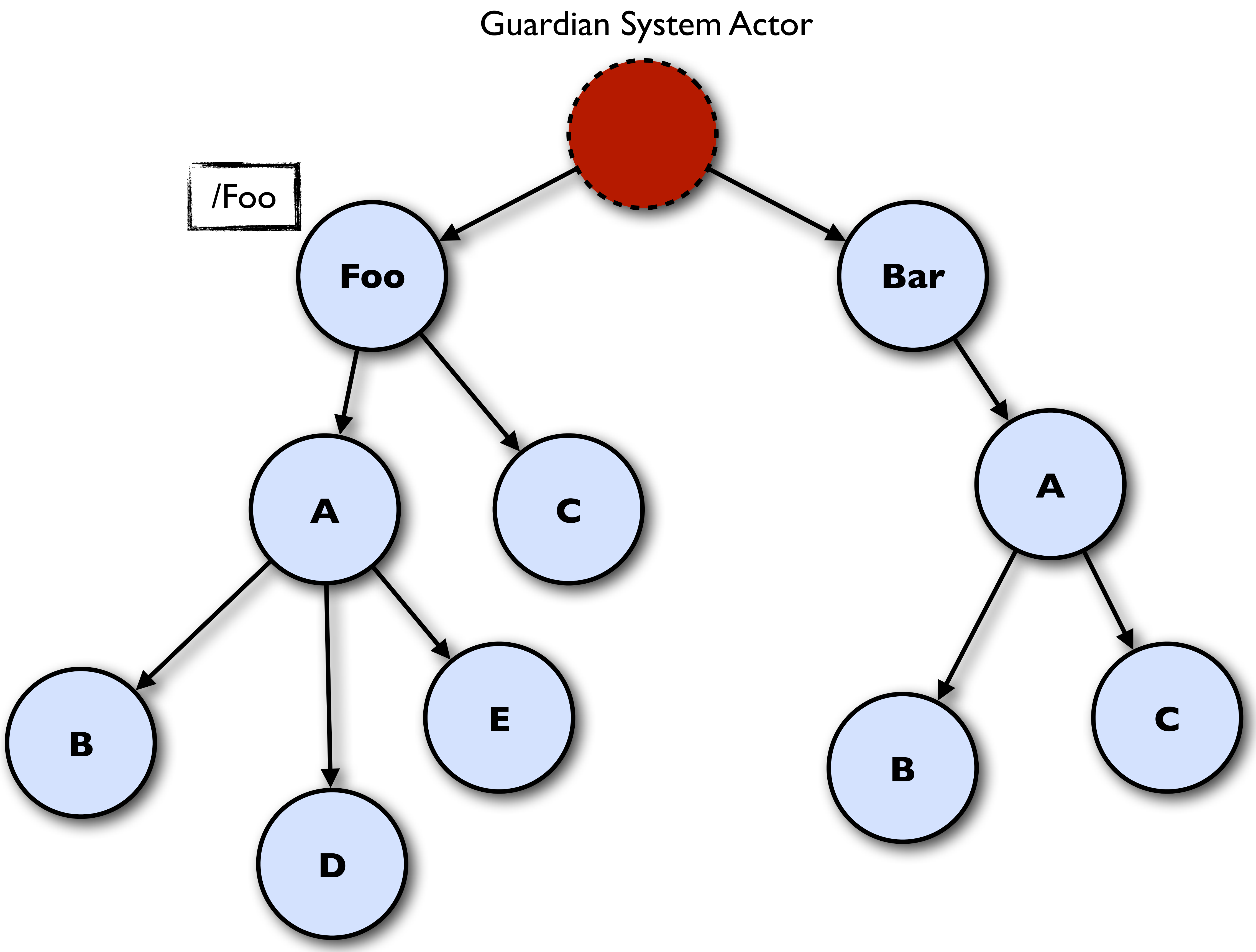
Actors can form hierarchies



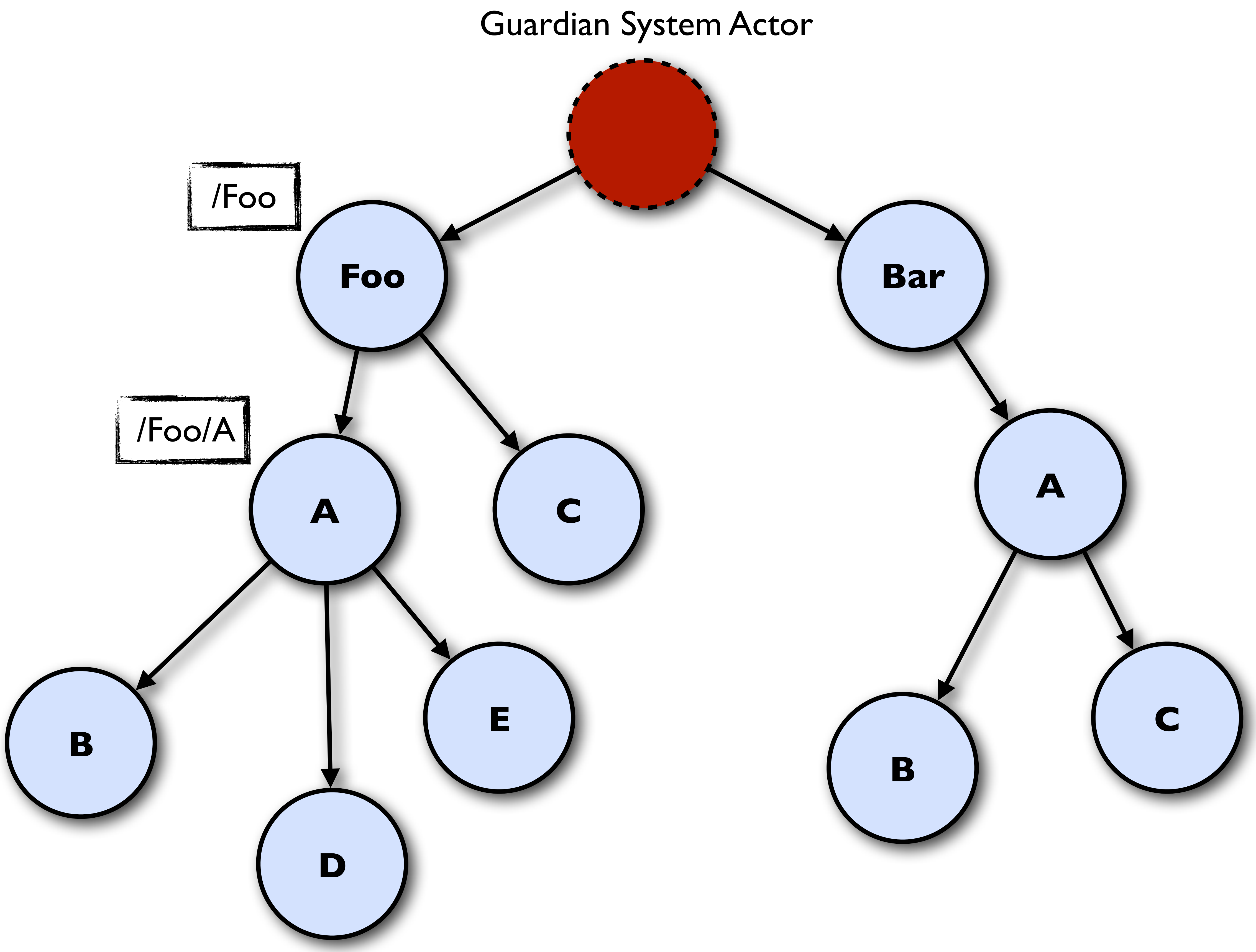
Name resolution - like a file-system



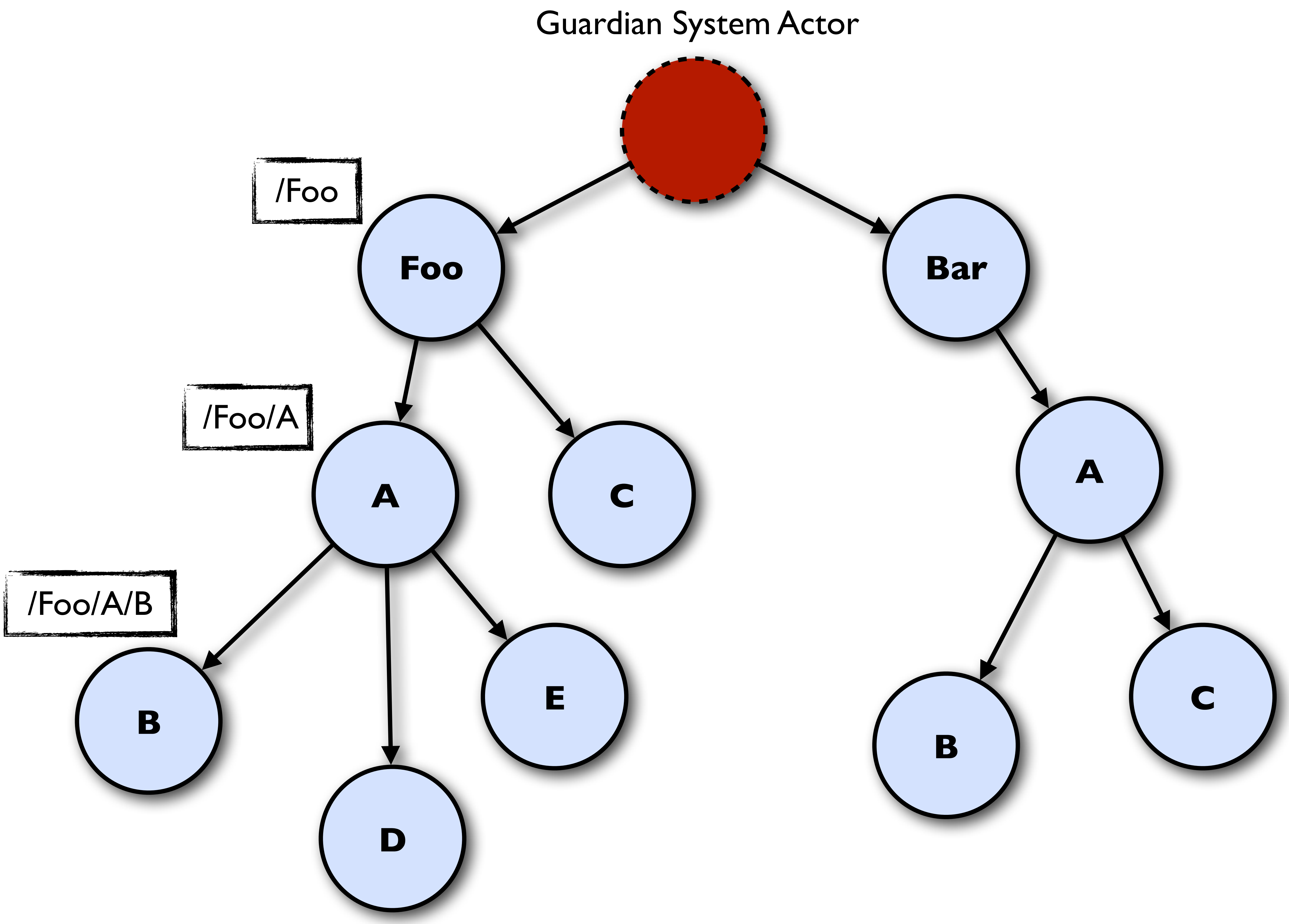
Name resolution - like a file-system



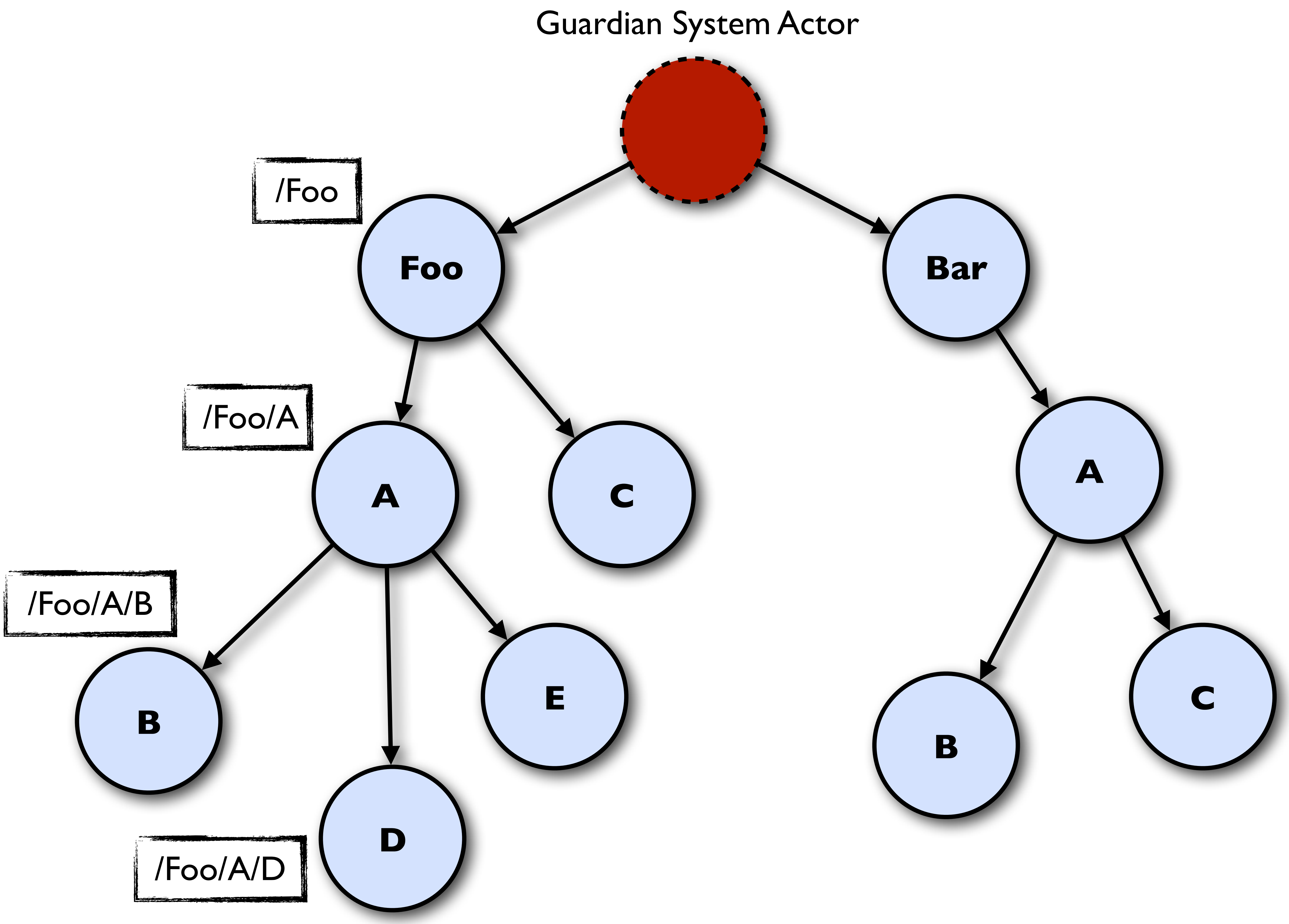
Name resolution - like a file-system



Name resolution - like a file-system



Name resolution - like a file-system



2. SEND

2. SEND

- SEND - sends a message to an Actor

2. SEND

- SEND - sends a message to an Actor
- Asynchronous and Non-blocking - Fire-forget

2. SEND

- SEND - sends a message to an Actor
- Asynchronous and Non-blocking - Fire-forget
- EVERYTHING is asynchronous and lockless

2. SEND

- SEND - sends a message to an Actor
- Asynchronous and Non-blocking - Fire-forget
- EVERYTHING is asynchronous and lockless
- Everything happens Reactively

2. SEND

- SEND - sends a message to an Actor
- Asynchronous and Non-blocking - Fire-forget
- EVERYTHING is asynchronous and lockless
- Everything happens Reactively
  - An Actor is passive until a message is sent to it, which triggers something within the Actor

2. SEND

- SEND - sends a message to an Actor
- Asynchronous and Non-blocking - Fire-forget
- EVERYTHING is asynchronous and lockless
- Everything happens Reactively
  - An Actor is passive until a message is sent to it, which triggers something within the Actor
  - Messages is the Kinetic Energy in an Actor system

2. SEND

- SEND - sends a message to an Actor
- Asynchronous and Non-blocking - Fire-forget
- EVERYTHING is asynchronous and lockless
- Everything happens Reactively
  - An Actor is passive until a message is sent to it, which triggers something within the Actor
  - Messages is the Kinetic Energy in an Actor system
  - Actors can have lots of buffered Potential Energy but can't do anything with it until it is triggered by a message

SEND message

```scala
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
greeter ! Greeting("Charlie Parker")
```

SEND message

```scala
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
greeter ! Greeting("Charlie Parker")
```

Send the message

Full example

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
greeter ! Greeting("Charlie Parker")
```

Routers

Load Balancing

Routers

```
val router =
 system.actorOf(
   Props[SomeActor].withRouter(
    RoundRobinRouter(nrOfInstances = 5)))
```

Router + Resizer

```
val resizer =
  DefaultResizer(lowerBound = 2, upperBound = 15)

val router =
  system.actorOf(
    Props[ExampleActor1].withRouter(
      RoundRobinRouter(resizer = Some(resizer))))
```

...or from config

```
akka.actor.deployment {
  /path/to/actor {
    router = round-robin
    nr-of-instances = 5
  }
