Scaling software with

Jonas Bonér

CTO Typesafe

@jboner

Scaling
software with

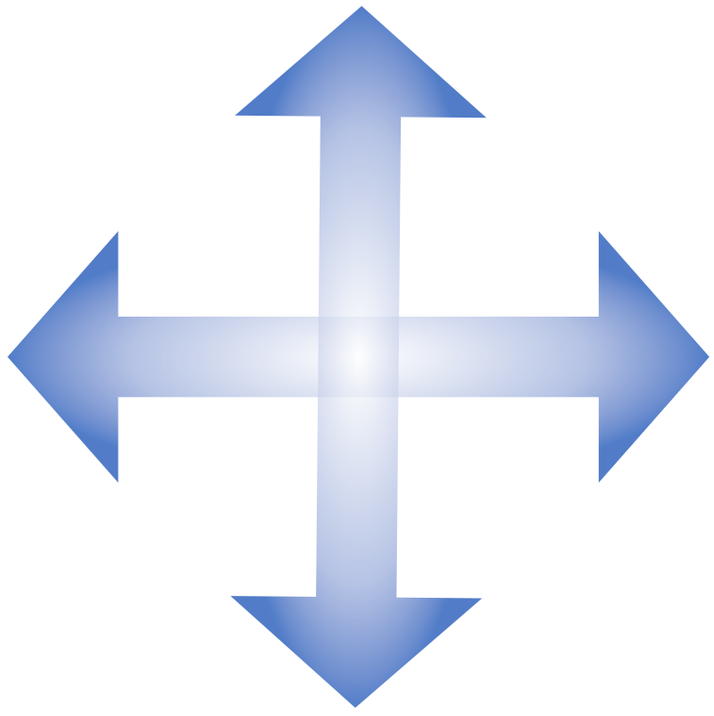
Leverage what you already know

All this is working with the Java investment; software, skills, tools and techniques, you have today:

- USE it from JAVA today
- ...with the option of using leveraging SCALA LATER (if the urge comes up)
- INTEGRATE with, and deploy into, your current infrastructure/environment

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
UNIBET
DRW TRADING GROUP
OOYALA
novus
navirec
T8 Webware
CARTOMAPIC
Moshi Monsters
BBC
Maritime Poker
banksimple
zeebox The best thing to happen to TV since TV
Answers.com The world's leading Q&A site
azavea



Typesafe

akka

Manage System Overload

Typesafe

akka

Automatic Replication & Distribution

for Fault-tolerance & Scalability

Typesafe

akka

Program at a Higher Level

Program at a Higher Level

Typesafe

akka

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

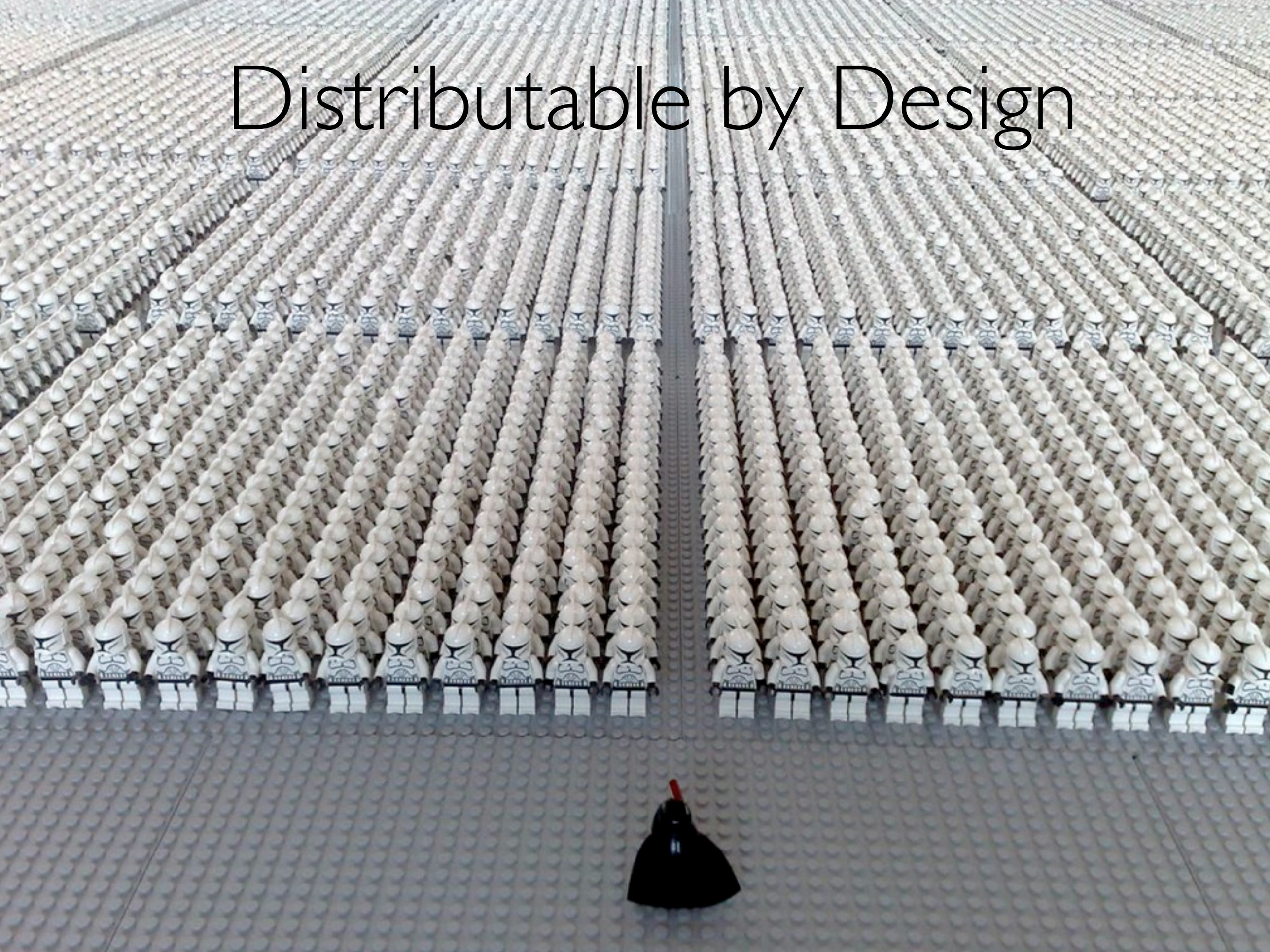
Program at a Higher Level

- 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

Distributable by Design



Distributable by Design

Typesafe

akka

Distributable by Design

- Actors are location transparent & distributable by design

Typesafe

akka

Distributable by Design

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

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

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

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

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?

Typesafe

akka

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

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}
```

0. DEFINE

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

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}
```

0. DEFINE

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

Define the Actor class

```
public class Greeting implements Serializable {
    public final String who;
    public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
    }
}
```

Typesafe

akka

0. DEFINE

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

```
public class Greeting implements Serializable {
    public final String who;
    public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
          log.info("Hello " + ((Greeting) message).who);
      }
    }
}
```

Define the Actor class

Define the Actor's behavior

Typesafe

akka

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

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
```

CREATE Actor

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
```

Create an Actor system

CREATE Actor

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
```

Create an Actor system

Actor configuration

CREATE Actor

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
```

Create an Actor system

Actor configuration

Give it a name

Typesafe

akka

CREATE Actor

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
```

Create an Actor system

Actor configuration

Create the Actor

Give it a name

Typesafe

akka

CREATE Actor

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
```

Create an Actor system

Actor configuration

You get an ActorRef back

Create the Actor

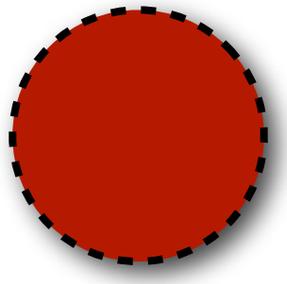
Give it a name

Typesafe

akka

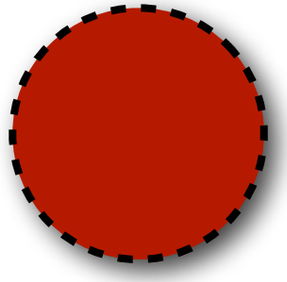
Actors can form hierarchies

Guardian System Actor



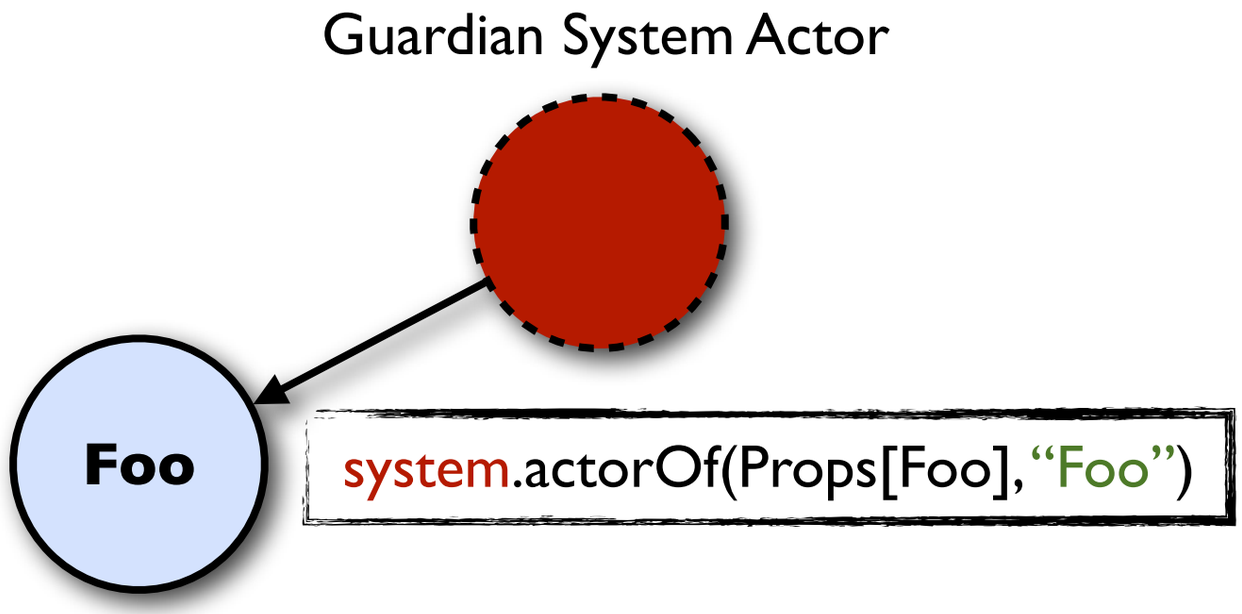
Actors can form hierarchies

Guardian System Actor

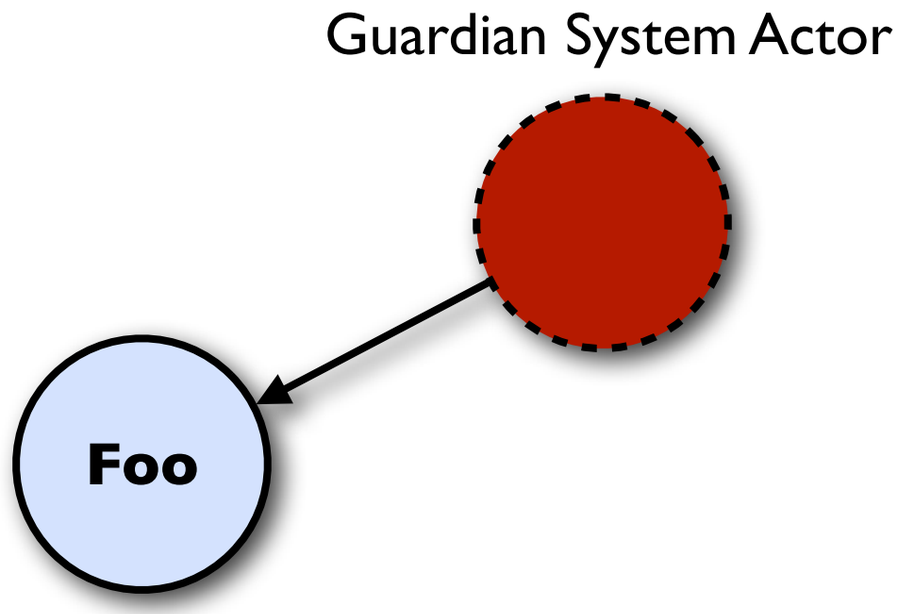


system.actorOf(Props[Foo], “Foo”)