}
```

…or from config

```
akka.actor.deployment {
  /path/to/actor {
    router = round-robin
    resizer {
      lower-bound = 12
      upper-bound = 15
    }
  }
}
```

3. BECOME

3. BECOME

- BECOME - dynamically redefines Actor's behavior

3. BECOME

- BECOME - dynamically redefines Actor's behavior
- Triggered reactively by receive of message

3. BECOME

- BECOME - dynamically redefines Actor's behavior
- Triggered reactively by receive of message
- In a type system analogy it is as if the object changed type - changed interface, protocol & implementation

3. BECOME

- BECOME - dynamically redefines Actor's behavior
- Triggered reactively by receive of message
- In a type system analogy it is as if the object changed type - changed interface, protocol & implementation
- Will now react differently to the messages it receives

3. BECOME

- BECOME - dynamically redefines Actor's behavior
- Triggered reactively by receive of message
- In a type system analogy it is as if the object changed type - changed interface, protocol & implementation
- Will now react differently to the messages it receives
- Behaviors are stacked & can be pushed and popped

Why would I want to do that?

Why would I want to do that?

- Let a highly contended Actor adaptively transform itself into an Actor Pool or a Router

Why would I want to do that?

- Let a highly contended Actor adaptively transform itself into an Actor Pool or a Router
- Implement an FSM (Finite State Machine)

Why would I want to do that?

- Let a highly contended Actor adaptively transform itself into an Actor Pool or a Router
- Implement an FSM (Finite State Machine)
- Implement graceful degradation

Why would I want to do that?

- Let a highly contended Actor adaptively transform itself into an Actor Pool or a Router
- Implement an FSM (Finite State Machine)
- Implement graceful degradation
- Spawn up (empty) generic Worker processes that can become whatever the Master currently needs

Why would I want to do that?

- Let a highly contended Actor adaptively transform itself into an Actor Pool or a Router
- Implement an FSM (Finite State Machine)
- Implement graceful degradation
- Spawn up (empty) generic Worker processes that can become whatever the Master currently needs
- Other: Use your imagination!

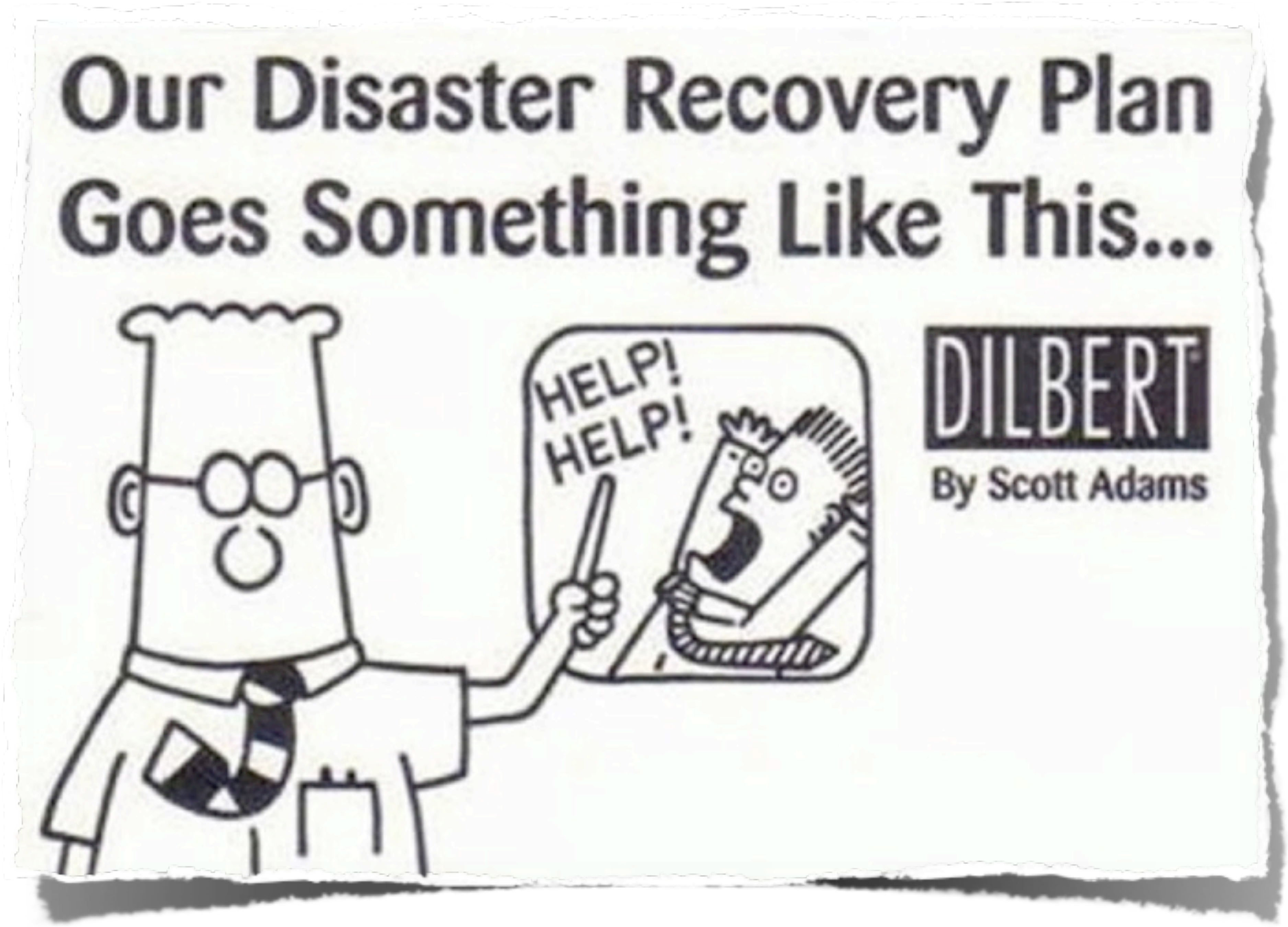
Why would I want to do that?

- Let a highly contended Actor adaptively transform itself into an Actor Pool or a Router
- Implement an FSM (Finite State Machine)
- Implement graceful degradation
- Spawn up (empty) generic Worker processes that can become whatever the Master currently needs
- Other: Use your imagination!
- Very useful once you get the used to it

become

```
context become {
  // new body
  case NewMessage =>
    ...
}
```

Failure Recovery



Failure Recovery in Java/C/C# etc.

Failure Recovery in Java/C/C# etc.

- You are given a SINGLE thread of control

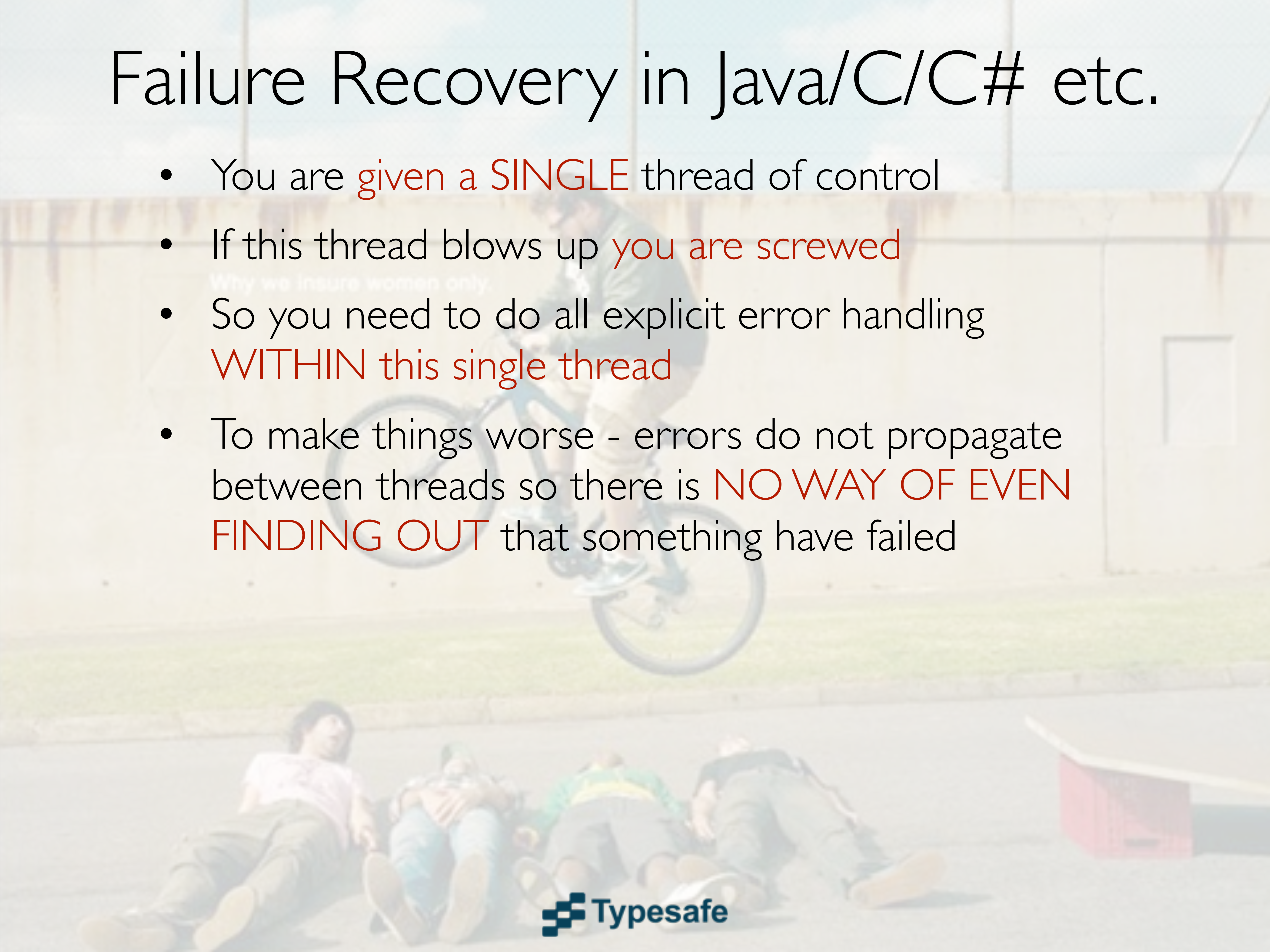
Failure Recovery in Java/C/C# etc.

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

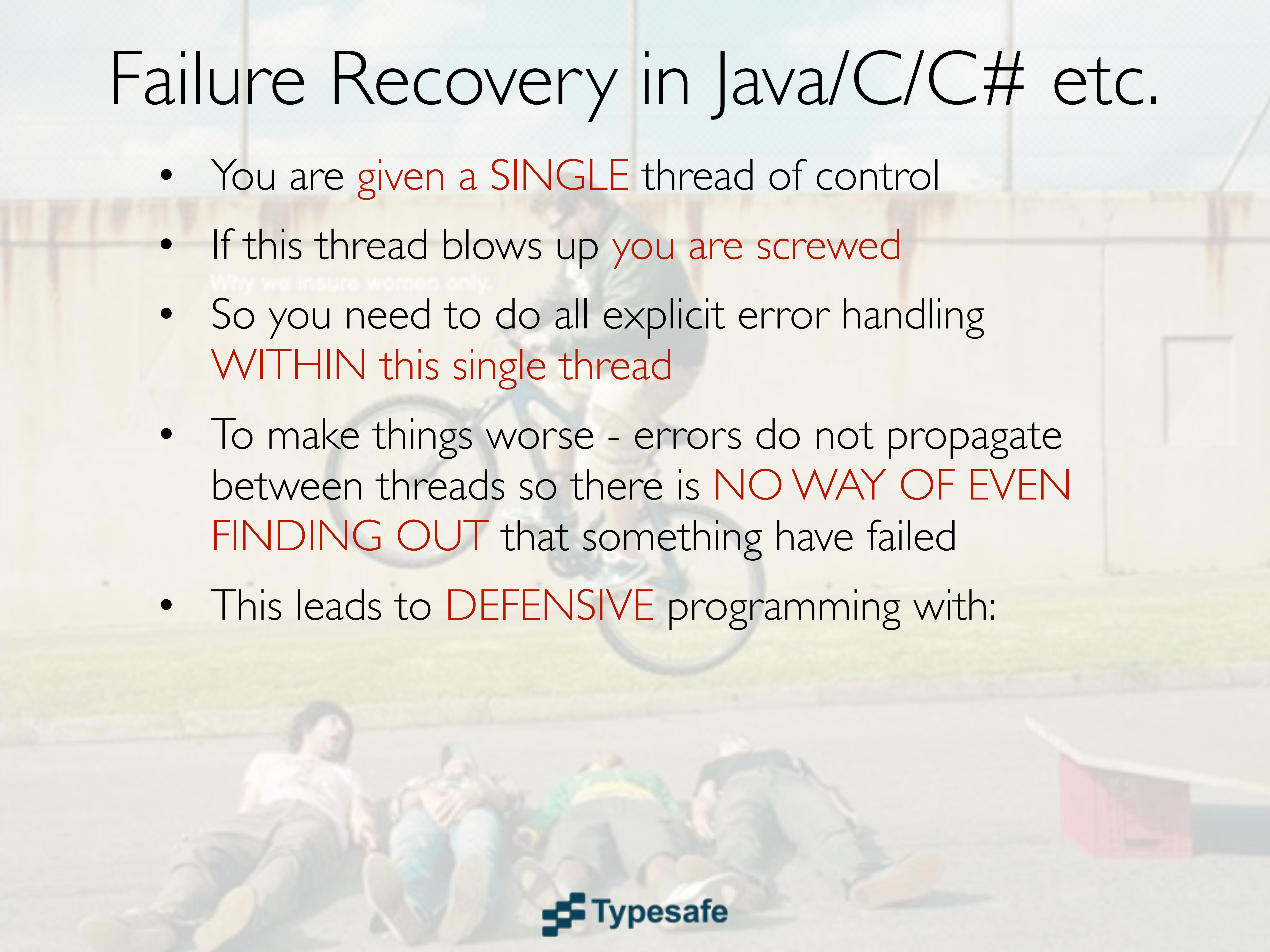
Failure Recovery in Java/C/C# etc.

- 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.



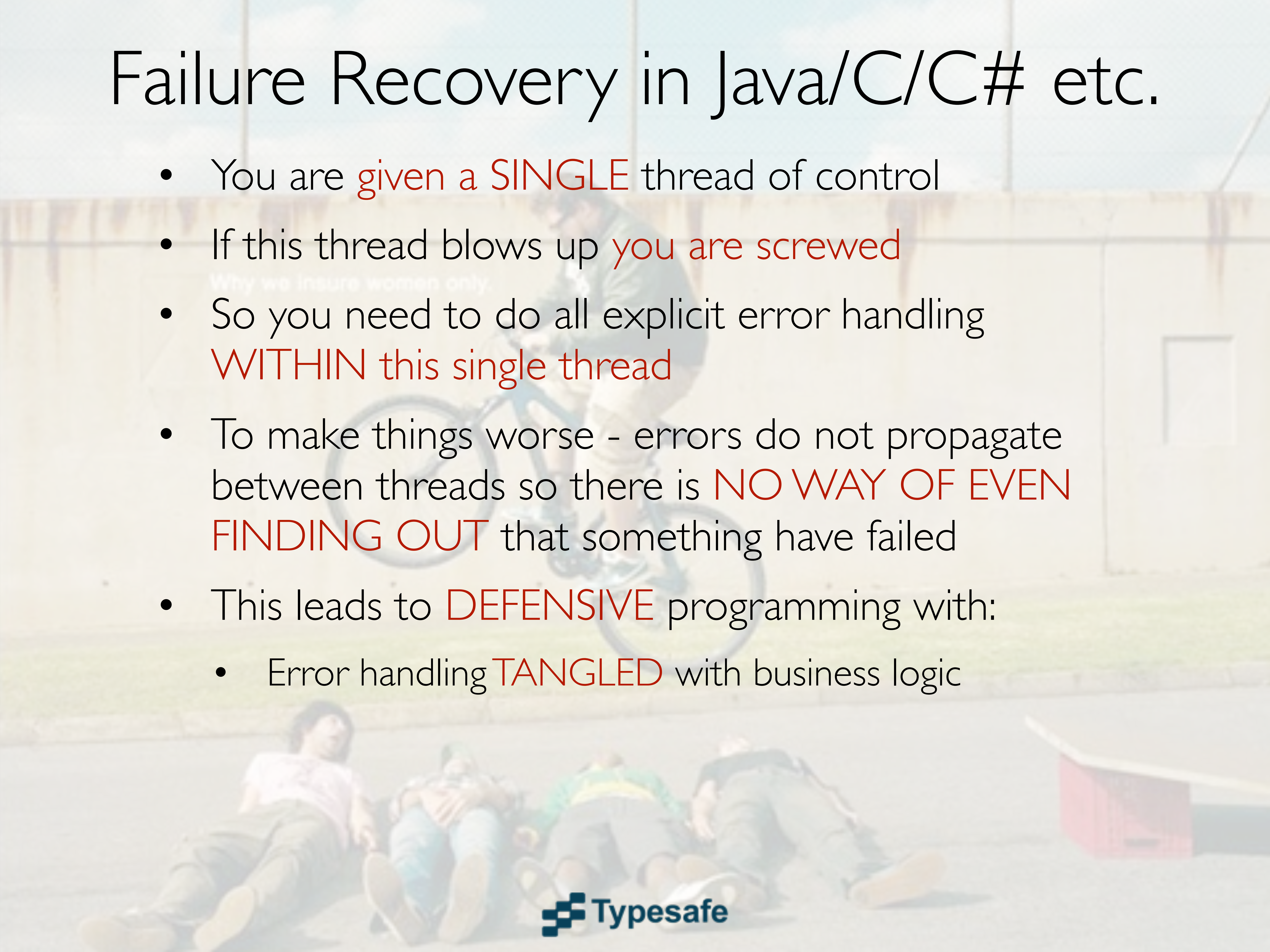
- You are given a SINGLE thread of control
- If this thread blows up you are screwed
- So you need to do all explicit error handling WITHIN this single thread
- To make things worse - errors do not propagate between threads so there is NO WAY OF EVEN FINDING OUT that something have failed



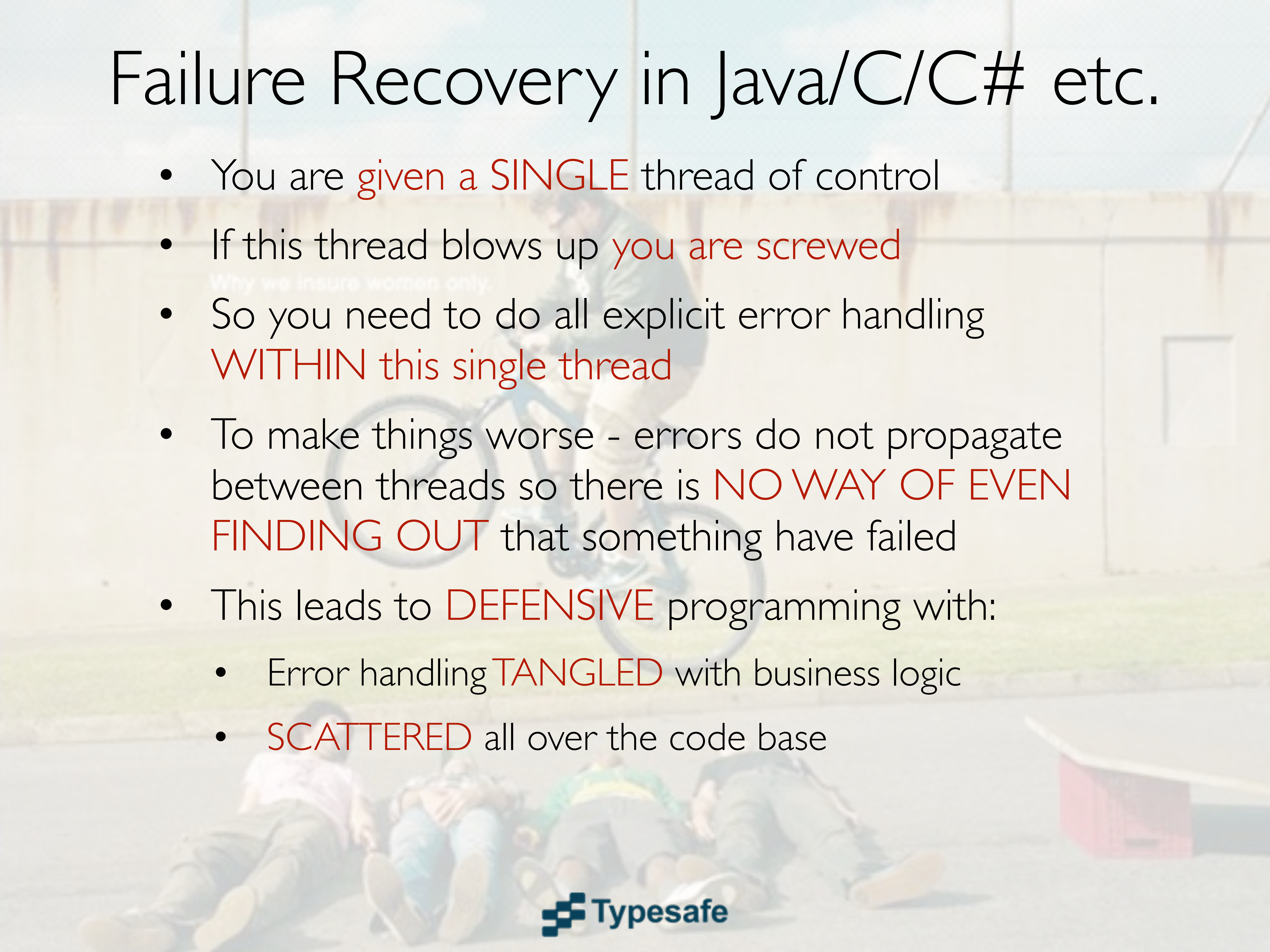
Failure Recovery in Java/C/C# etc.

- 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.



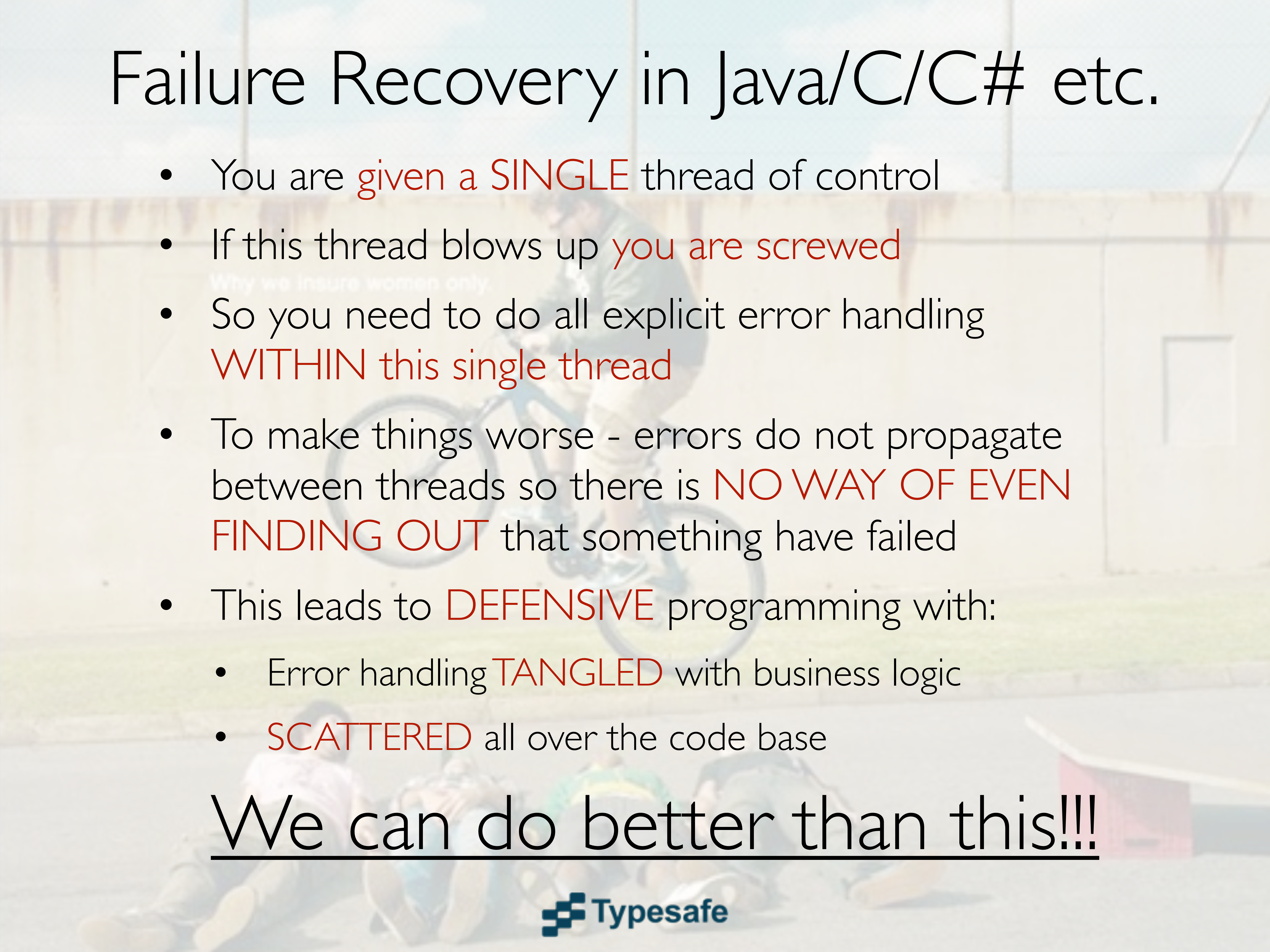
- You are given a SINGLE thread of control