Actors can form hierarchies

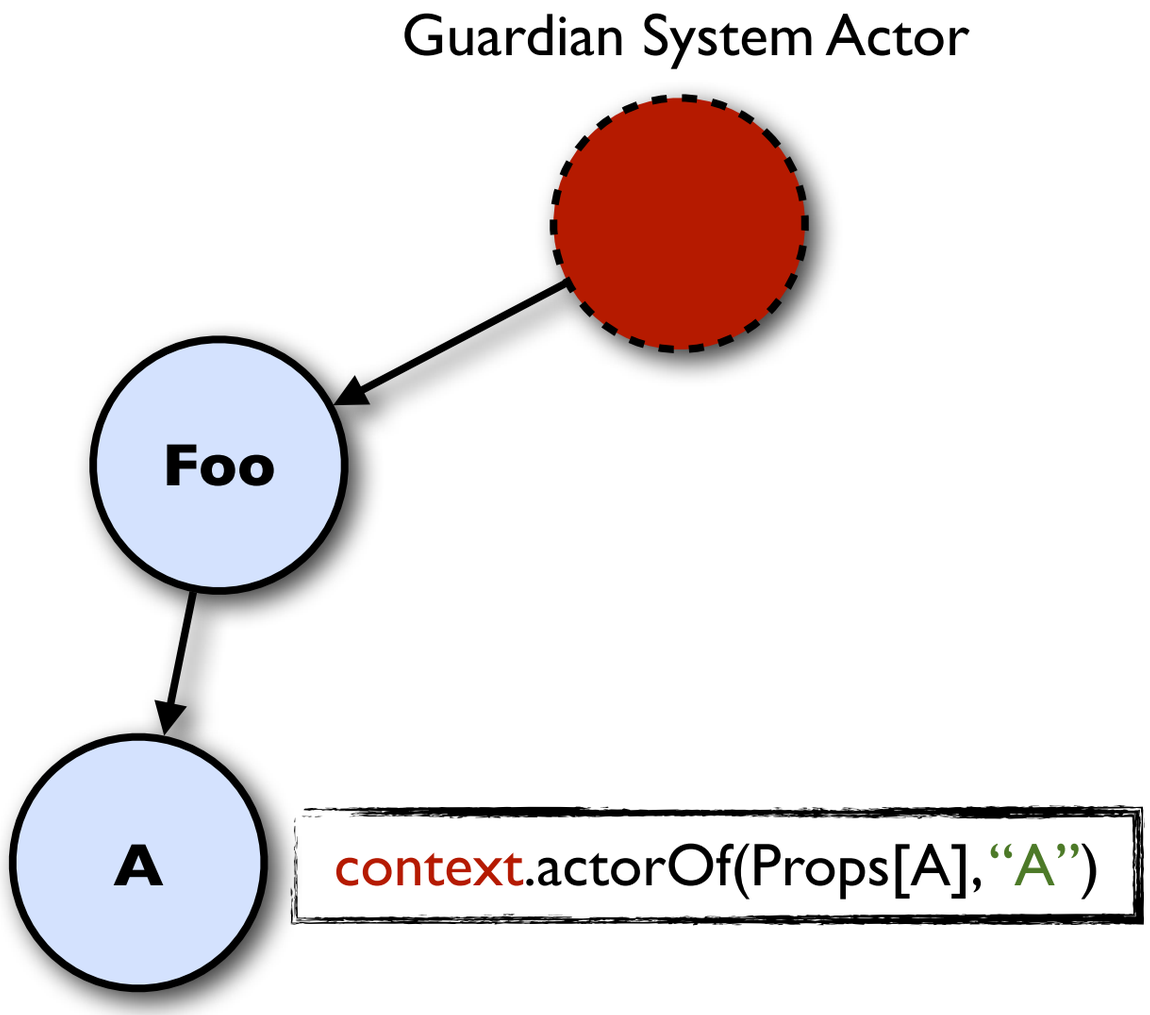


Actors can form hierarchies

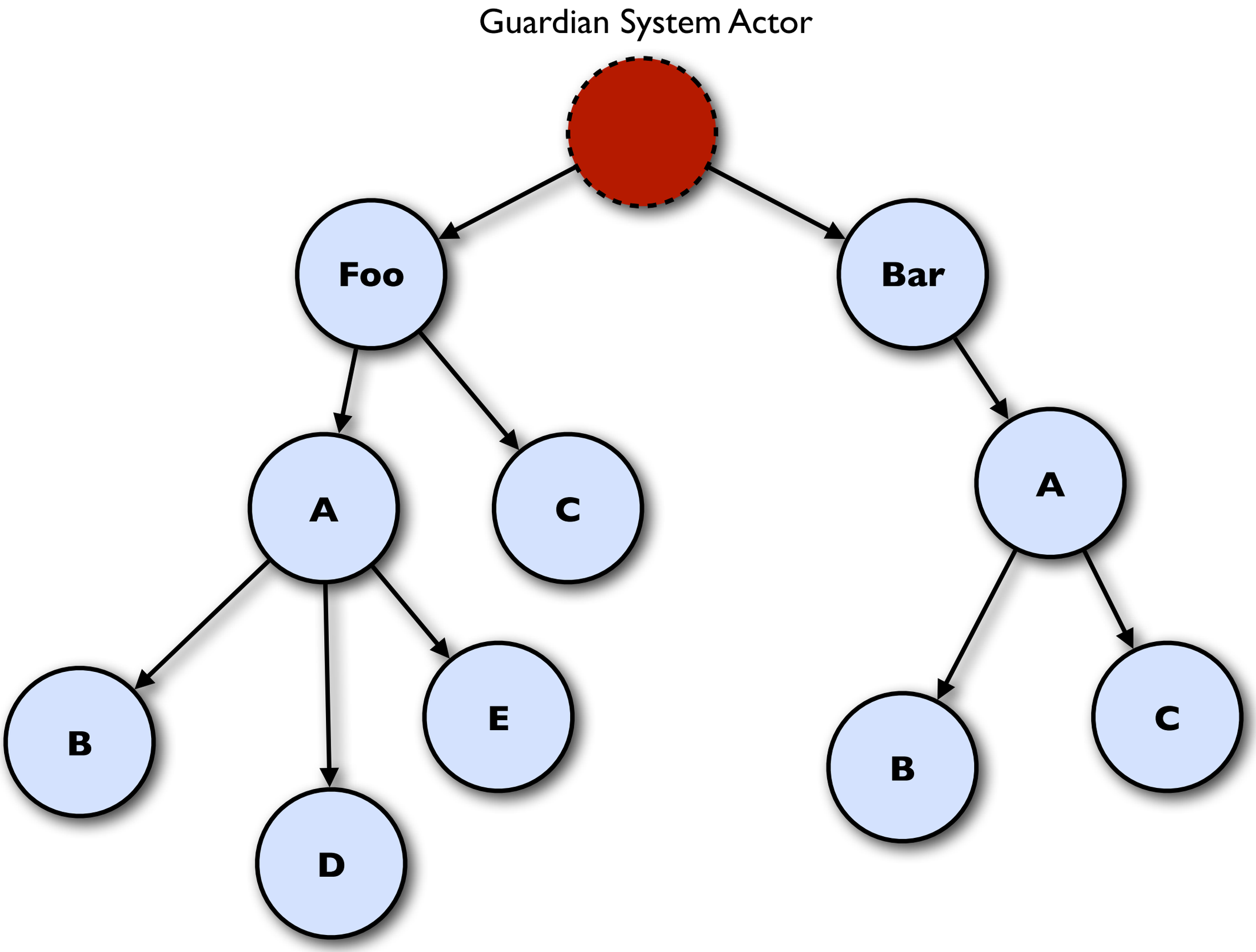



```
context.actorOf(Props[A], "A")
```