- If this thread blows up you are screwed
- So you need to do all explicit error handling WITHIN this single thread
- To make things worse - errors do not propagate between threads so there is NO WAY OF EVEN FINDING OUT that something have failed
- This leads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

Typesafe



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

<u>We can do better than this!!!</u>

Typesafe

Just

LET IT CRASH

4. SUPERVISE

4. SUPERVISE

- SUPERVISE - manage another Actor's failures

4. SUPERVISE

- SUPERVISE - manage another Actor's failures
- Error handling in actors is handle by letting Actors monitor (supervise) each other for failure

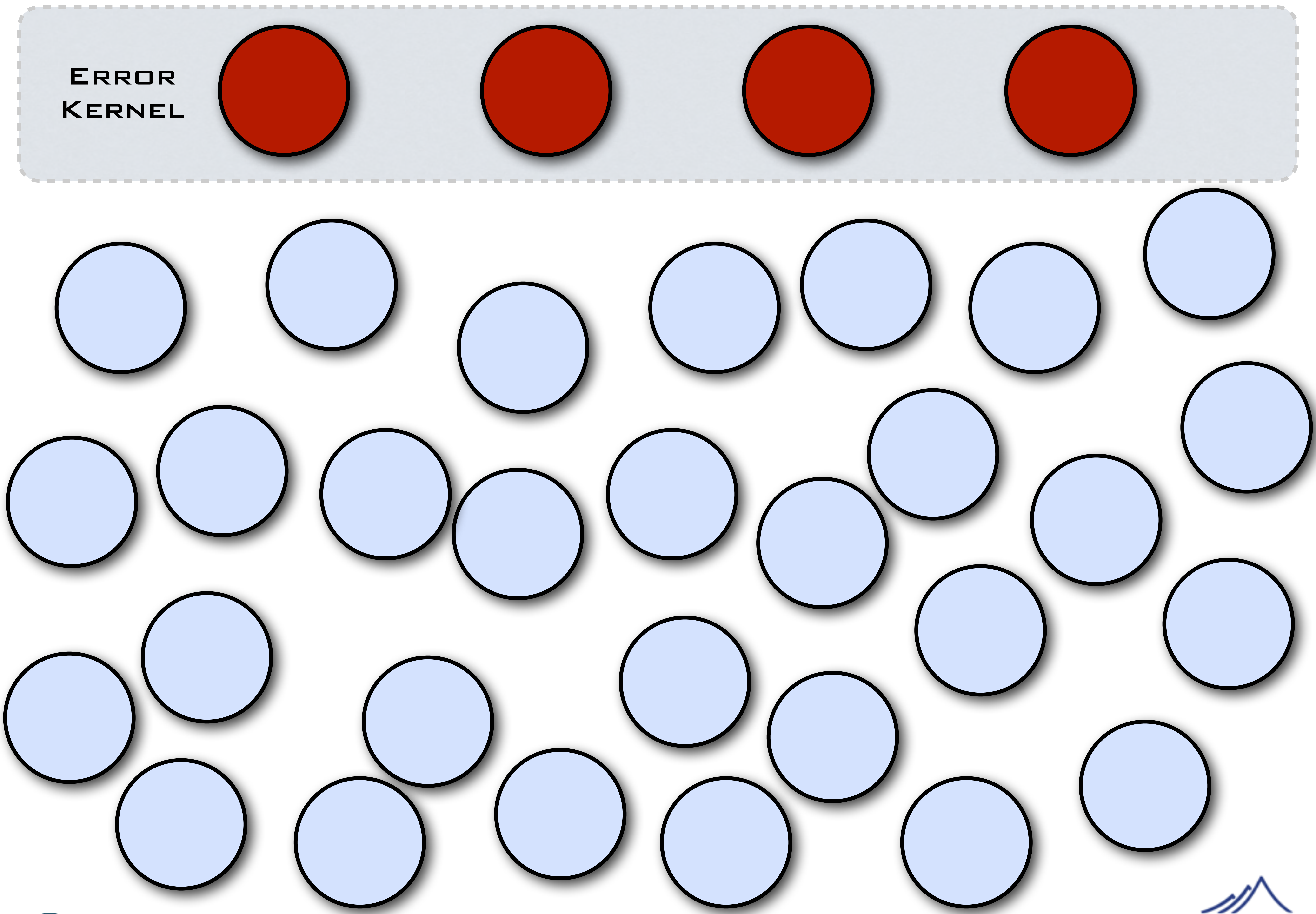
4. SUPERVISE

- SUPERVISE - manage another Actor's failures
- Error handling in actors is handle by letting Actors monitor (supervise) each other for failure
- This means that if an Actor crashes, a notification will be sent to his supervisor, who can react upon the failure