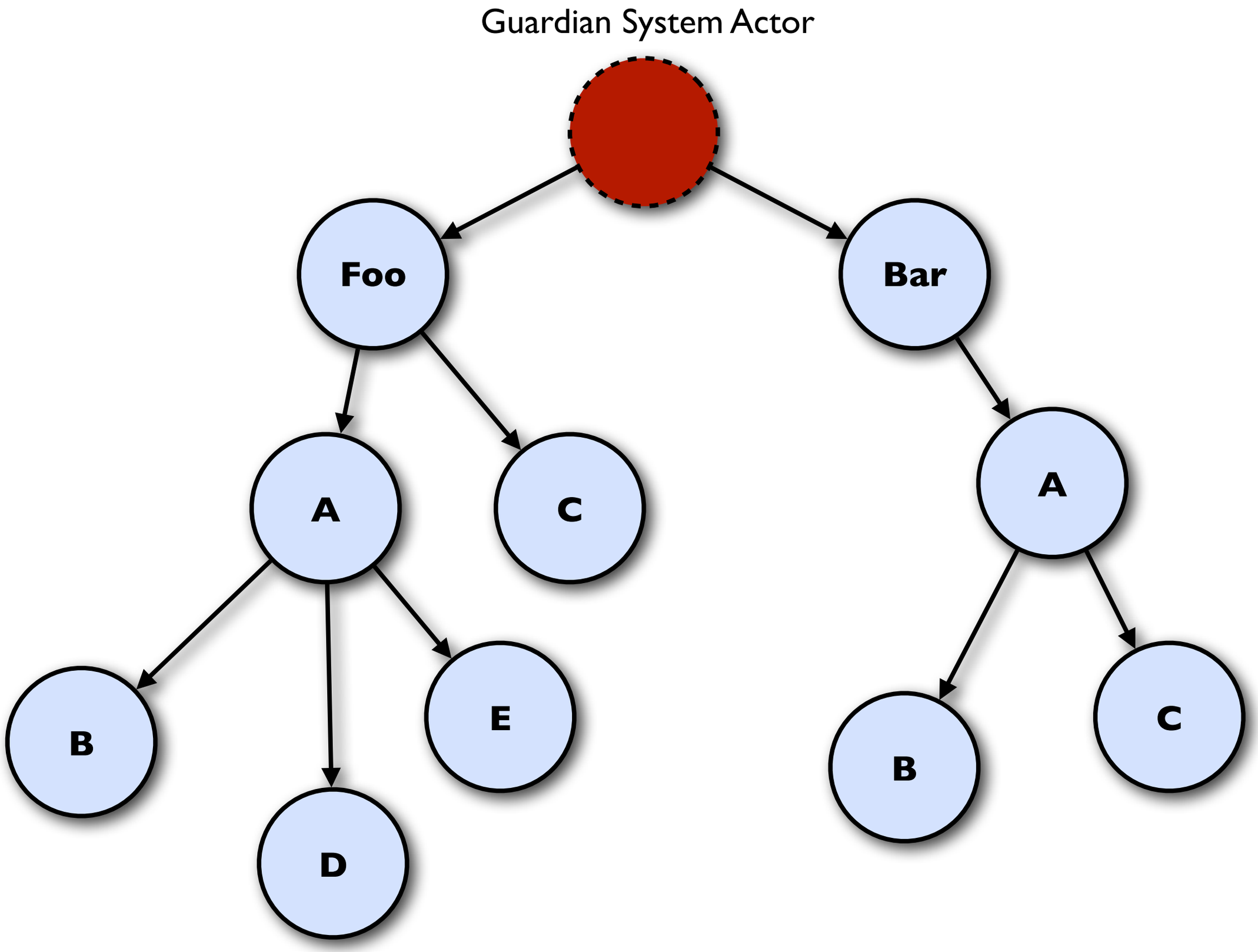
Actors can form hierarchies



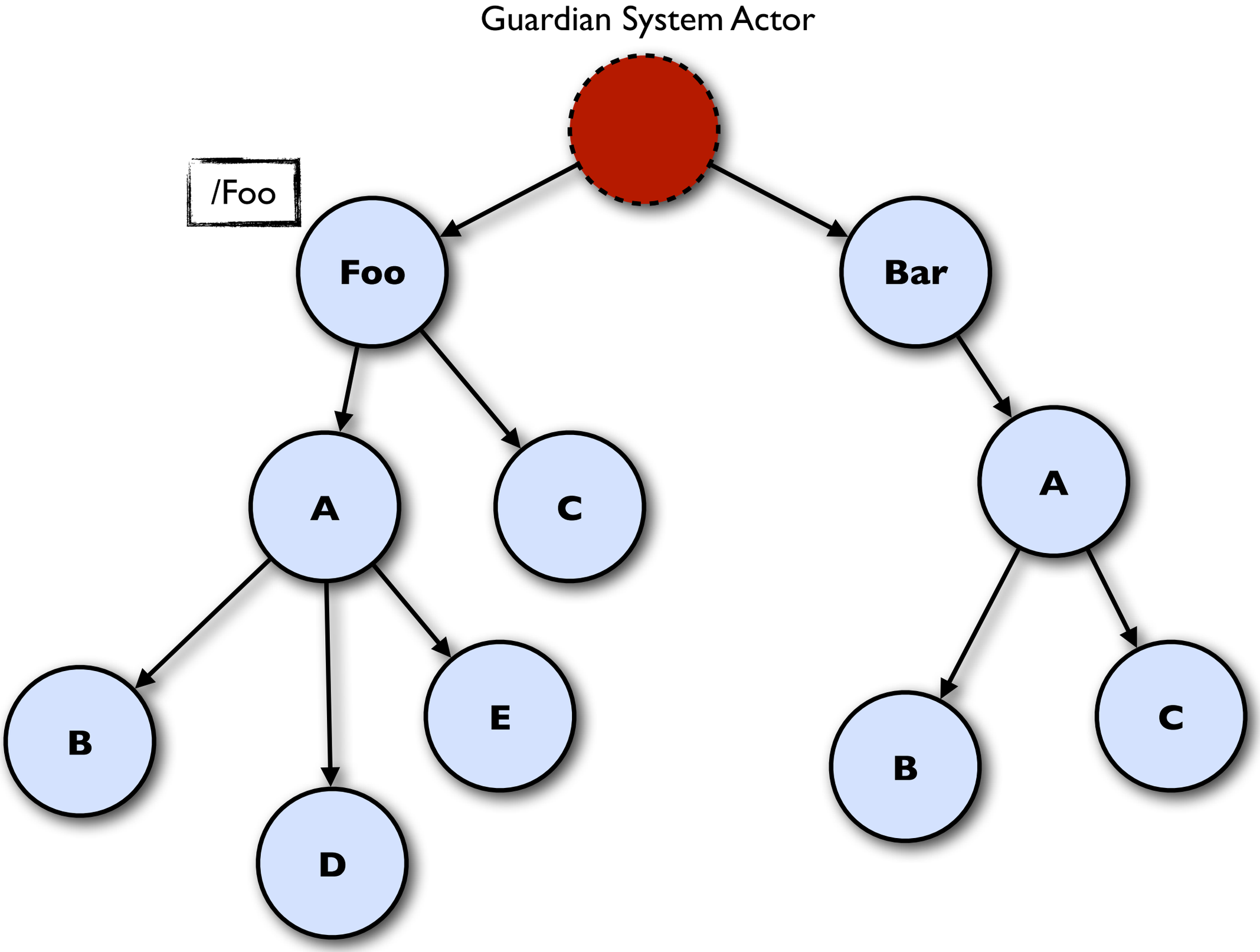
Actors can form hierarchies



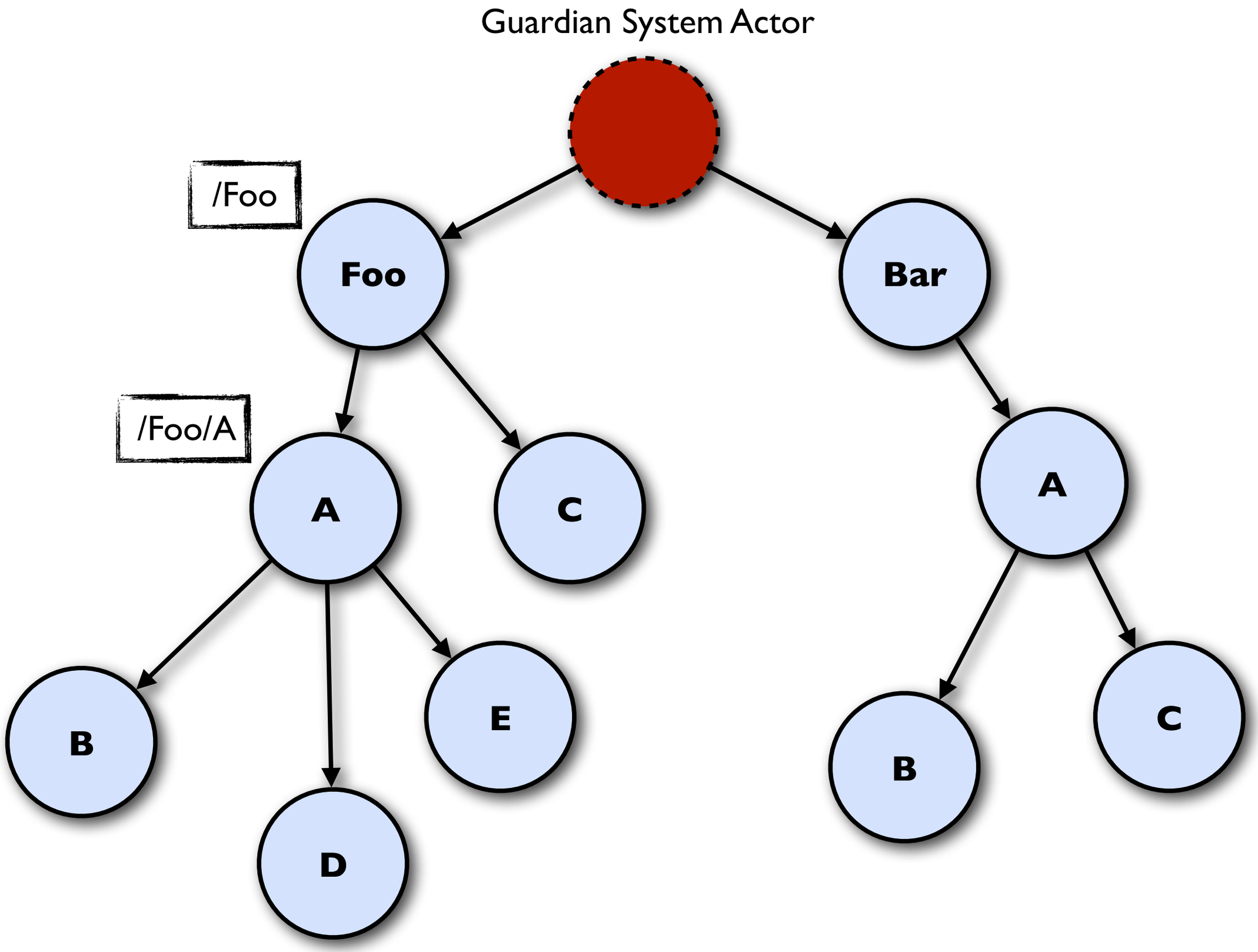
Name resolution - like a file-system



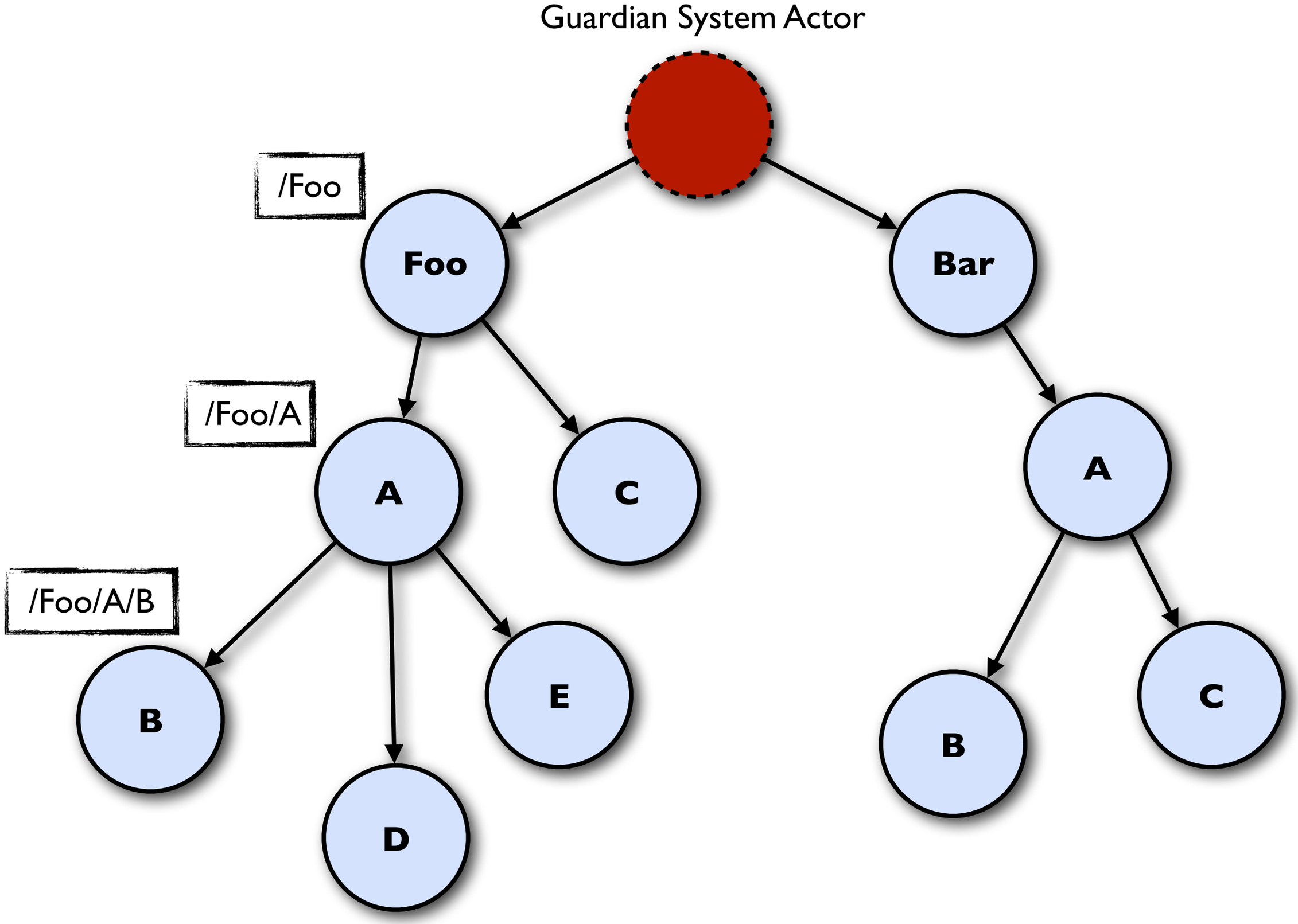
Name resolution - like a file-system



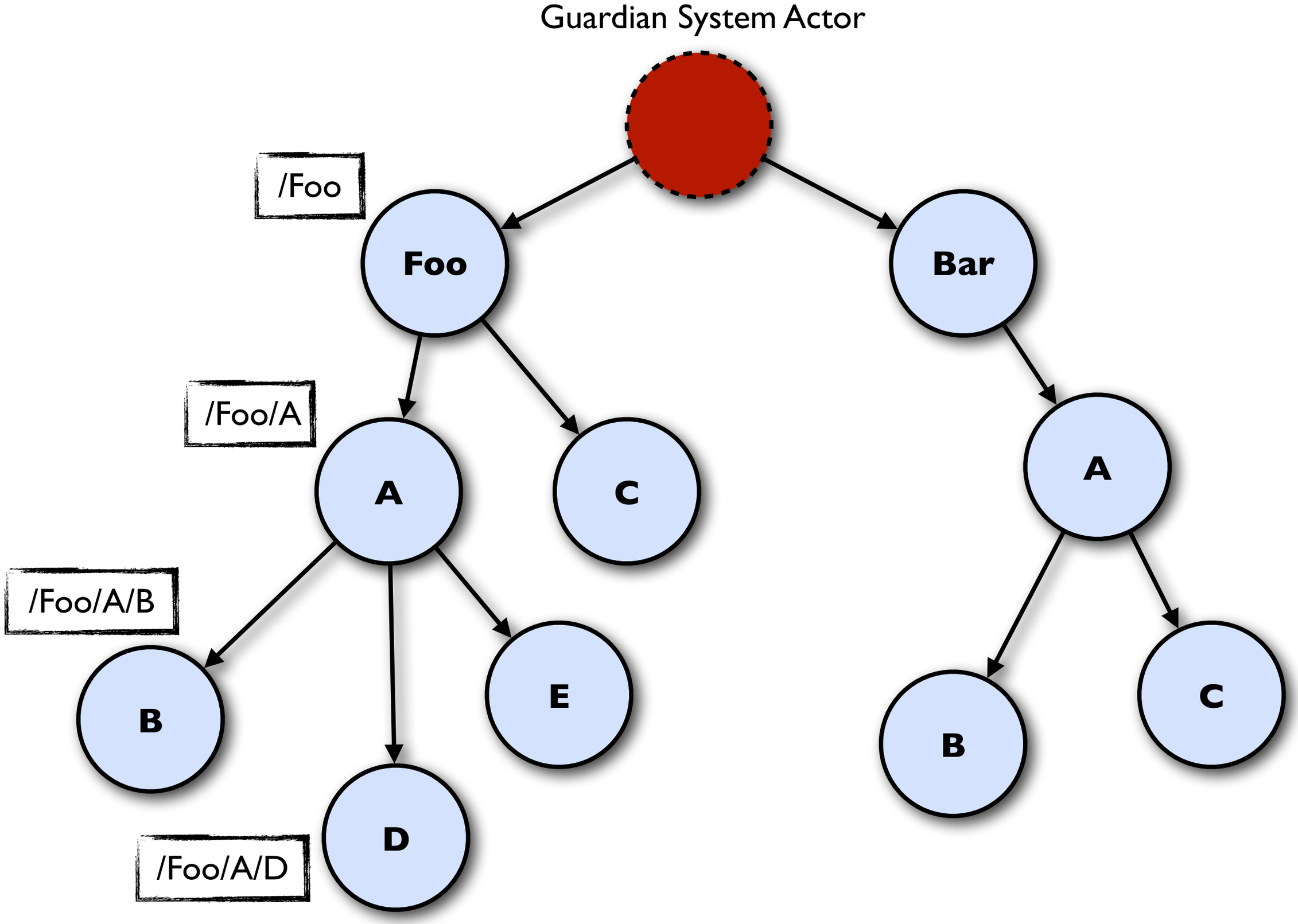
Name resolution - like a file-system



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

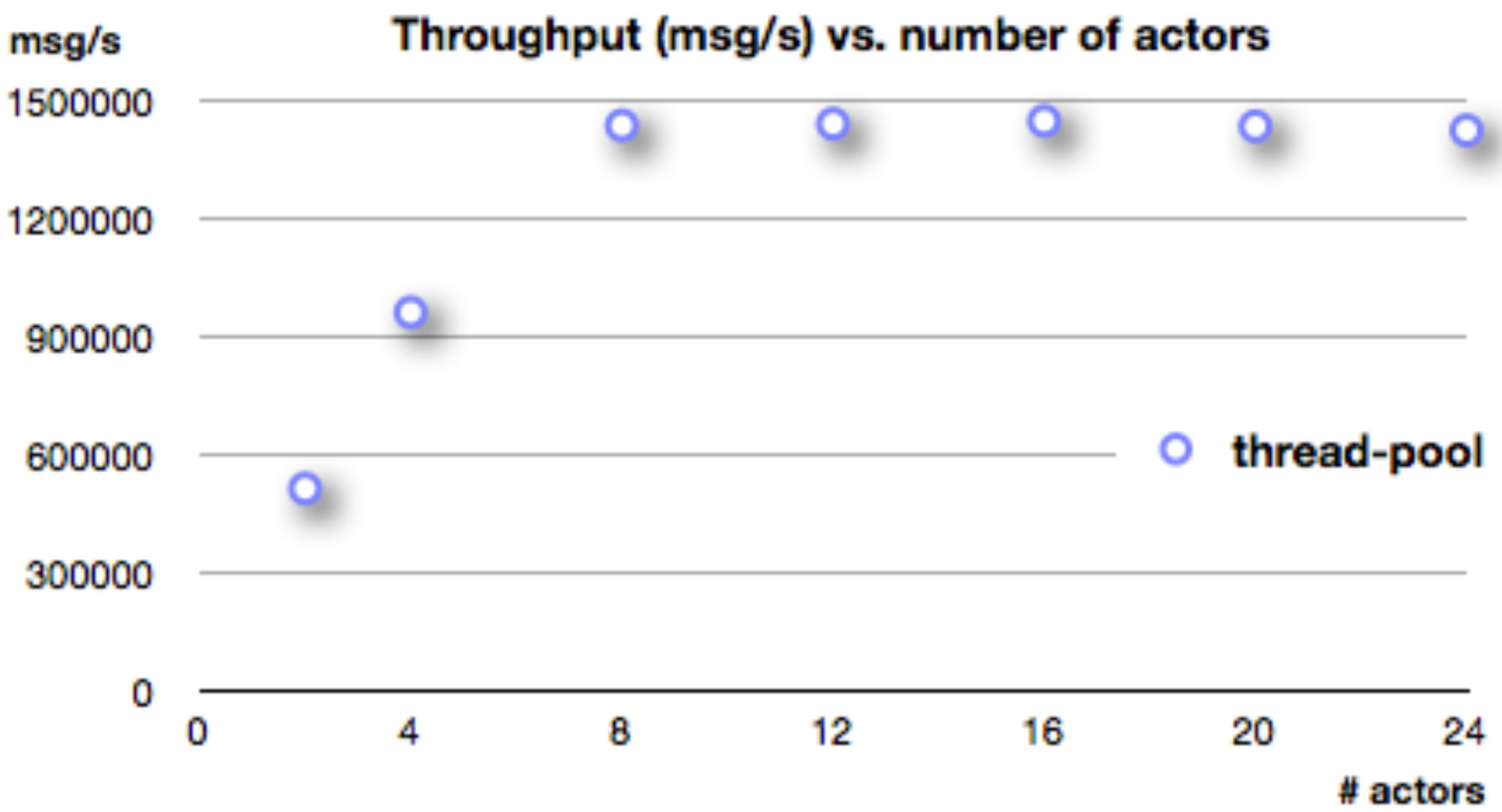
- SEND - sends a message to an Actor
- Asynchronous and Non-blocking - Fire-forget
- EVERYTHING is asynchronous and lockless
- Everything happens Reactively
  - An Actor is passive until a message is sent to it, which triggers something within the Actor
  - Messages is the Kinetic Energy in an Actor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

ThreadPoolExecutor

@ 48 cores

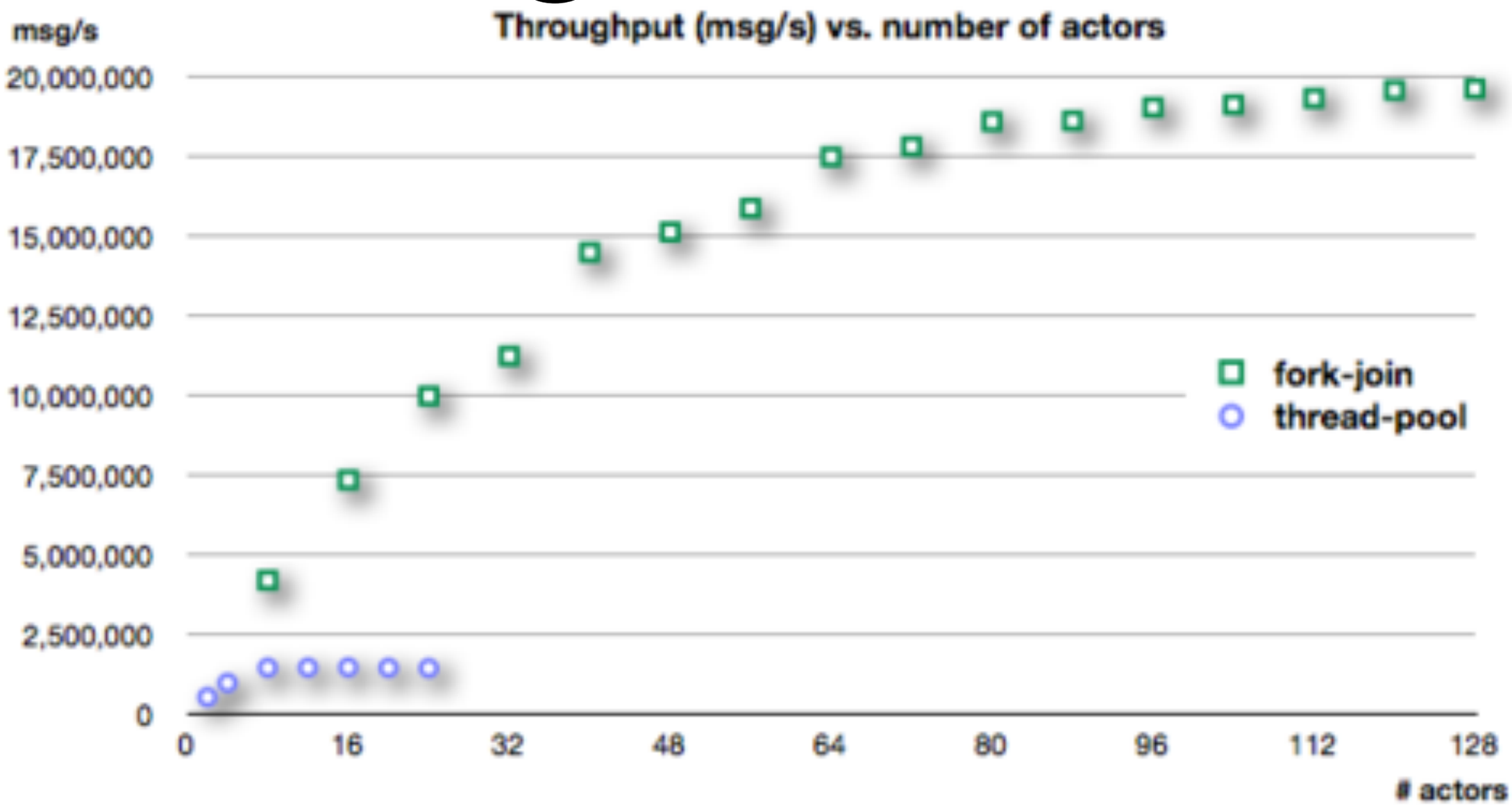


Typesafe

akka

new ForkJoinPool

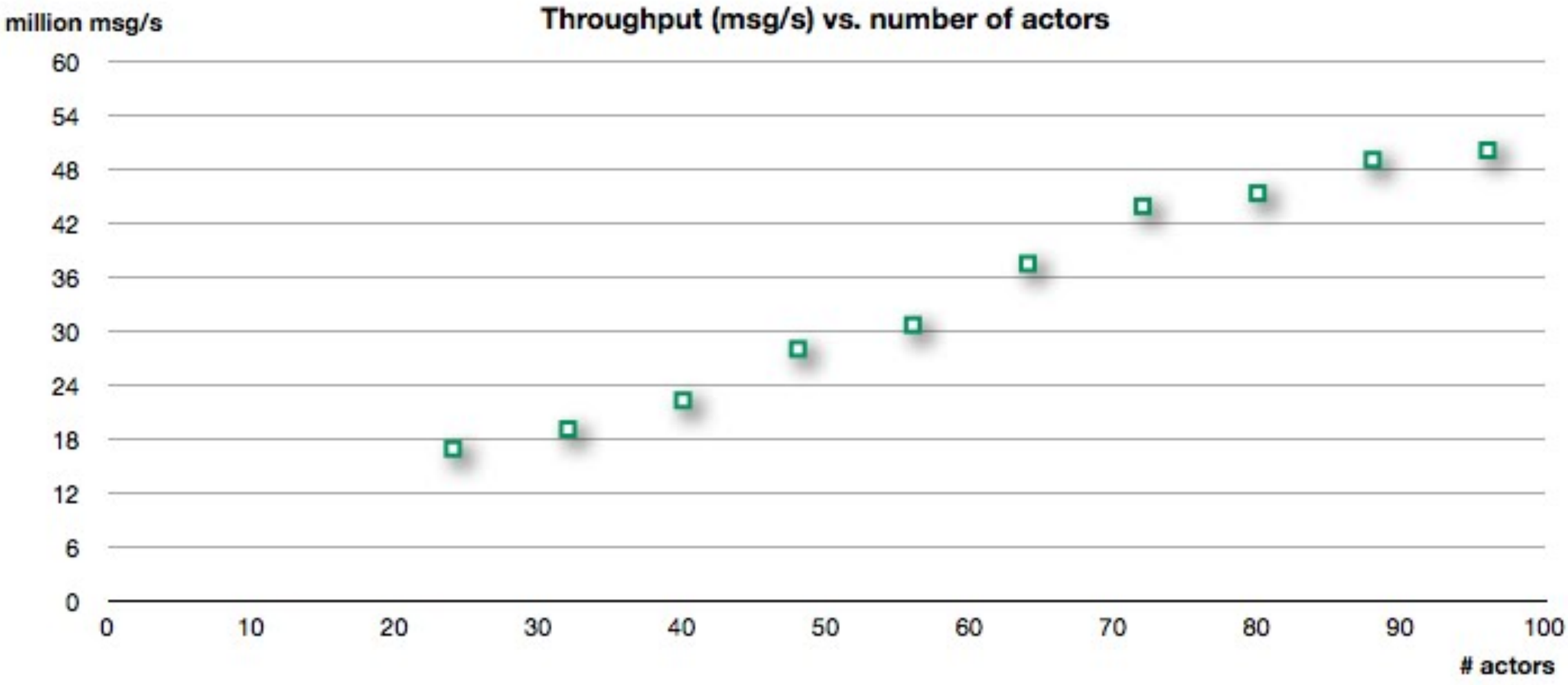
@ 48 cores



Typesafe

akka

Throughput on a single box @ 48 cores



+50 million messages per second

Typesafe

akka

SEND message

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
greeter.tell(new Greeting("Charlie Parker"));
```

SEND message

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
greeter.tell(new Greeting("Charlie Parker"));
```

Send the message

Typesafe

akka

Full example

```
public class Greeting implements Serializable {
   public final String who;
   public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) throws Exception {
      if (message instanceof Greeting)
         log.info("Hello " + ((Greeting) message).who);
      }
   }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
greeter.tell(new Greeting("Charlie Parker"));
```

Remote deployment

Just feed the ActorSystem with this configuration

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

Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

Configure a Remote Provider

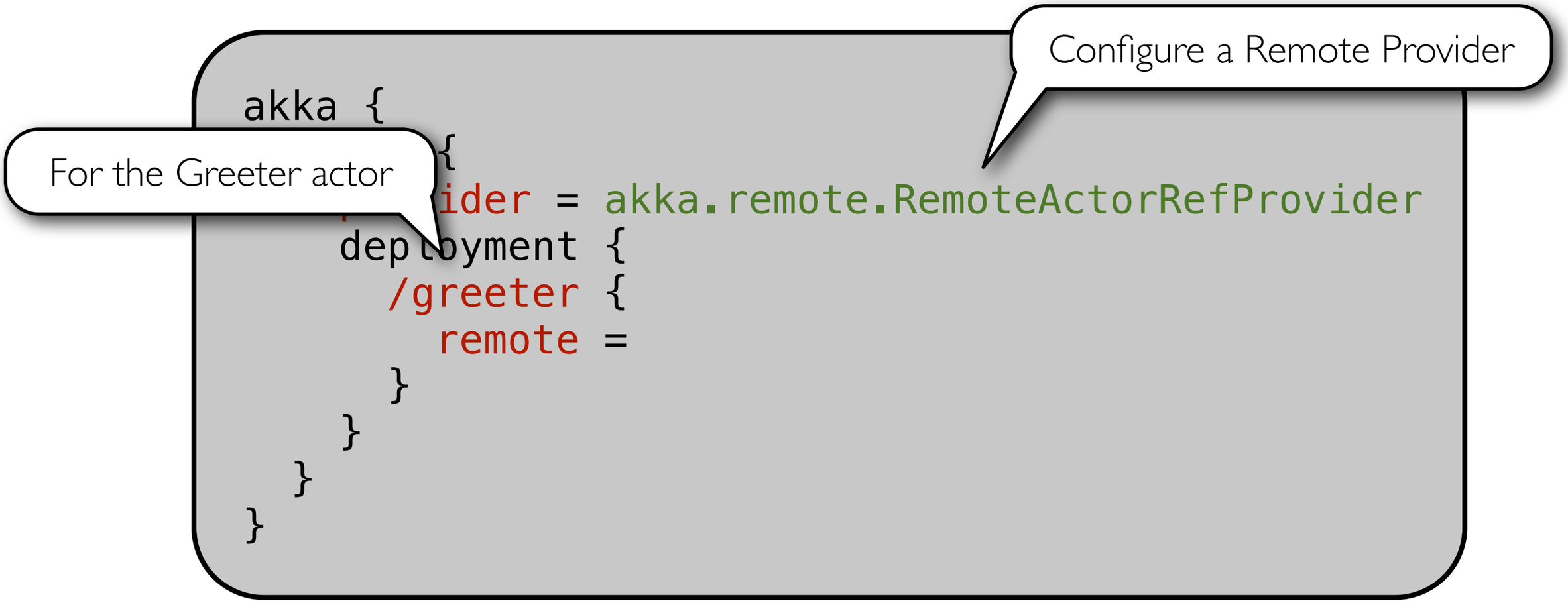
```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote =
      }
    }
  }
}
```

Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

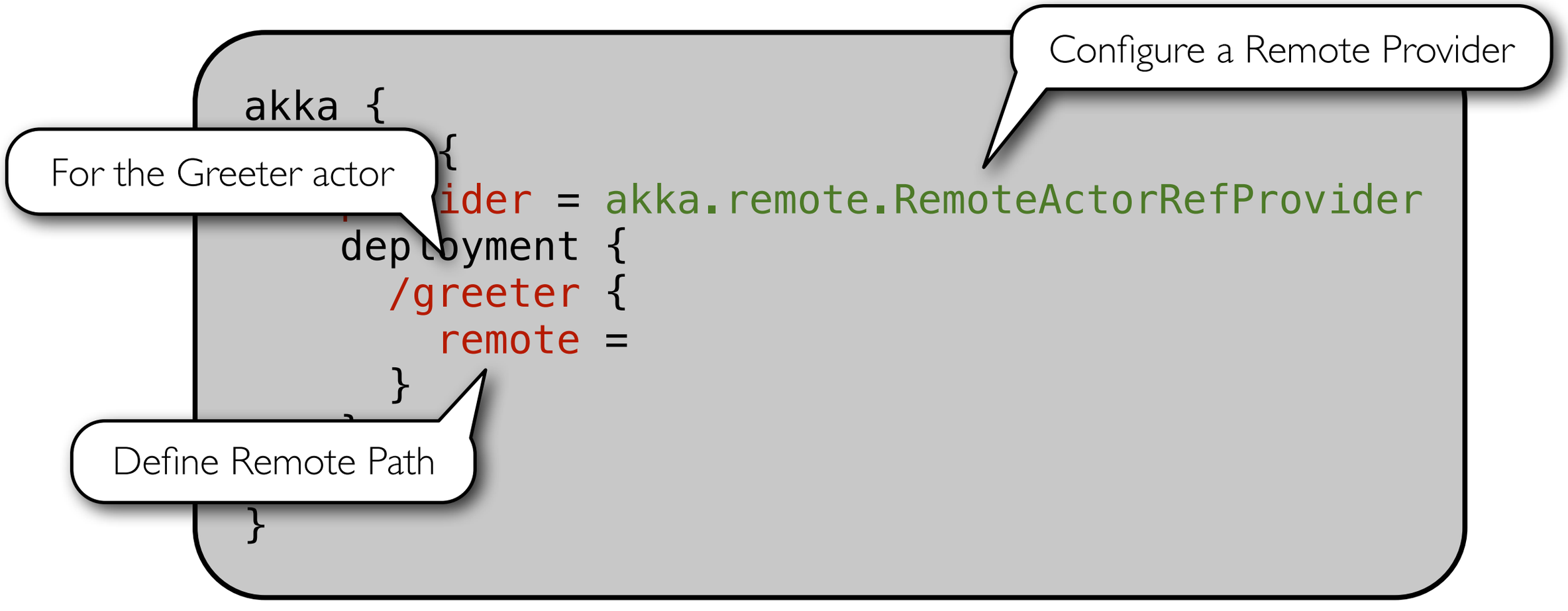


Typesafe

akka

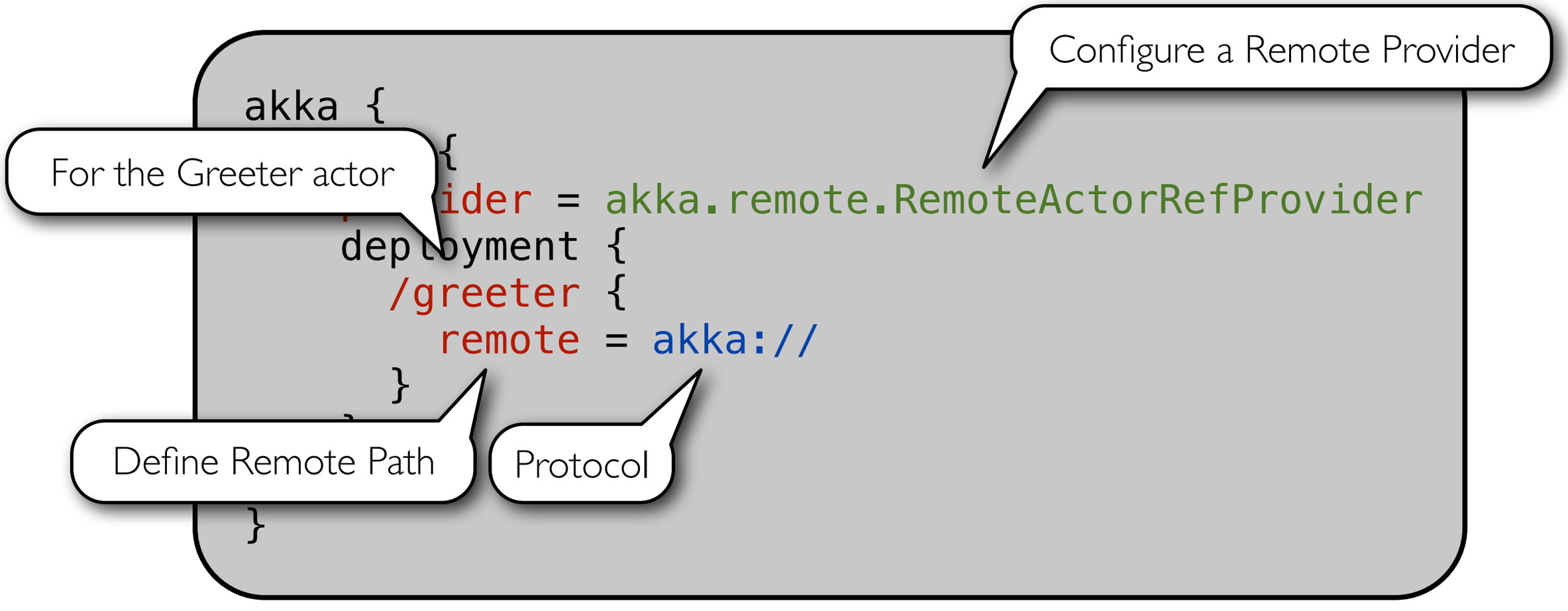
Remote deployment

Just feed the ActorSystem with this configuration



Remote deployment

Just feed the ActorSystem with this configuration

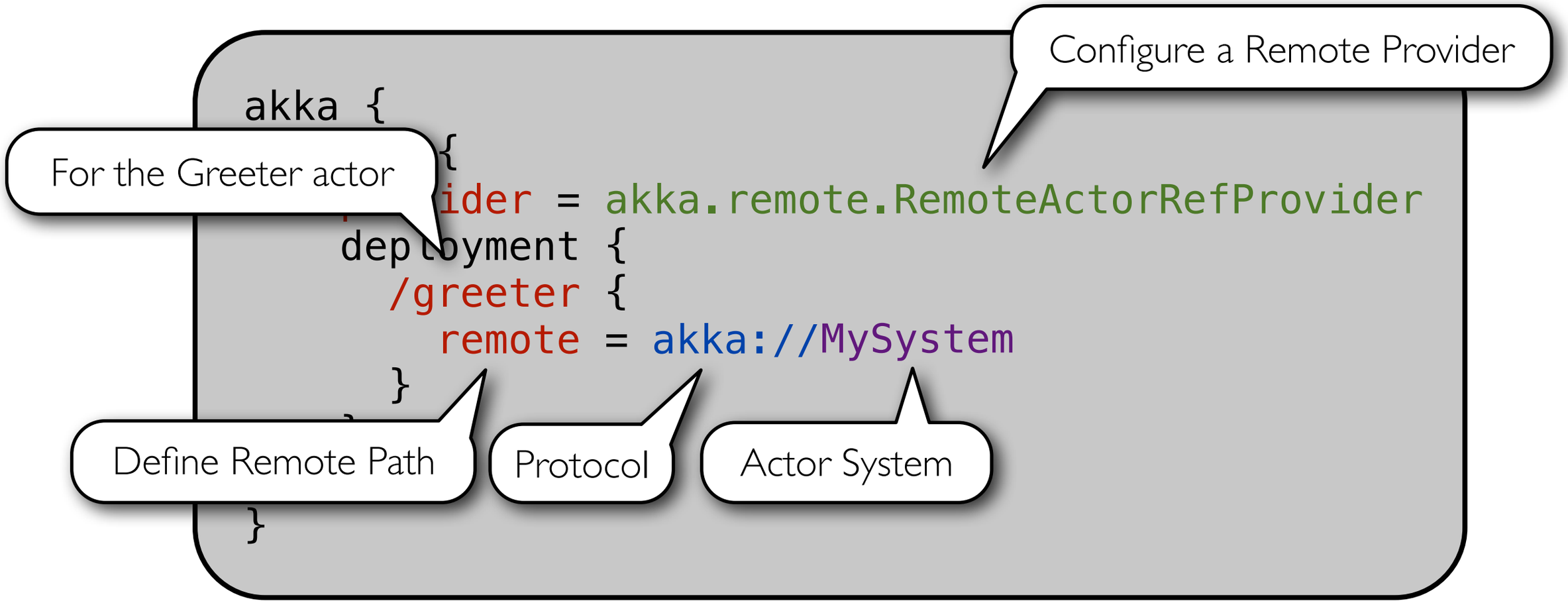


Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

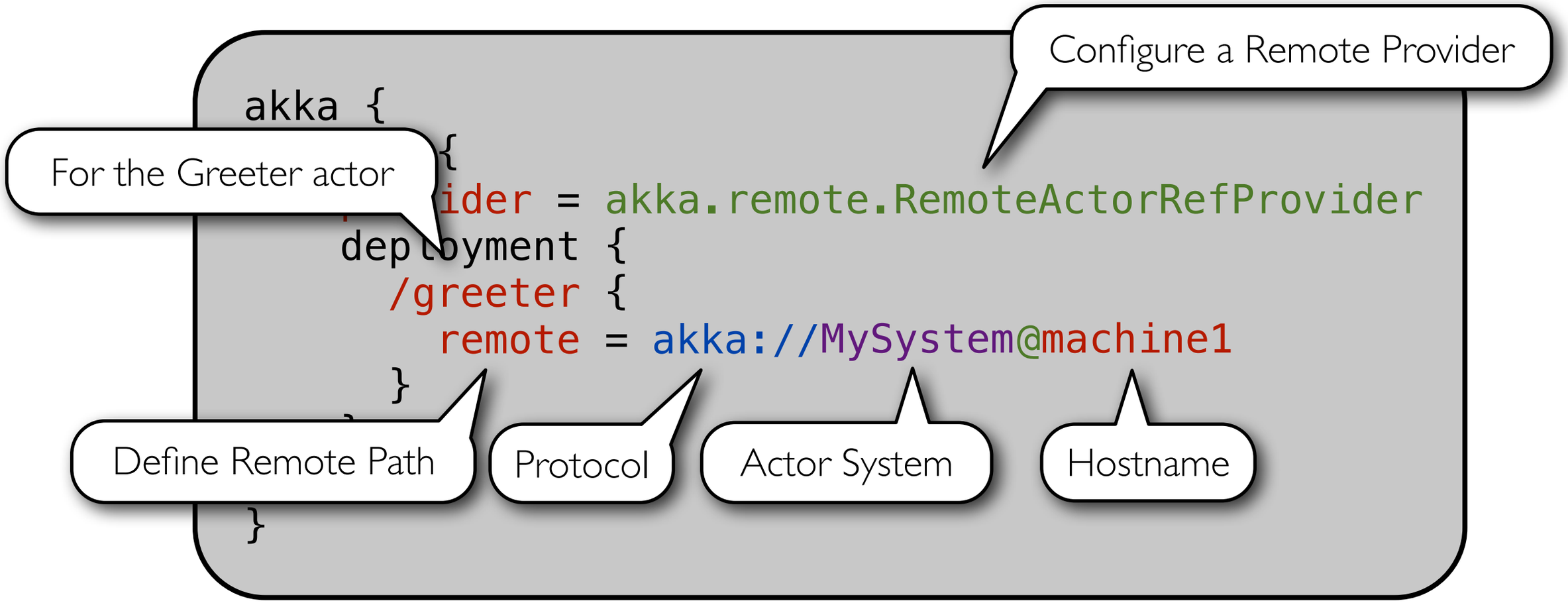


Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

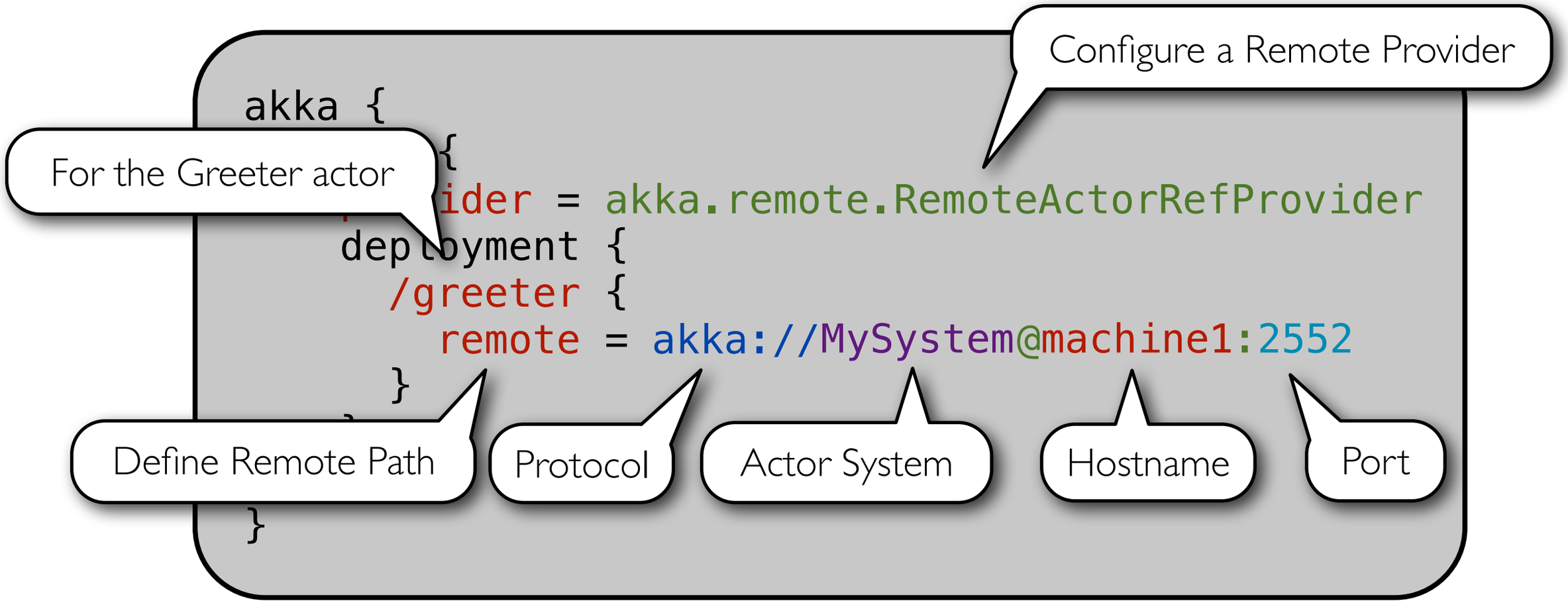


Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

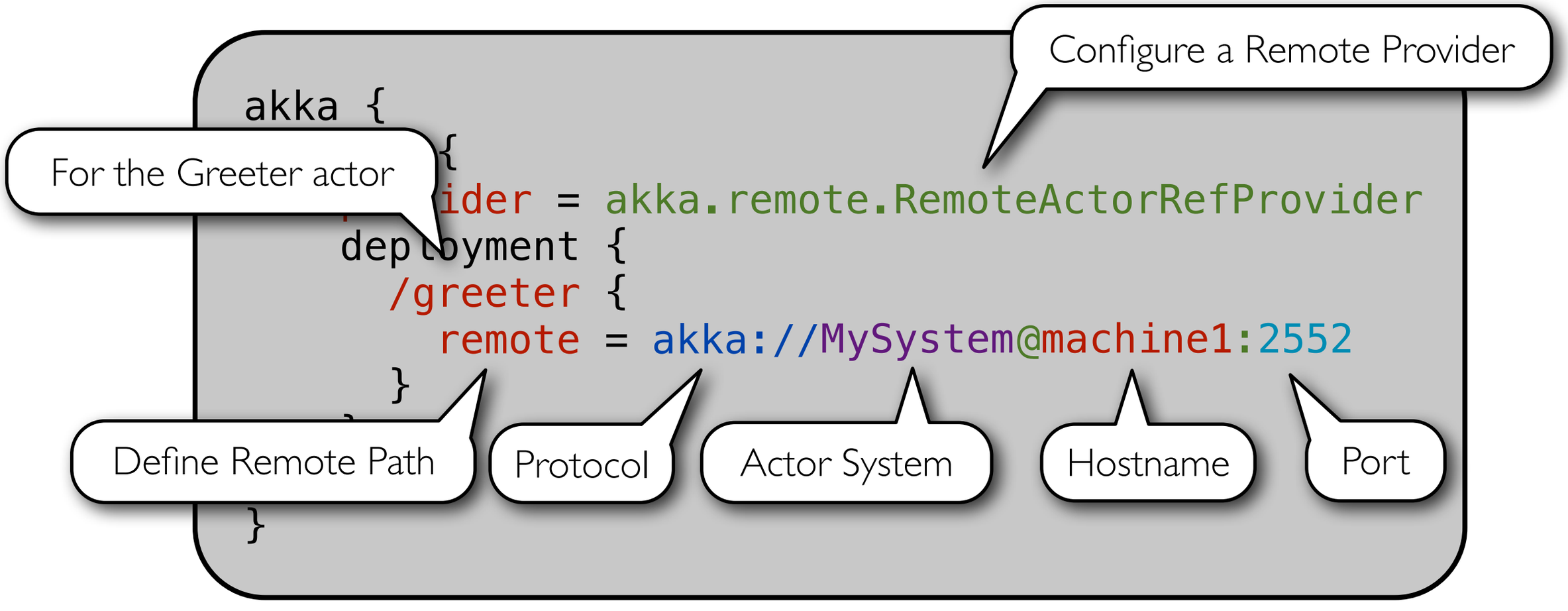


Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration



Zero code changes

Typesafe

akka

Routers

Load Balancing

Routers

```
ActorRef routerActor =
  getContext().actorOf(
    new Props(ExampleActor.class).
    withRouter(new RoundRobinRouter(nrOfInstances))
  );
```

Router + Resizer

```
Resizer resizer =
  new DefaultResizer(lowerBound, upperBound);

ActorRef routerActor =
  getContext().actorOf(
    new Props(ExampleActor.class).
    withRouter(new RoundRobinRouter(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

Typesafe

akka

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 Procedure[Object]() {
  void apply(Object msg) {
    // new body
    if (msg instanceof NewMessage) {
      NewMessage newMsg = (NewMessage)msg;
      ...
    }
  }
});
```

become

Actor context available from within an Actor

```
context.become(new Procedure[Object]() {
  void apply(Object msg) {
    // new body
    if (msg instanceof NewMessage) {
      NewMessage newMsg = (NewMessage)msg;
      ...
    }
  }
});
```

Failure Recovery

Failure Recovery in Java/C/C# etc.