4. SUPERVISE

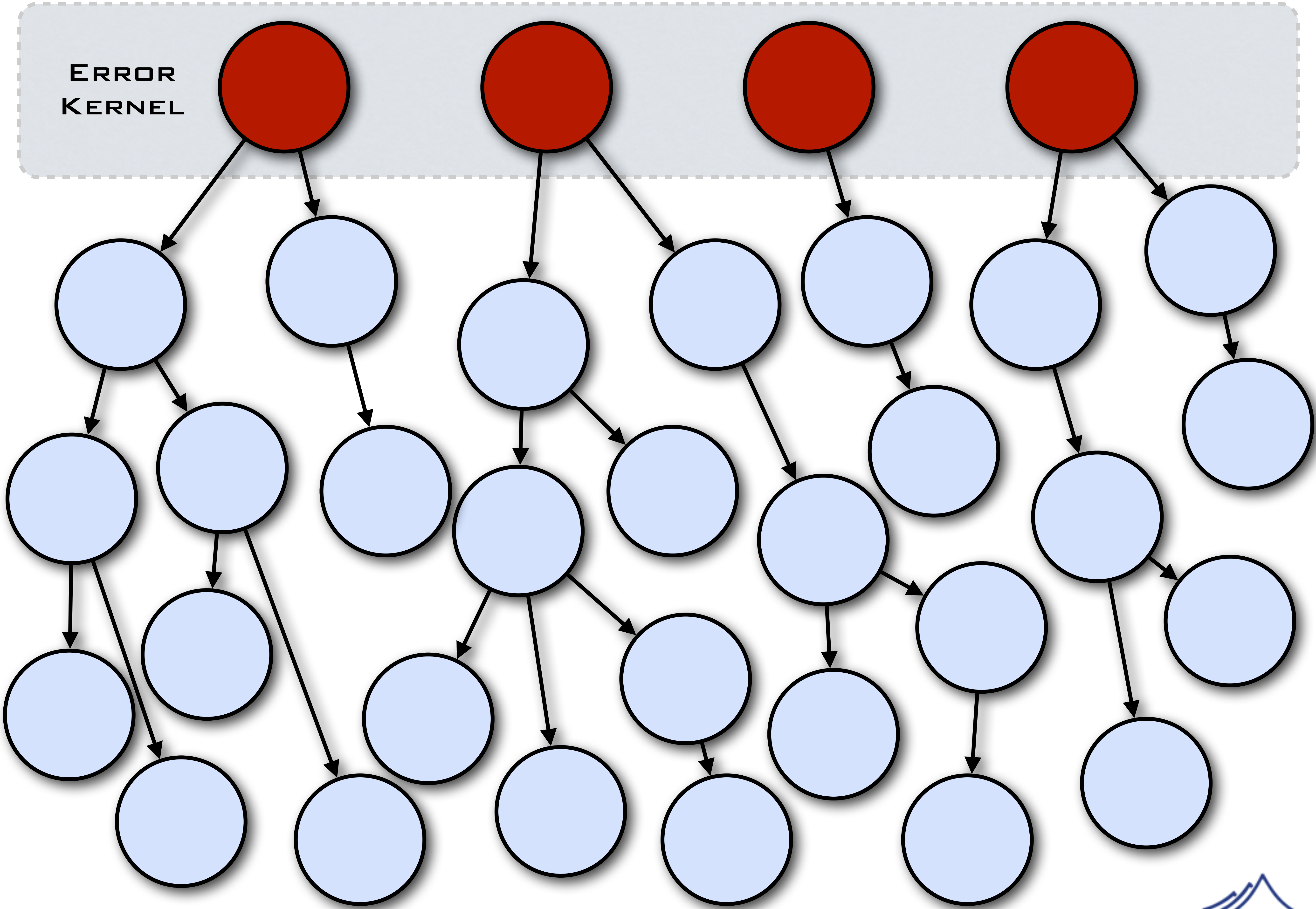
- SUPERVISE - manage another Actor's failures
- Error handling in actors is handle by letting Actors monitor (supervise) each other for failure
- This means that if an Actor crashes, a notification will be sent to his supervisor, who can react upon the failure
- This provides clean separation of processing and error handling

Fault-tolerant onion-layered Error Kernel



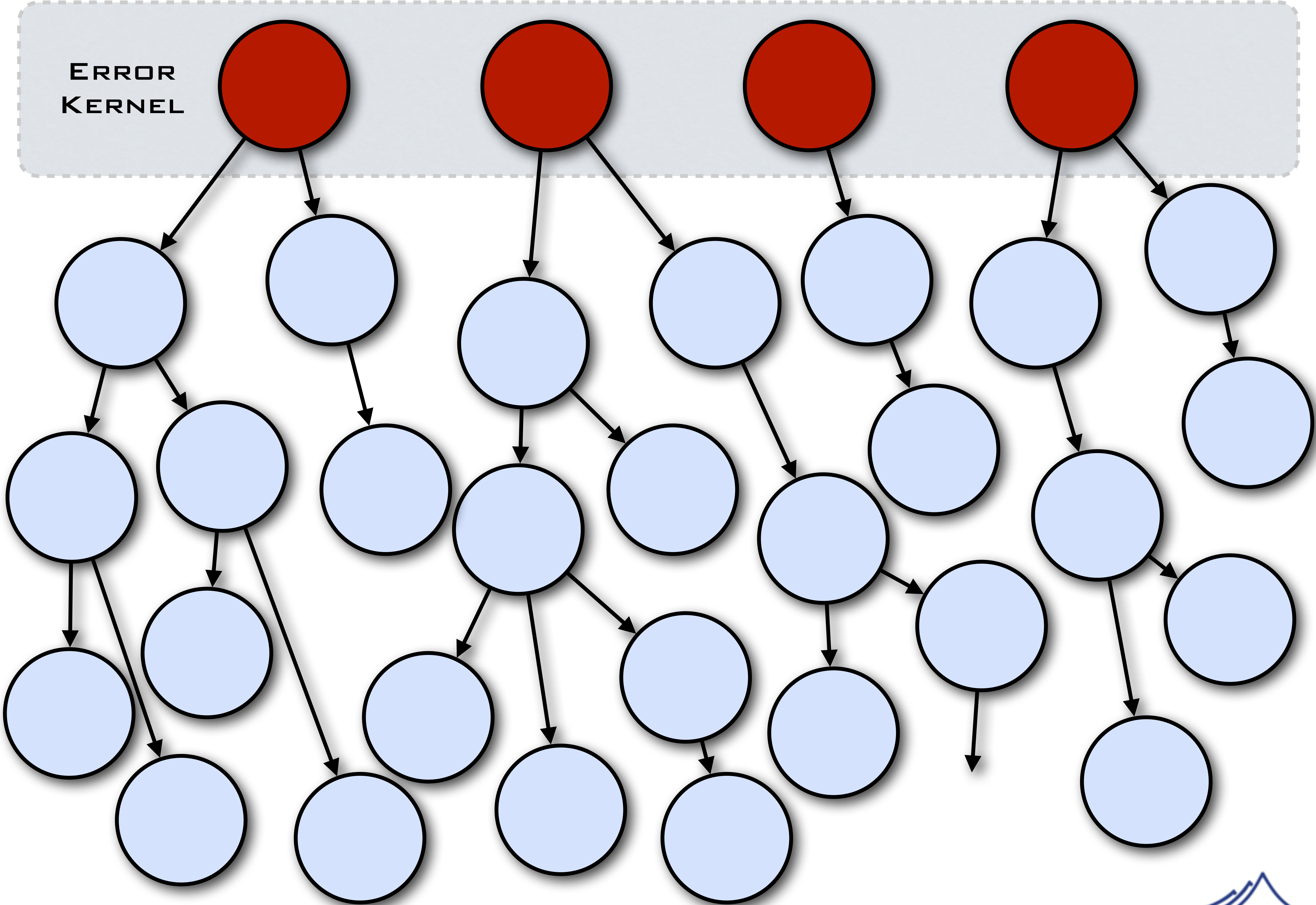
Typesafe

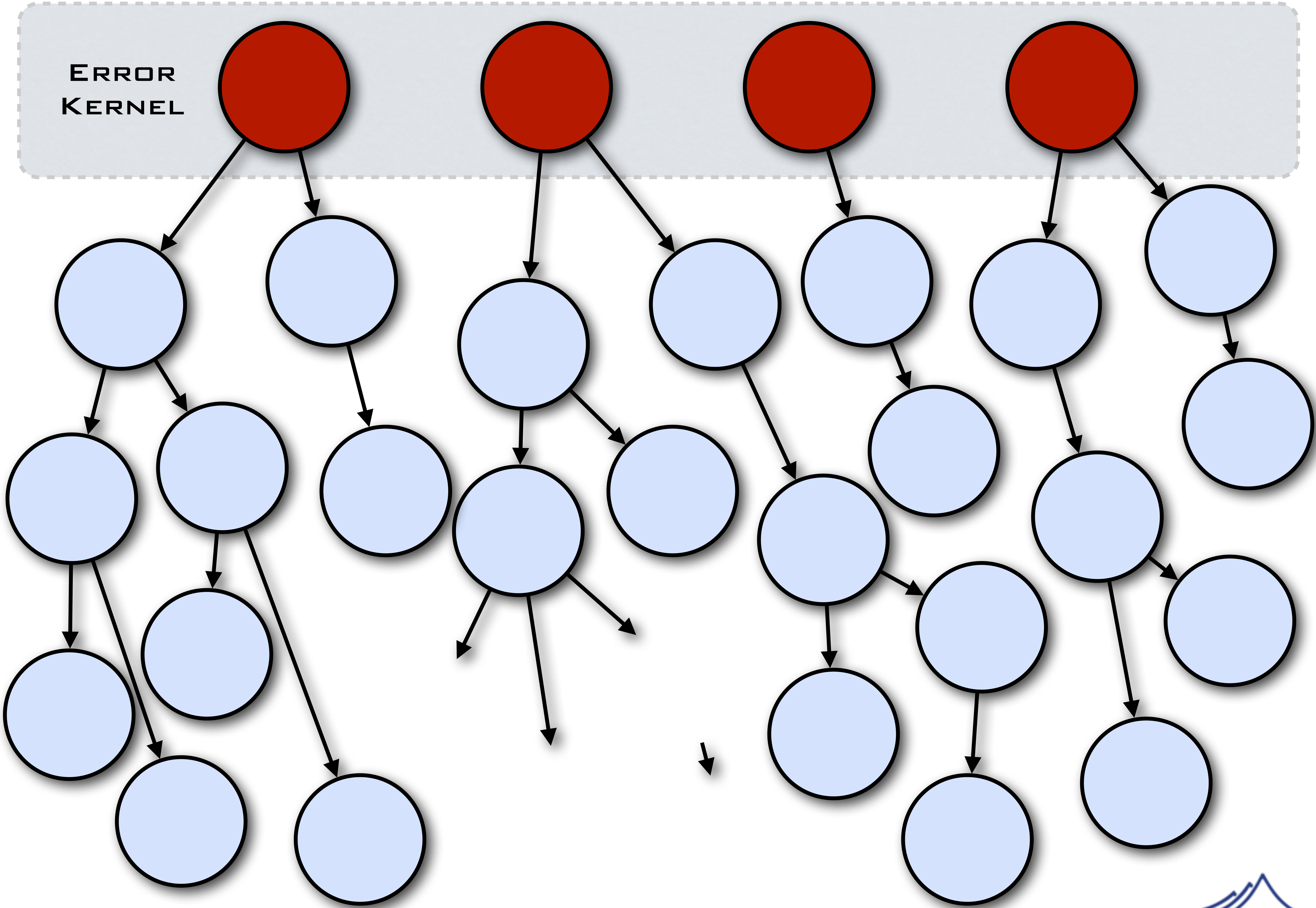
akka



Typesafe

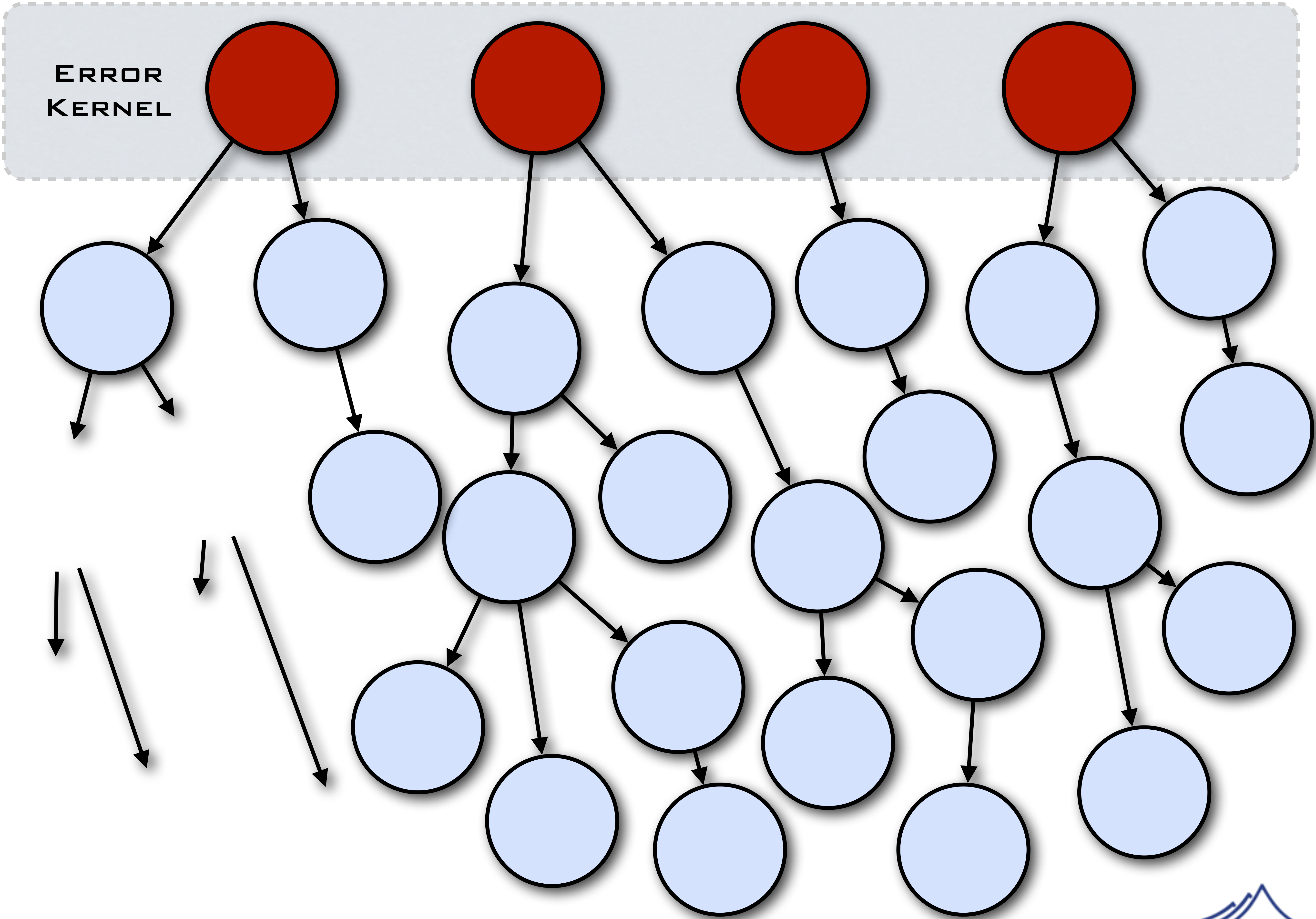
akka

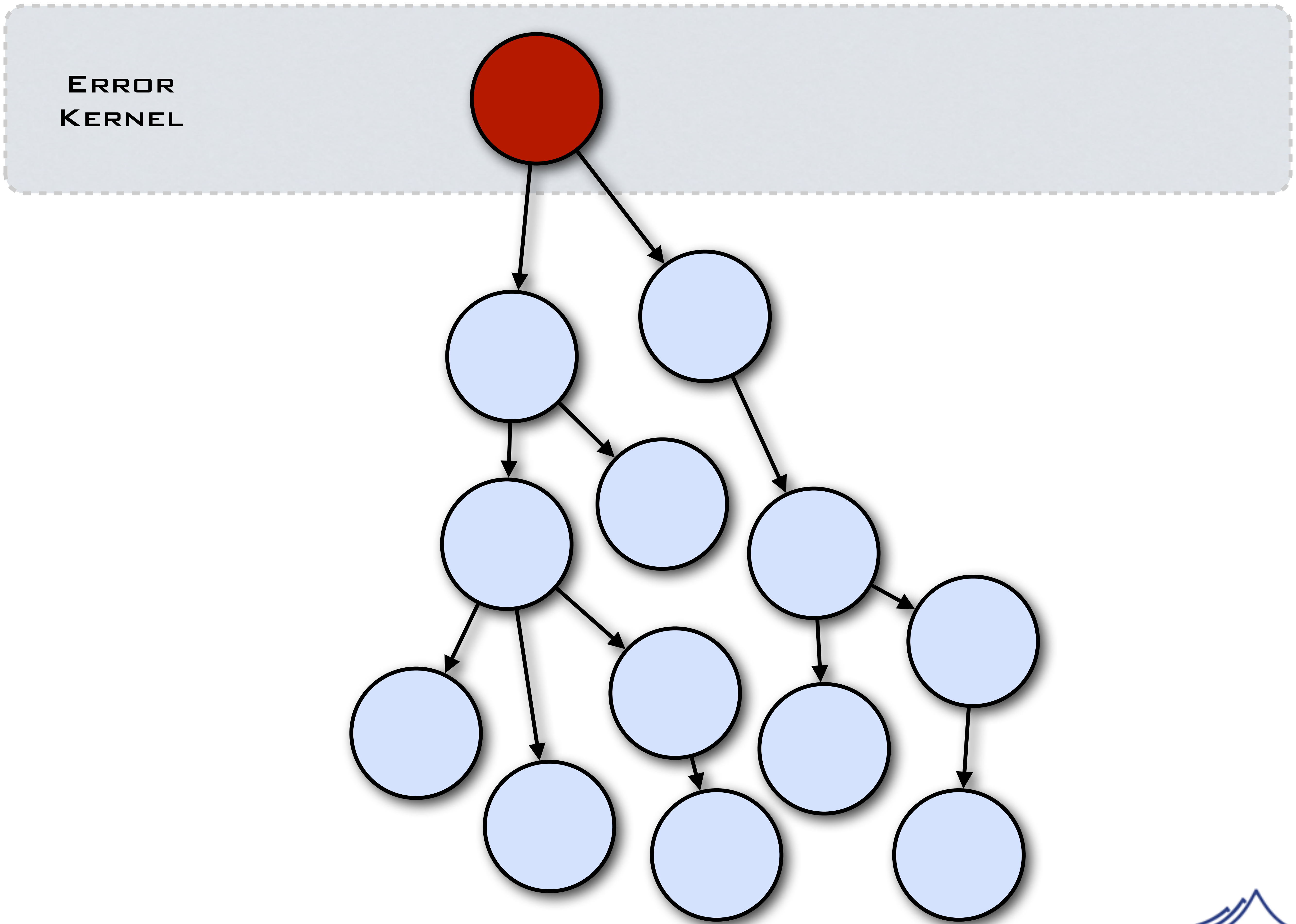

Error Kernel

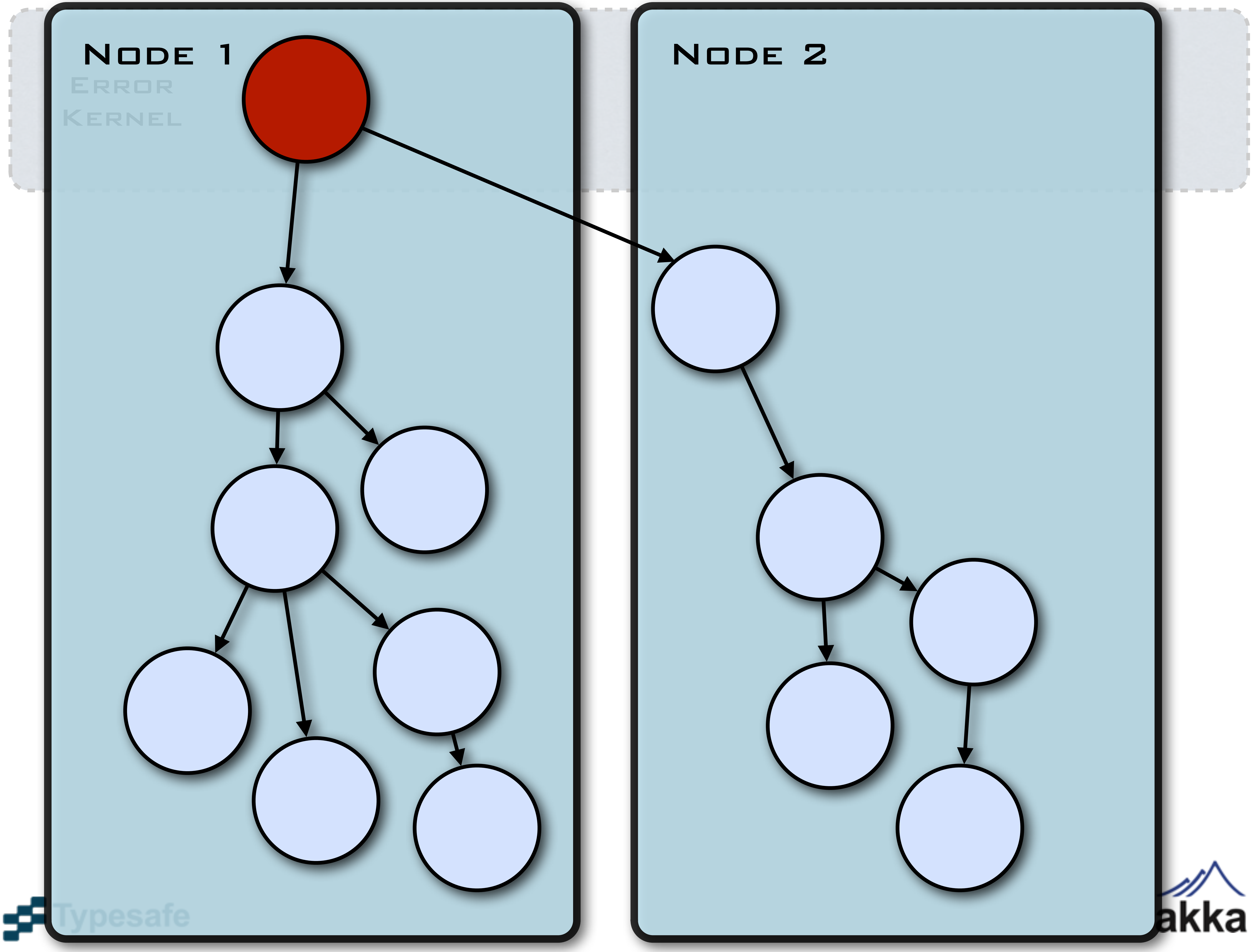


Typesafe

akka


Error
Kernel


Error
Kernel


Node 1
Error
Kernel
Node 2

SUPERVISE Actor

Every single actor has a
default supervisor strategy.
Which is usually sufficient.
But it can be overridden.

SUPERVISE Actor

Every single actor has a default supervisor strategy. Which is usually sufficient. But it can be overridden.