Failure Recovery in Java/C/C# etc.

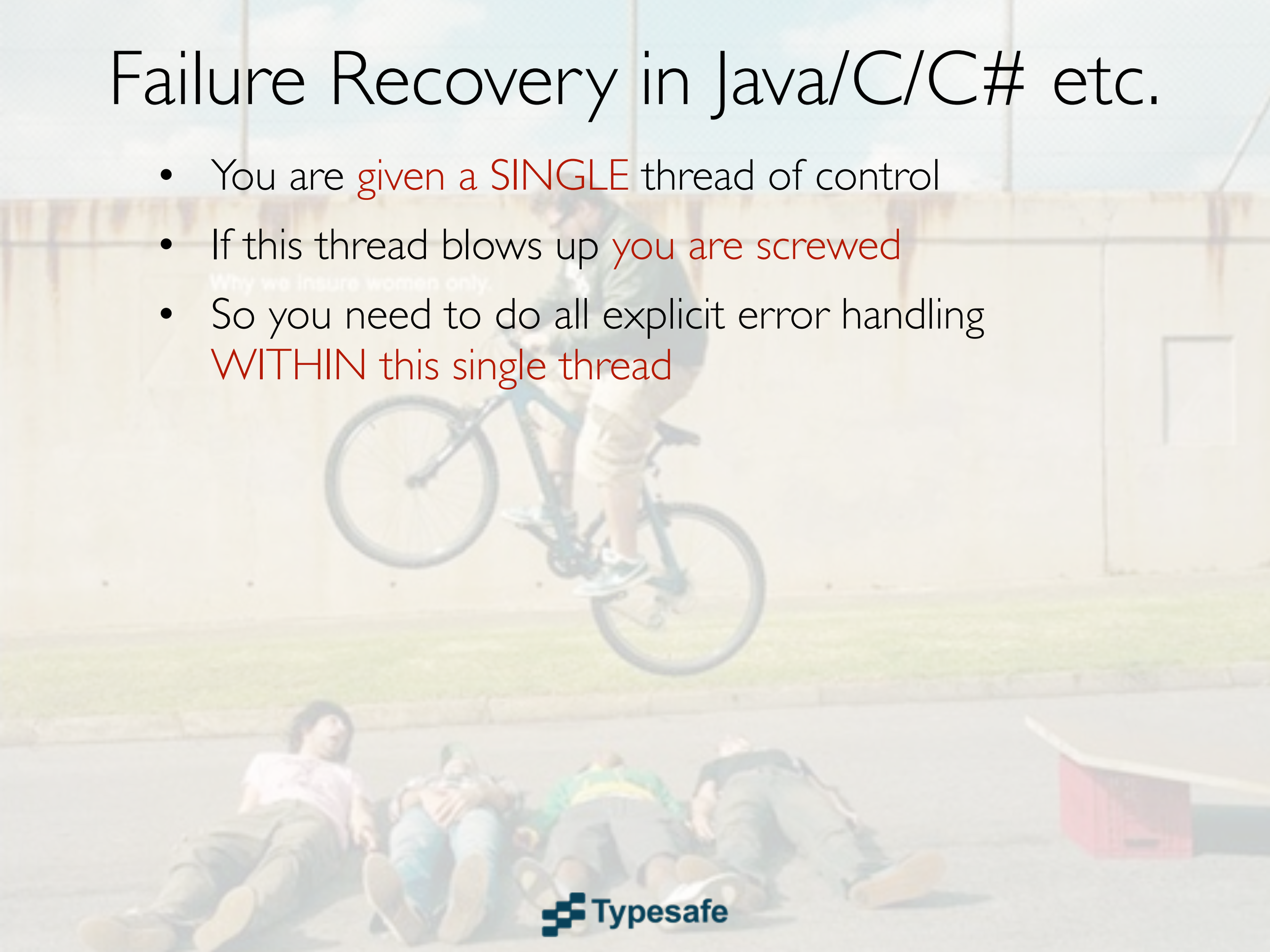
- You are given a SINGLE thread of control

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:

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

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 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

We can do better than this!!!

Typesafe

Just

LET IT CRASH

Typesafe

akka

Typesafe

akka

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

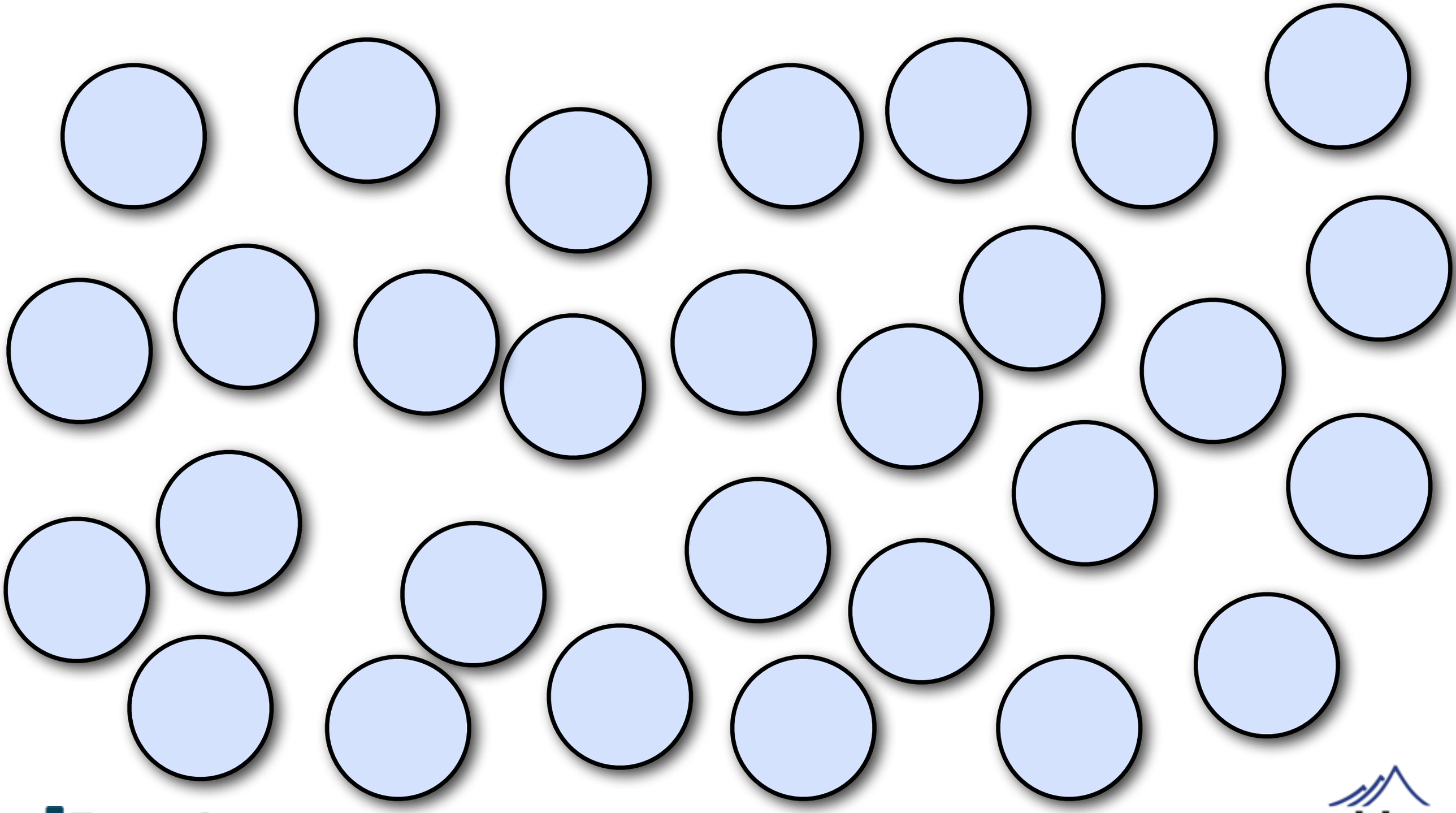
- SUPERVISE - manage another Actor's failures
- Error handling in actors is handle by letting Actors monitor (supervise) each other for failure
- This means that if an Actor crashes, a notification will be sent to his supervisor, who can react upon the failure

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

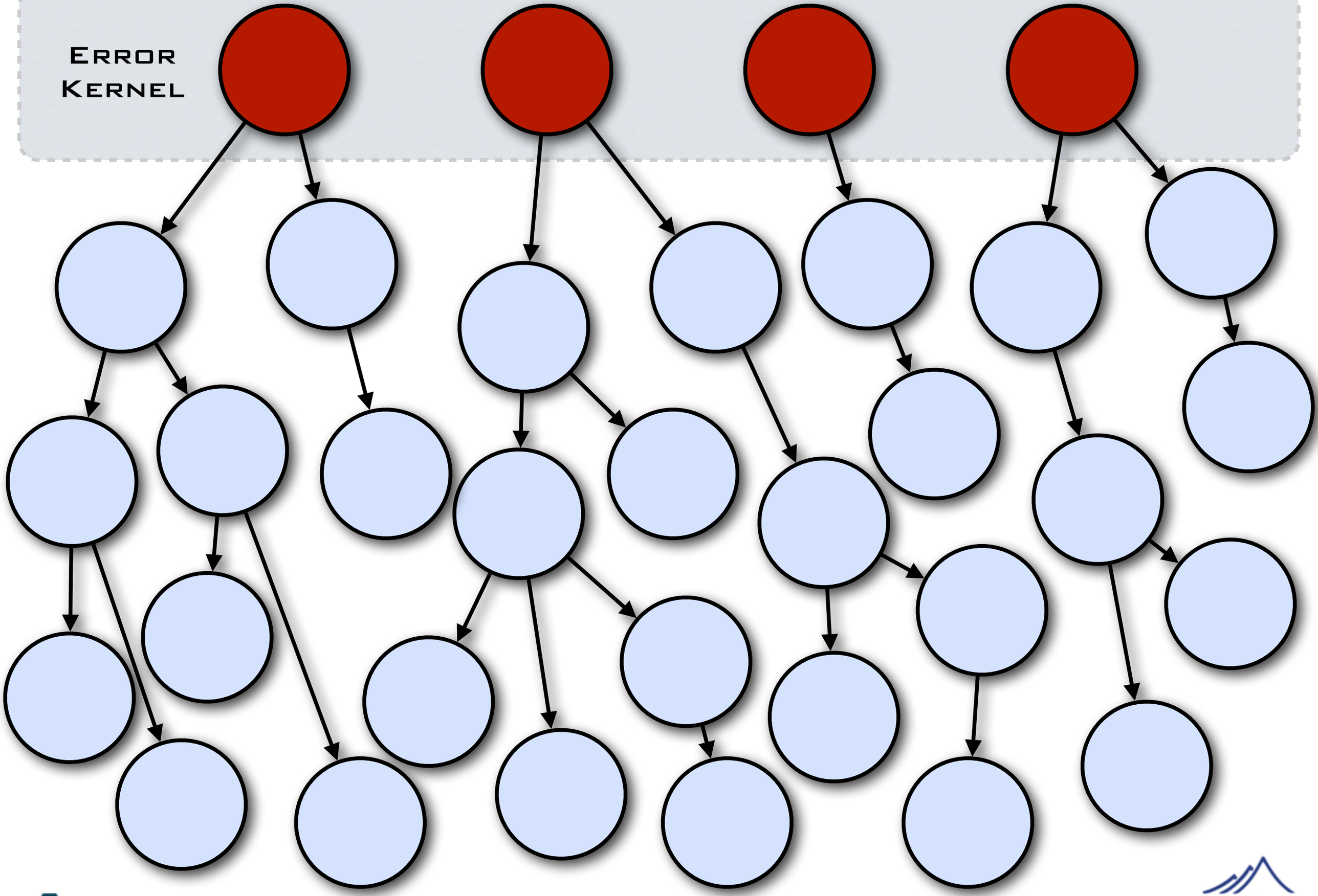
Error Kernel



Typesafe

akka

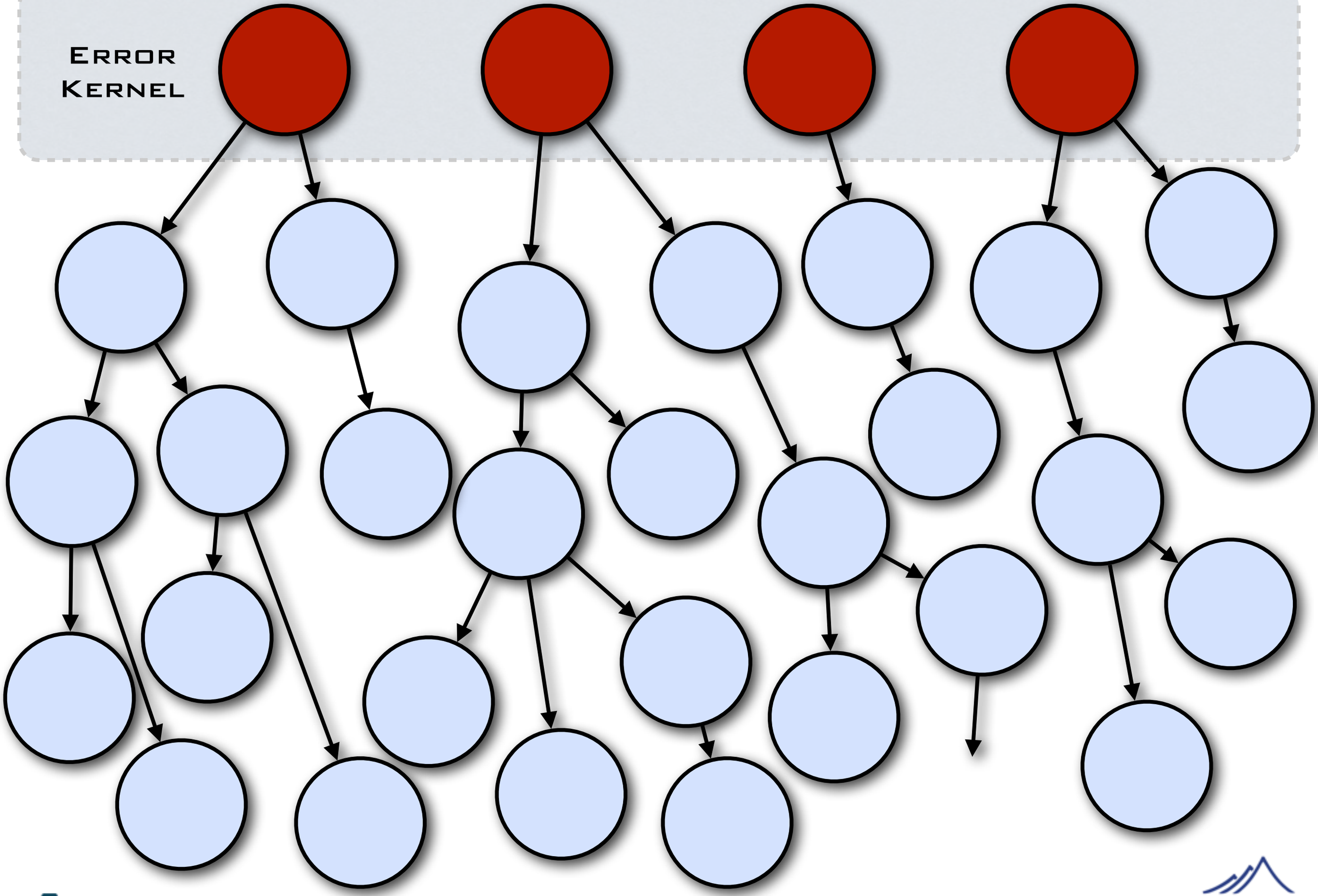
Error Kernel



Typesafe

akka

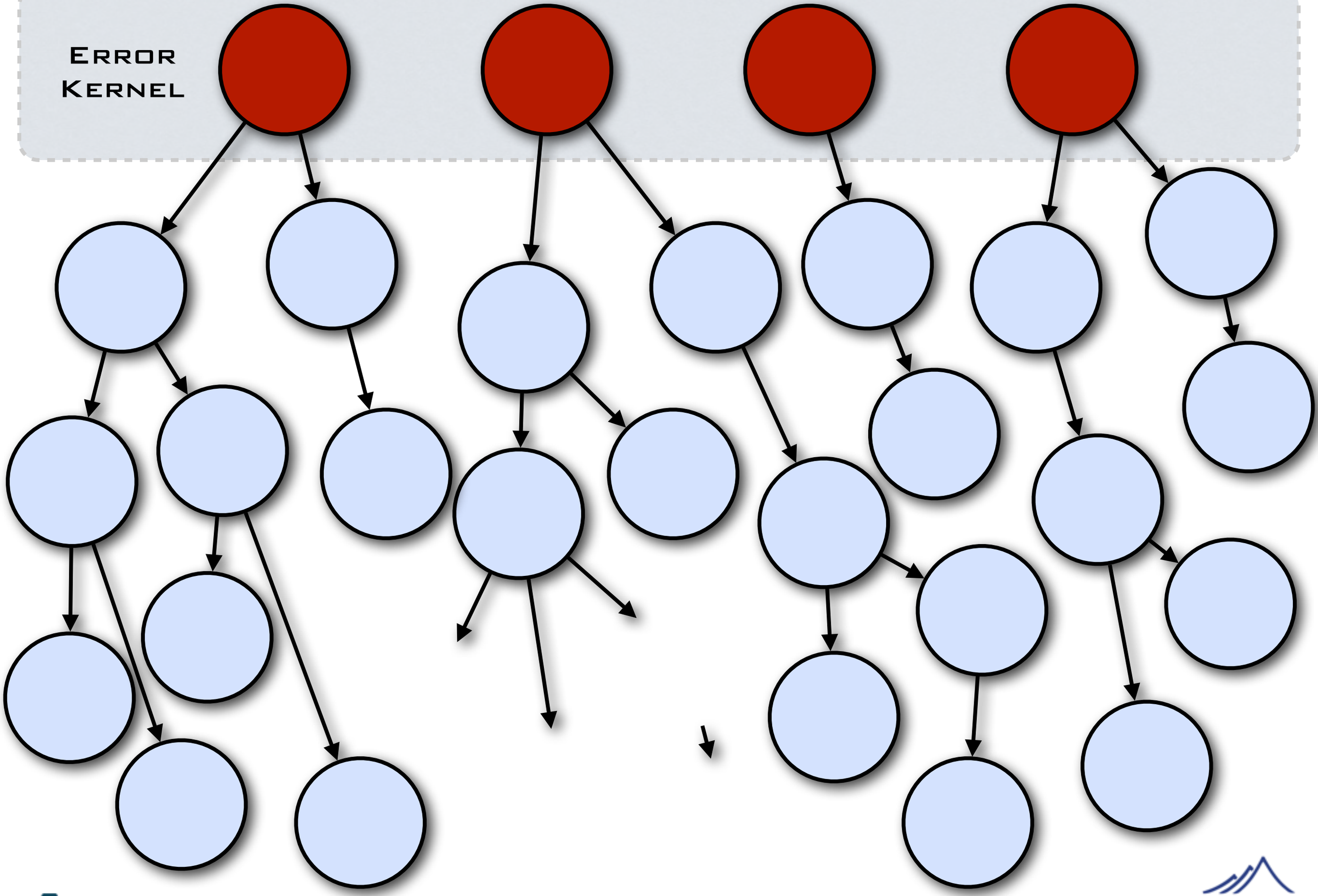
Error Kernel



Typesafe

akka

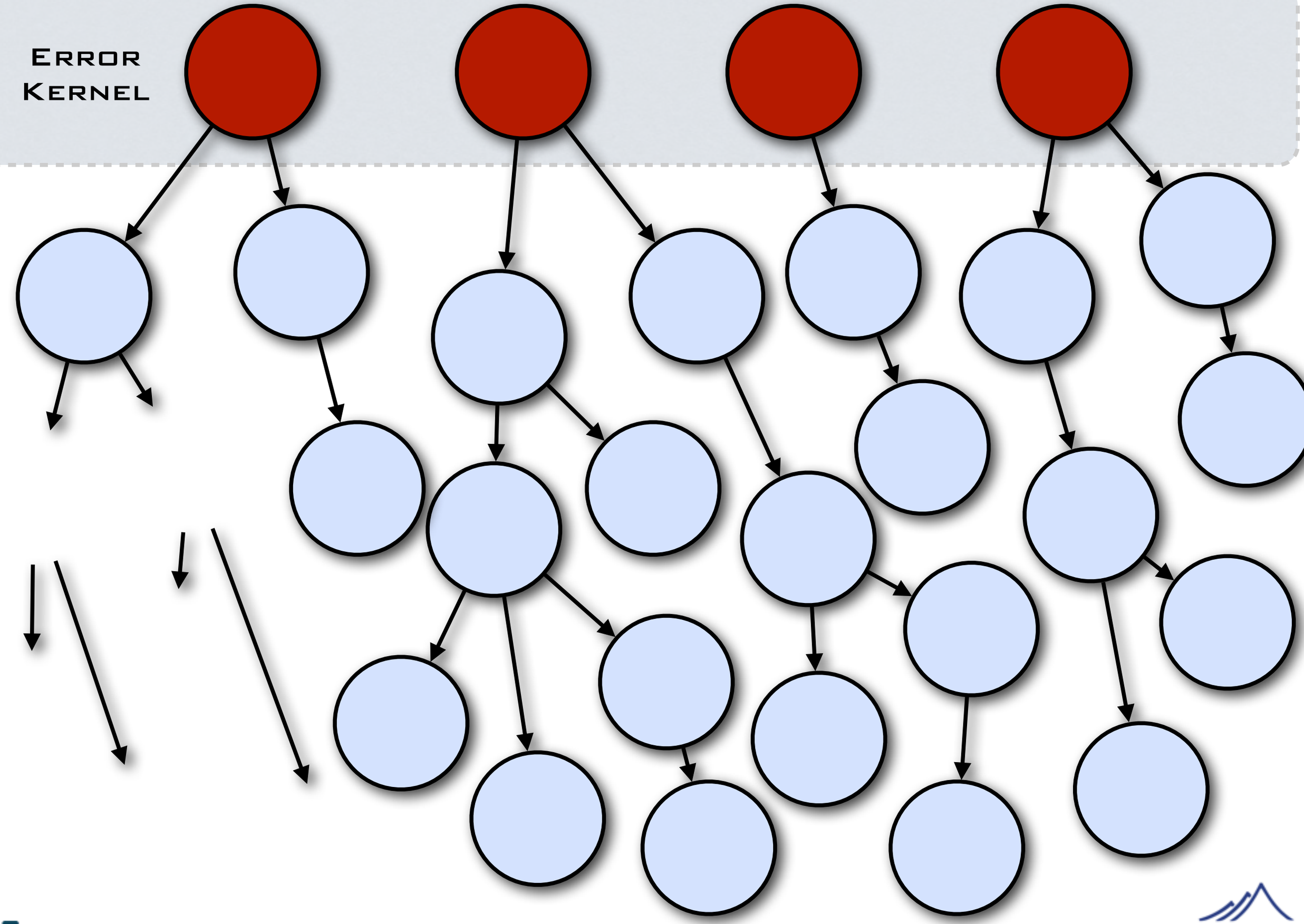
Error Kernel



Typesafe

akka

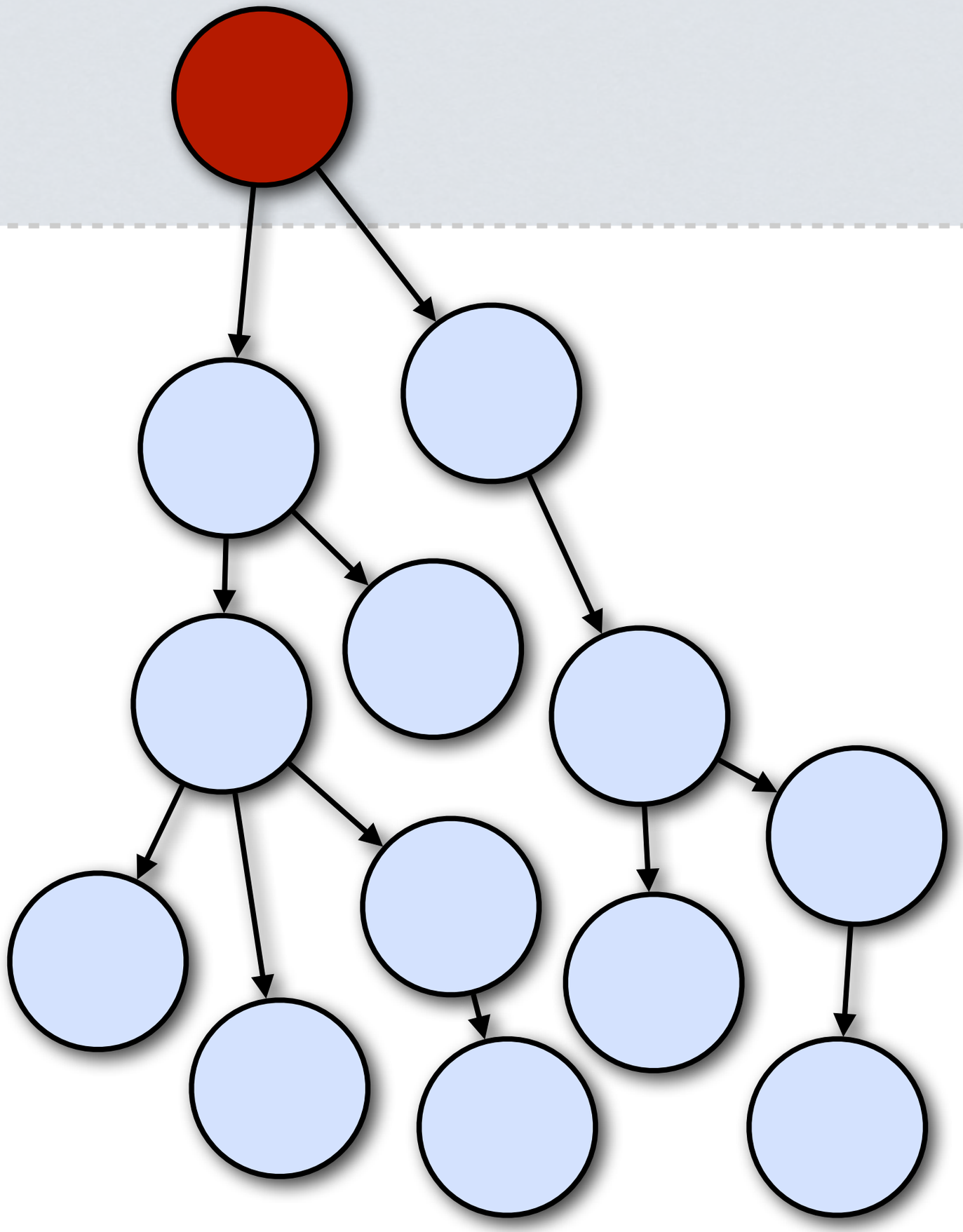
Error Kernel



Typesafe

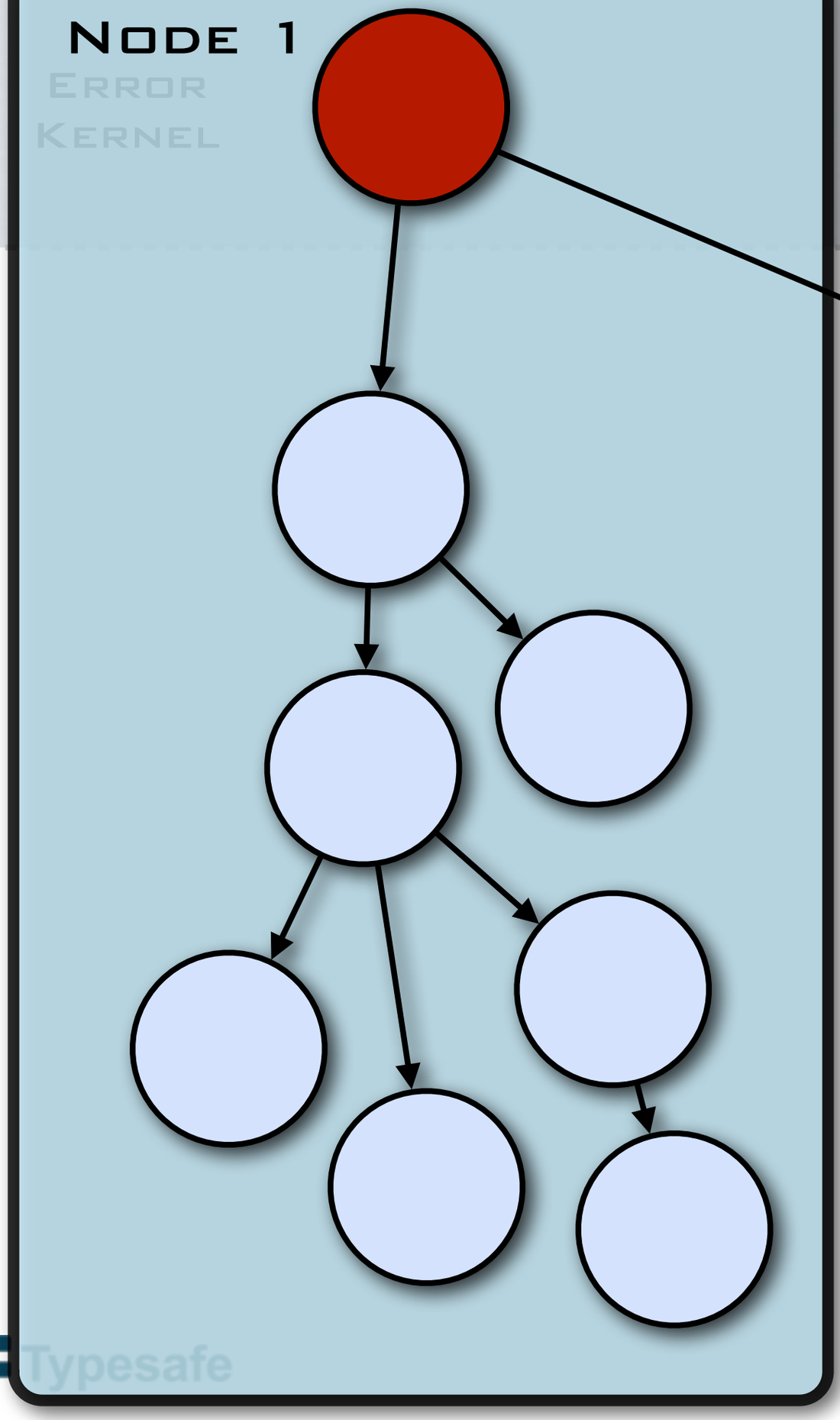
akka

Error Kernel

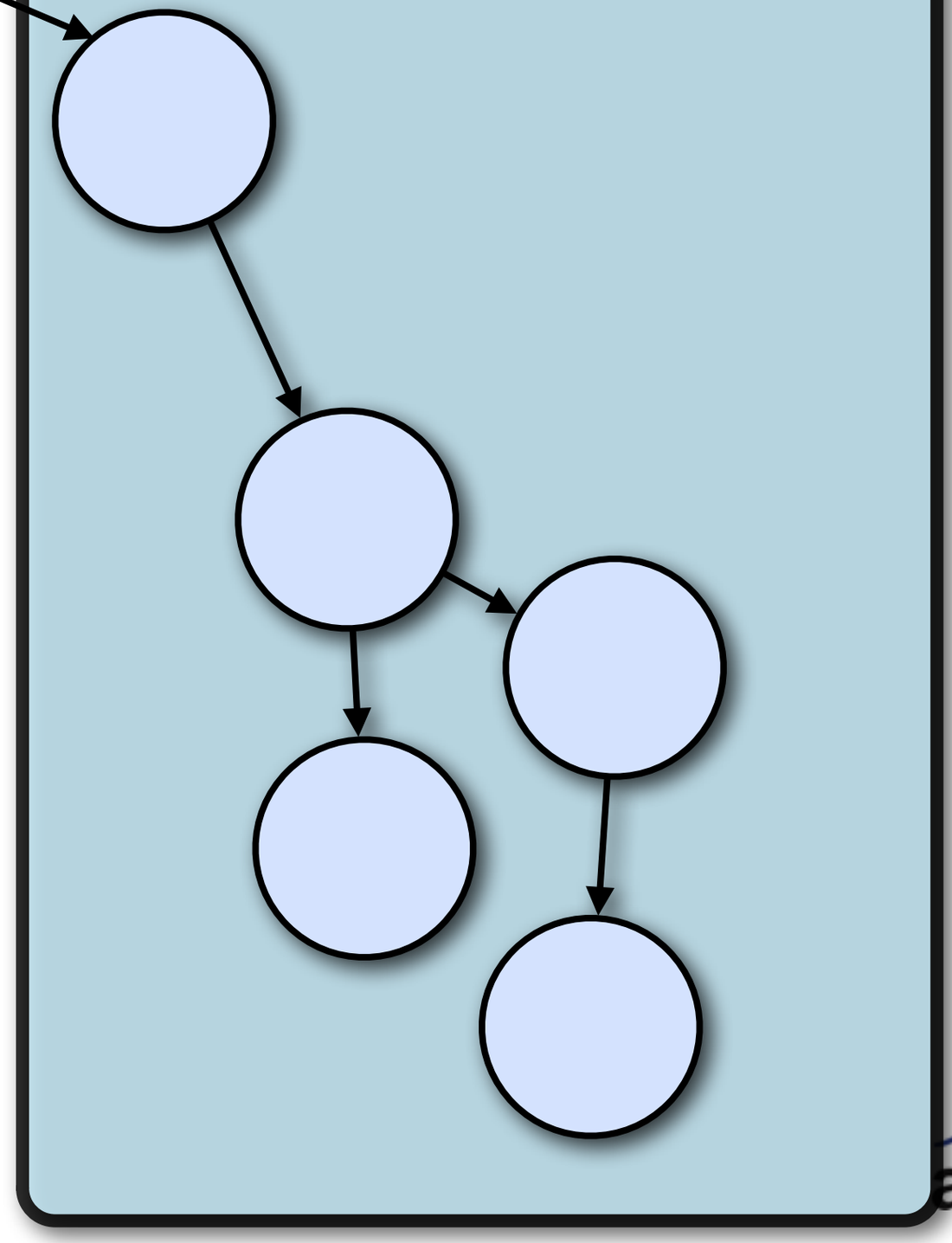


Node 1

Error Kernel



Node 2



Typesafe

akka

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 UntypedActor {
  private SupervisorStrategy strategy = new OneForOneStrategy(
    10,
    Duration.parse("1 minute"),
    new Function<Throwable, Directive>() {
      @Override public Directive apply(Throwable t) {
        if (t instanceof ArithmeticException)       return resume();
        else if (t instanceof NullPointerException) return restart();
        else                                        return escalate();
    }
  });

  @Override public SupervisorStrategy supervisorStrategy() {
```

Typesafe

akka

SUPERVISE Actor

```
class Supervisor extends UntypedActor {
  private SupervisorStrategy strategy = new OneForOneStrategy(
    10,
    Duration.parse("1 minute"),
    new Function<Throwable, Directive>() {
      @Override public Directive apply(Throwable t) {
        if (t instanceof ArithmeticException)        return resume();
        else if (t instanceof NullPointerException) return restart();
        else                                        return escalate();