```
class Supervisor extends Actor {
  override val supervisorStrategy =
                   (maxNrOfRetries = 10, withinTimeRange = 1 minute) {
      case _: ArithmeticException  => Resume
      case _: NullPointerException => Restart
      case _: Exception            => Escalate
  }
```

SUPERVISE Actor

```
class Supervisor extends Actor {
  override val supervisorStrategy =
                     (maxNrOfRetries = 10, withinTimeRange = 1 minute) {
      case _: ArithmeticException  => Resume
      case _: NullPointerException => Restart
      case _: Exception            => Escalate
  }

  val worker = context.actorOf(Props[Worker])

  def receive = {
    case n: Int => worker forward n
  }
}
```

Manage failure

```
class Worker extends Actor {
  ...

  override def preRestart(
    reason: Throwable, message: Option[Any]) {
    ... // clean up before restart
  }

  override def postRestart(reason: Throwable) {
    ... // init after restart
  }
}
```

Remoting

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote =
      }
    }
  }
}
```

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote =
      }
    }
  }
}
```

Configure a Remote Provider

Remote deployment

Just feed the ActorSystem with this configuration

Configure a Remote Provider

For the Greeter actor

```
akka {
         {
    ider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote =
      }
    }
  }
}
```

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
        {
    ider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote =
      }
    }

}
```

Configure a Remote Provider

For the Greeter actor

Define Remote Path

Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
        {
  p    ider = akka.remote.RemoteActorRefProvider
    dep oyment {
      /greeter {
        remote = akka://
      }
    }

}
```

Configure a Remote Provider

For the Greeter actor

Define Remote Path

Protocol

Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
        {
    ider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote = akka://MySystem
      }
    }
}
```

Configure a Remote Provider

For the Greeter actor

Define Remote Path

Protocol

Actor System

Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
         {
     ider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote = akka://MySystem@machine1
      }
    }
  }
}
```

Configure a Remote Provider

For the Greeter actor

Define Remote Path

Protocol

Actor System

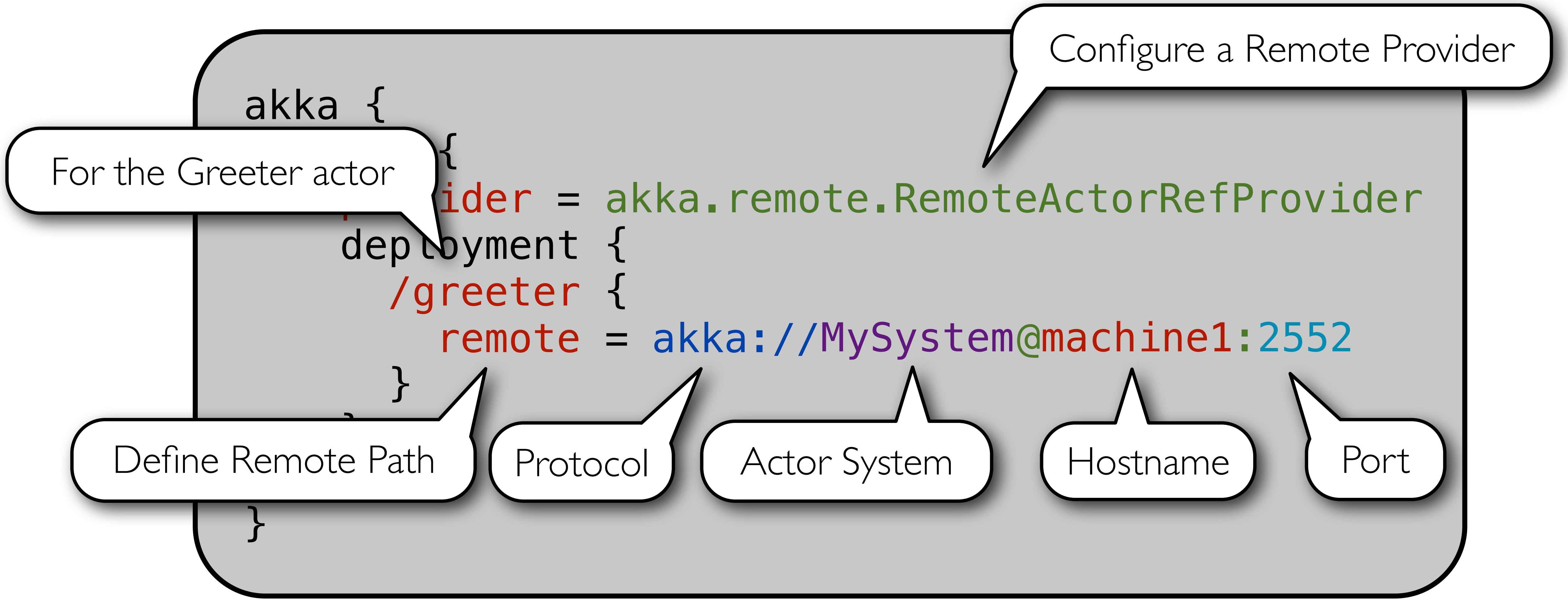
Hostname

Typesafe

akka

Remote deployment

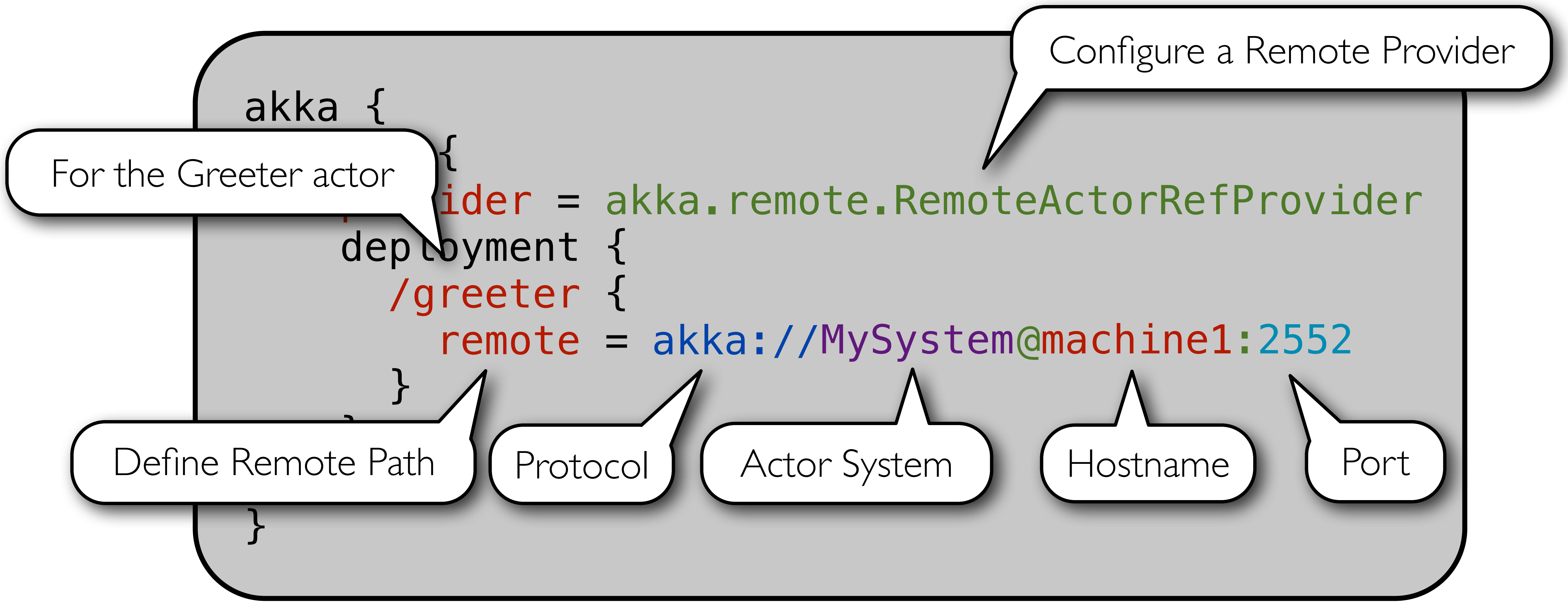
Just feed the ActorSystem with this configuration



```
akka {
         {
    ider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote = akka://MySystem@machine1:2552
      }

}
```

Remote deployment

Just feed the ActorSystem with this configuration



```
akka {
        {
        ider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote = akka://MySystem@machine1:2552
      }


}
```

Zero code changes

Remote Lookup

```scala
val greeter = system.actorFor(
    "akka://MySystem@machine1:2552/user/greeter")
```

Can you see the **problem**?

Fixed Addresses

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
         remote = akka://MySystem@machine1:2552
      }
    }
  }
}
```

```
val greeter = system.actorFor(
    "akka://MySystem@machine1:2552/user/greeter")
```

Akka Cluster

Features

- Gossip-based Cluster Membership
- Leader determination
- Accrual Failure Detector
- Cluster DeathWatch
- Cluster-Aware Routers

Enable clustering

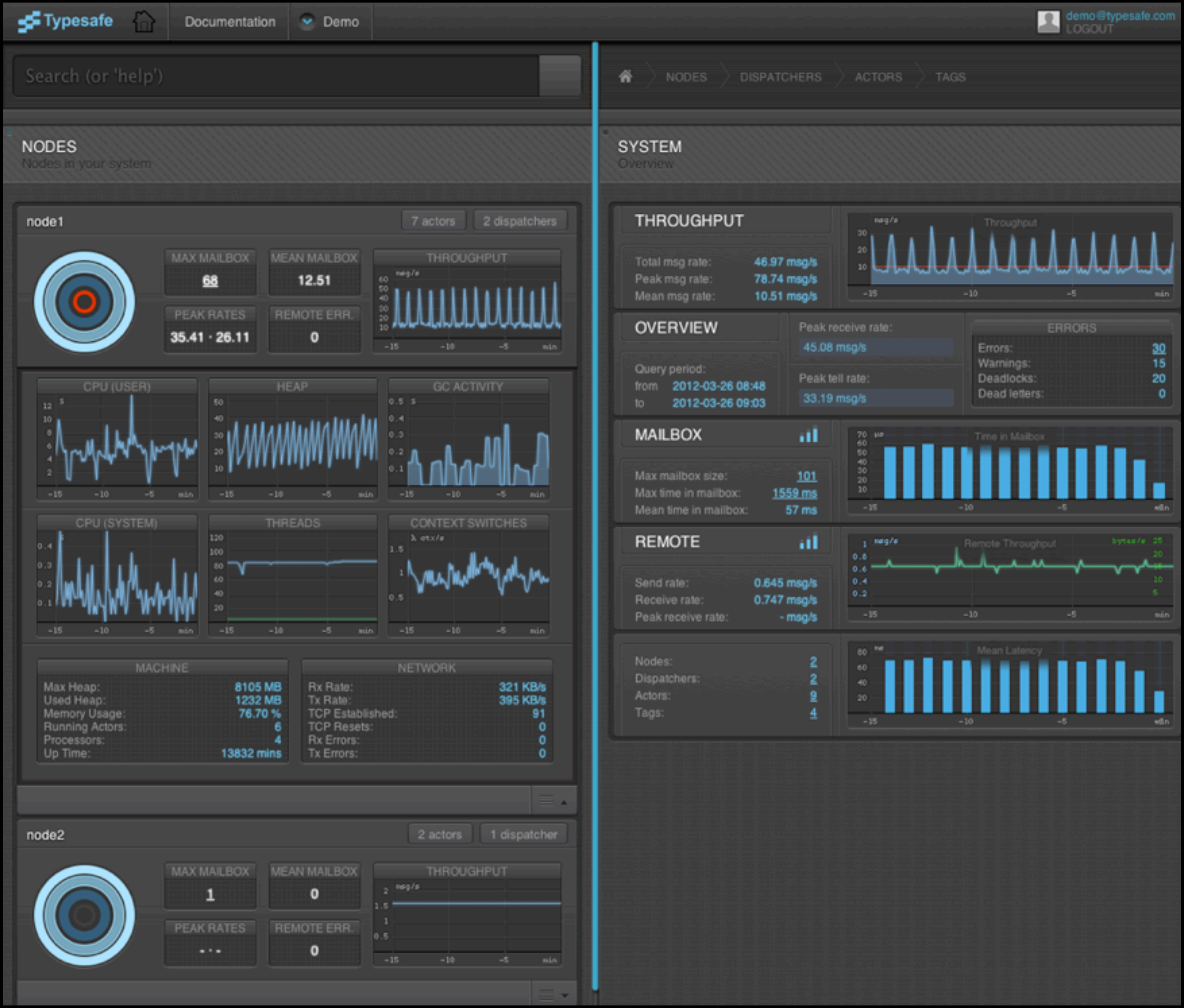
```
akka {
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
    ...
  }

  extensions = ["akka.cluster.Cluster"]

  cluster {
    seed-nodes = [
      "akka://ClusterSystem@127.0.0.1:2551",
      "akka://ClusterSystem@127.0.0.1:2552"
    ]

    auto-down = on
  }
}
```

Configure a clustered router

```
akka.actor.deployment {
  /statsService/workerRouter {
    router = consistent-hashing
    nr-of-instances = 100

    cluster {
      enabled = on
      max-nr-of-instances-per-node = 3
      allow-local-routees = on
    }
  }
}
```

Typesafe **Console**

free for developers later in the fall



NODES
Nodes in your system

node1 — 7 actors — 2 dispatchers

MAX MAILBOX	MEAN MAILBOX
68	12.51
PEAK RATES	**REMOTE ERR.**
35.41 · 26.11	0

MACHINE

Max Heap:	8105 MB
Used Heap:	1232 MB
Memory Usage:	76.70 %
Running Actors:	6
Processors:	4
Up Time:	13832 mins

NETWORK

Rx Rate:	321 KB/s
Tx Rate:	395 KB/s
TCP Established:	91
TCP Resets:	0
Rx Errors:	0
Tx Errors:	0

node2 — 2 actors — 1 dispatcher

MAX MAILBOX	MEAN MAILBOX
1	0
PEAK RATES	**REMOTE ERR.**
- · -	0

SYSTEM
Overview

THROUGHPUT

Total msg rate:	46.97 msg/s
Peak msg rate:	78.74 msg/s
Mean msg rate:	10.51 msg/s

OVERVIEW

Query period:
from 2012-03-26 08:48
to 2012-03-26 09:03

Peak receive rate: 45.08 msg/s
Peak tell rate: 33.19 msg/s

ERRORS

Errors:	30
Warnings:	15
Deadlocks:	20
Dead letters:	0

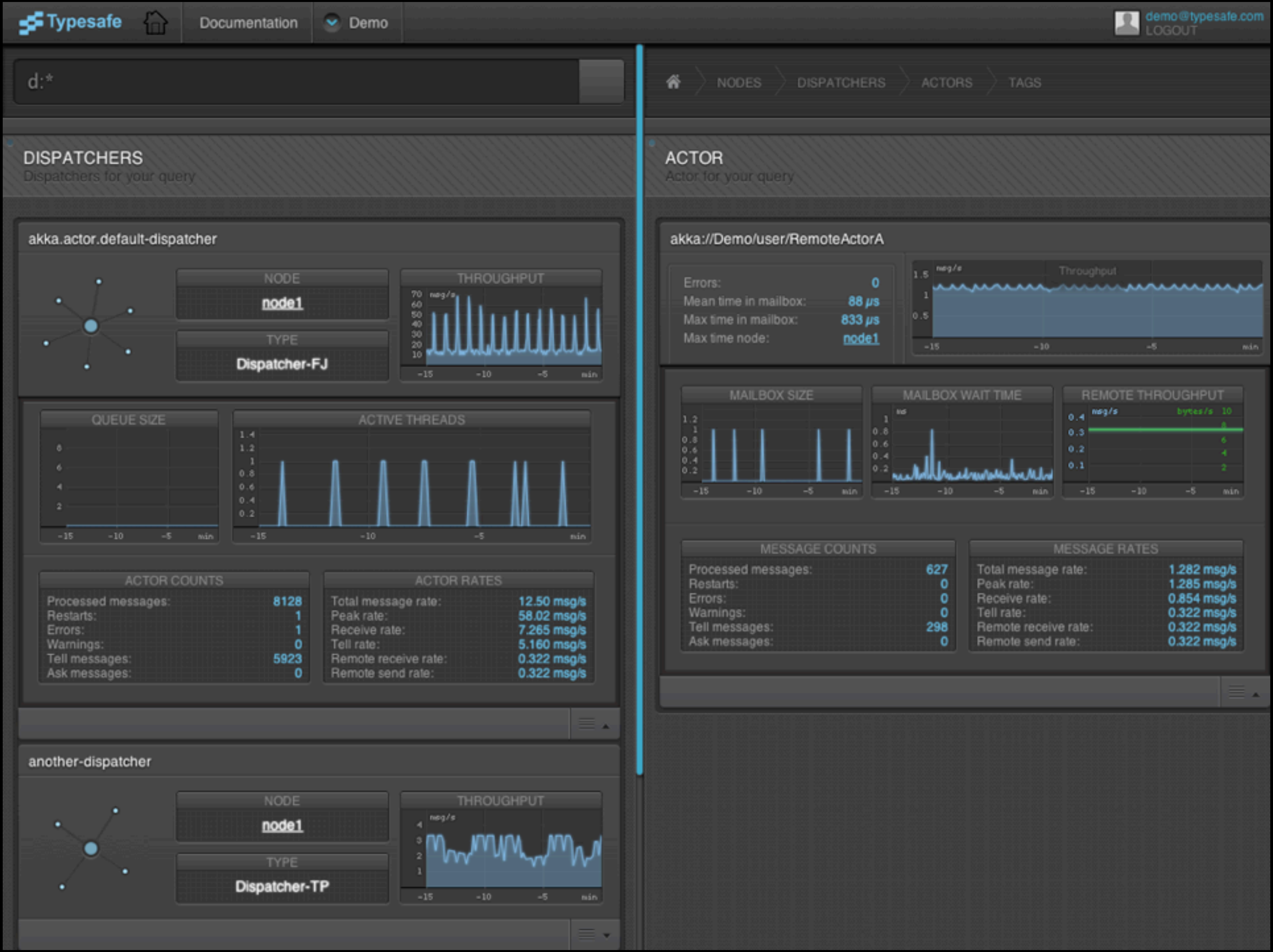
MAILBOX

Max mailbox size:	101
Max time in mailbox:	1559 ms
Mean time in mailbox:	57 ms

REMOTE

Send rate:	0.645 msg/s
Receive rate:	0.747 msg/s
Peak receive rate:	- msg/s

Nodes:	2
Dispatchers:	2
Actors:	9
Tags:	4



DISPATCHERS
Dispatchers for your query

akka.actor.default-dispatcher

NODE	node1
TYPE	Dispatcher-FJ

ACTOR COUNTS

Processed messages:	8128
Restarts:	1
Errors:	1
Warnings:	0
Tell messages:	5923
Ask messages:	0

ACTOR RATES

Total message rate:	12.50 msg/s
Peak rate:	58.02 msg/s
Receive rate:	7.265 msg/s
Tell rate:	5.160 msg/s
Remote receive rate:	0.322 msg/s
Remote send rate:	0.322 msg/s

another-dispatcher

NODE	node1
TYPE	Dispatcher-TP

ACTOR
Actor for your query

akka://Demo/user/RemoteActorA

Errors:	0
Mean time in mailbox:	88 µs
Max time in mailbox:	833 µs
Max time node:	node1

MESSAGE COUNTS

Processed messages:	627
Restarts:	0
Errors:	0
Warnings:	0
Tell messages:	298
Ask messages:	0

MESSAGE RATES

Total message rate:	1.282 msg/s
Peak rate:	1.285 msg/s
Receive rate:	0.854 msg/s
Tell rate:	0.322 msg/s
Remote receive rate:	0.322 msg/s
Remote send rate:	0.322 msg/s

live demo

http://console-demo.typesafe.com

...we have much **much** more

...we have much **much** more

EventBus

TestKit

FSM

Pub/Sub

Durable Mailboxes

IO

Camel

Microkernel

SLF4J

TypedActor

ZeroMQ

Dataflow

Transactors

Agents

Extensions

get it and learn more

http://akka.io

http://letitcrash.com

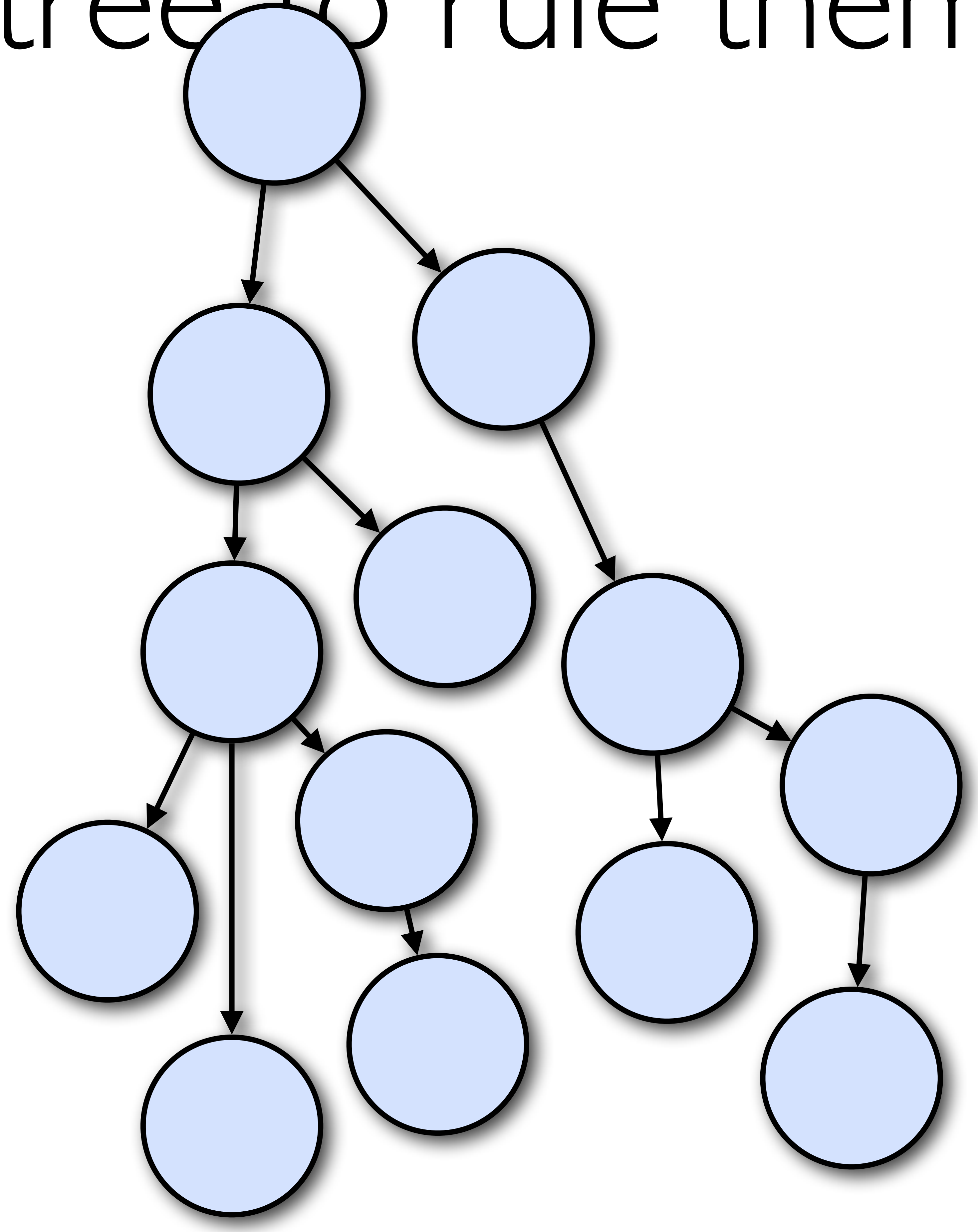
http://typesafe.com

E0F

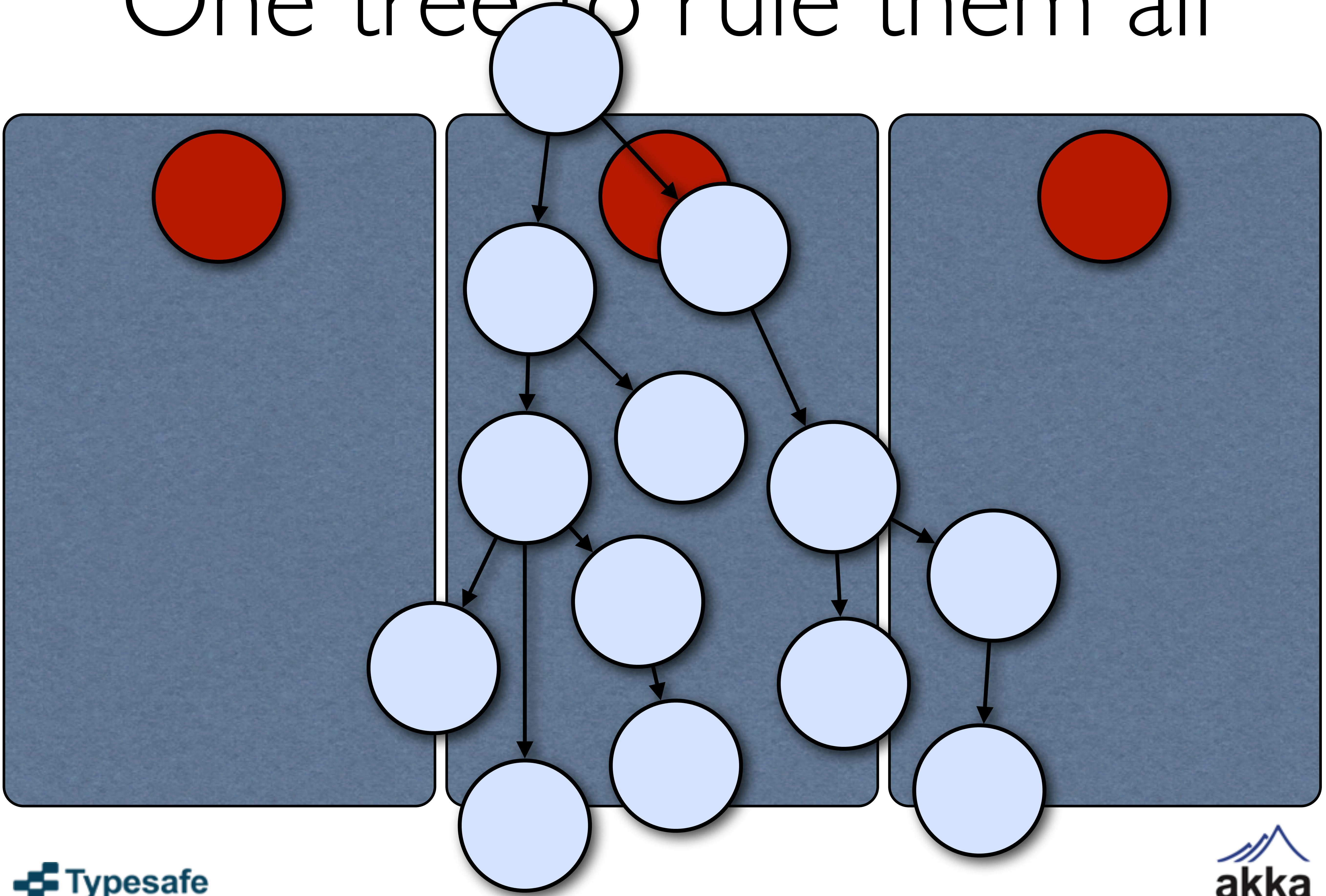
Akka Cluster

Upcoming features

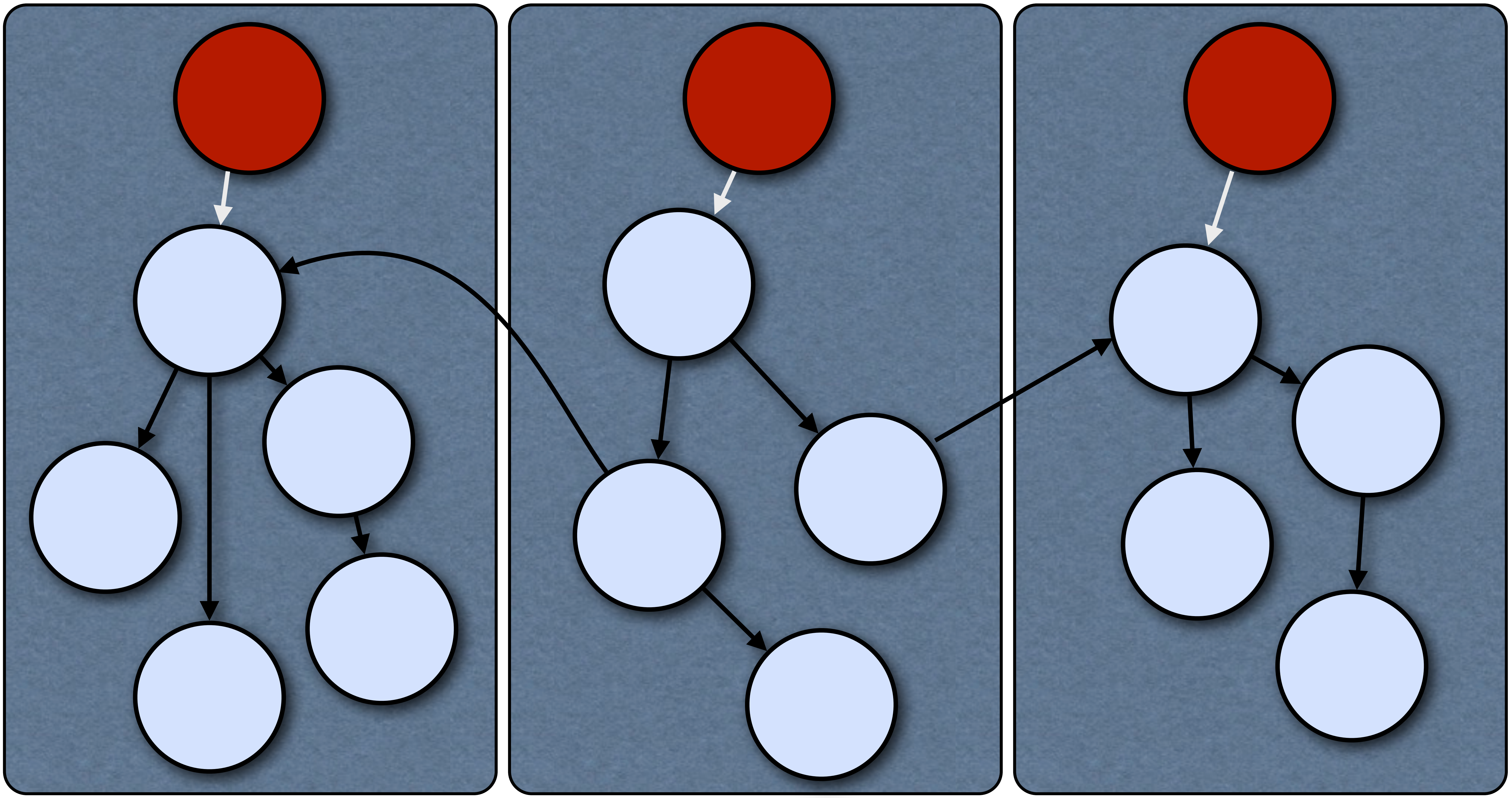
One tree to rule them all



One tree to rule them all



One tree to rule them all



Typesafe

akka

The Magic Sauce

- User code only sees `cluster://...` names
- ActorRef becomes repointable
  - local (current ActorCell)
  - remote (new RemoteActorCell)
- Can now move actors around transparently
  - Actor encapsulation makes it possible

What does this enable?

- Actor migration
- Actor replication
- Automatic cluster partitioning
  - later also based on runtime metrics
- Node fail-over
  - first for stateless actors
  - later for stateful actors using event sourcing

⇛ *Fault Tolerance & Distribution*