    }
  });

  @Override public SupervisorStrategy supervisorStrategy() {
    return strategy;
  }

  ActorRef worker = context.actorOf(new Props(Worker.class));

  public void onReceive(Object message) throws Exception {
    if (message instanceof Integer) worker.forward(message);
  }
}
```

Manage failure

```
class Worker extends UntypedActor {
  …
  @Override
  public void preRestart(Throwable cause, Option<Object> msg){
    … // clean up before restart
  }

  @Override
  public void postRestart(Throwable cause){
    … // init after restart
  }
}
```

This was

Akka **2.0**

Akka **2.1+**

Akka Cluster

Experimental module in 2.1

Highlights

- Automatic cluster-wide deployment
- Decentralized P2P gossip-based cluster membership
- Leader "election"
- Adaptive load-balancing (based on runtime metrics)
- Automatic replication with automatic fail-over upon node crash
- Automatic adaptive cluster rebalancing
- Highly available configuration service

Gossiping Protocol

Gossiping Protocol

Used for:

Gossiping Protocol

Used for:

- Cluster Membership

Gossiping Protocol

Used for:

- Cluster Membership
- Configuration data

Gossiping Protocol

Used for:

- Cluster Membership
- Configuration data
- Partitioning data

Gossiping Protocol

Used for:

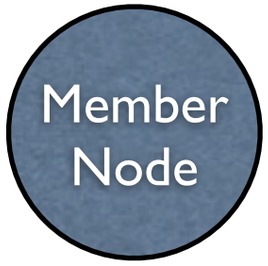
- Cluster Membership
- Configuration data
- Partitioning data
- Naming Service

Gossiping Protocol

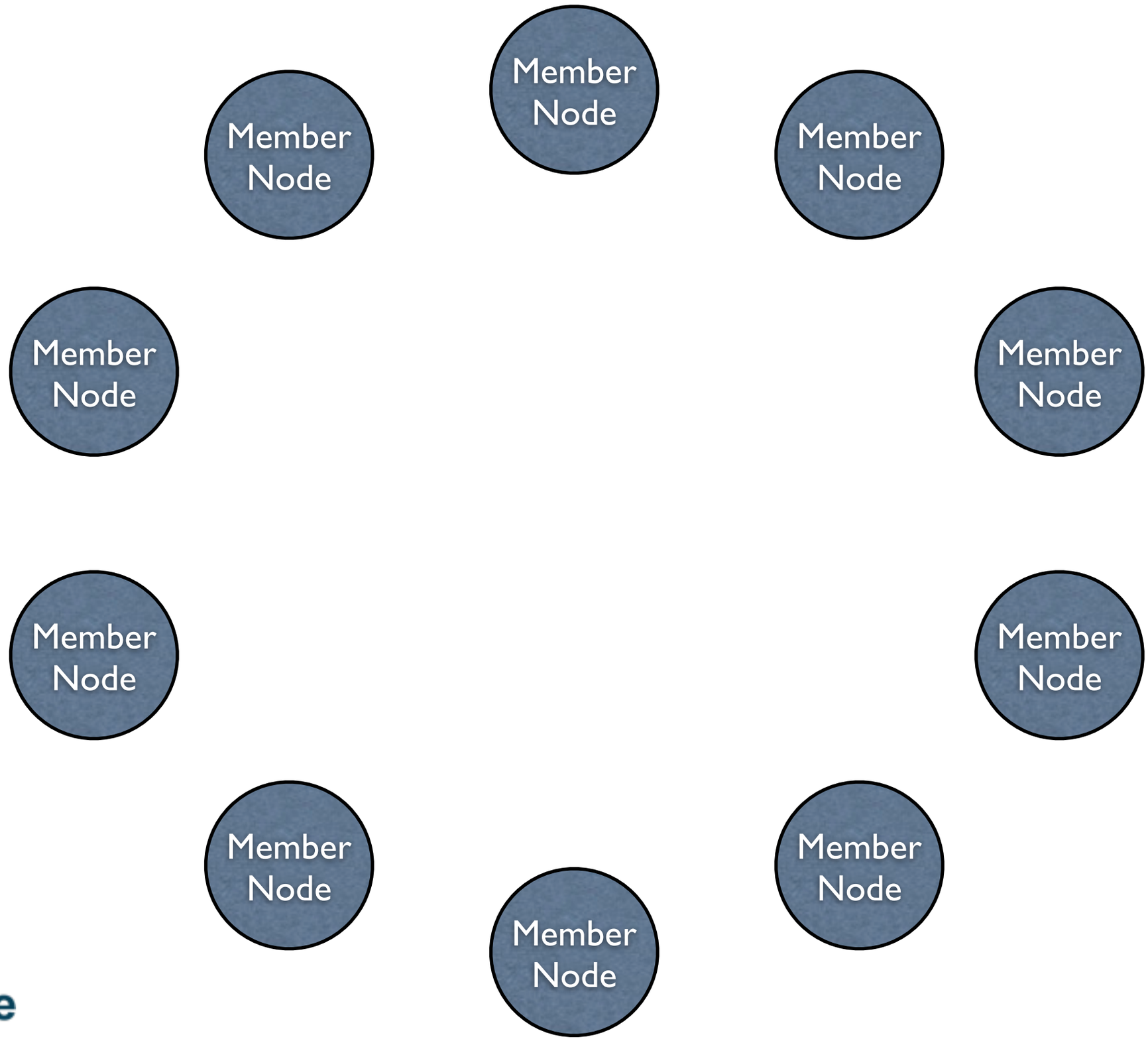
Used for:

- Cluster Membership
- Configuration data
- Partitioning data
- Naming Service
- Leader "election"

Node ring with gossiping members



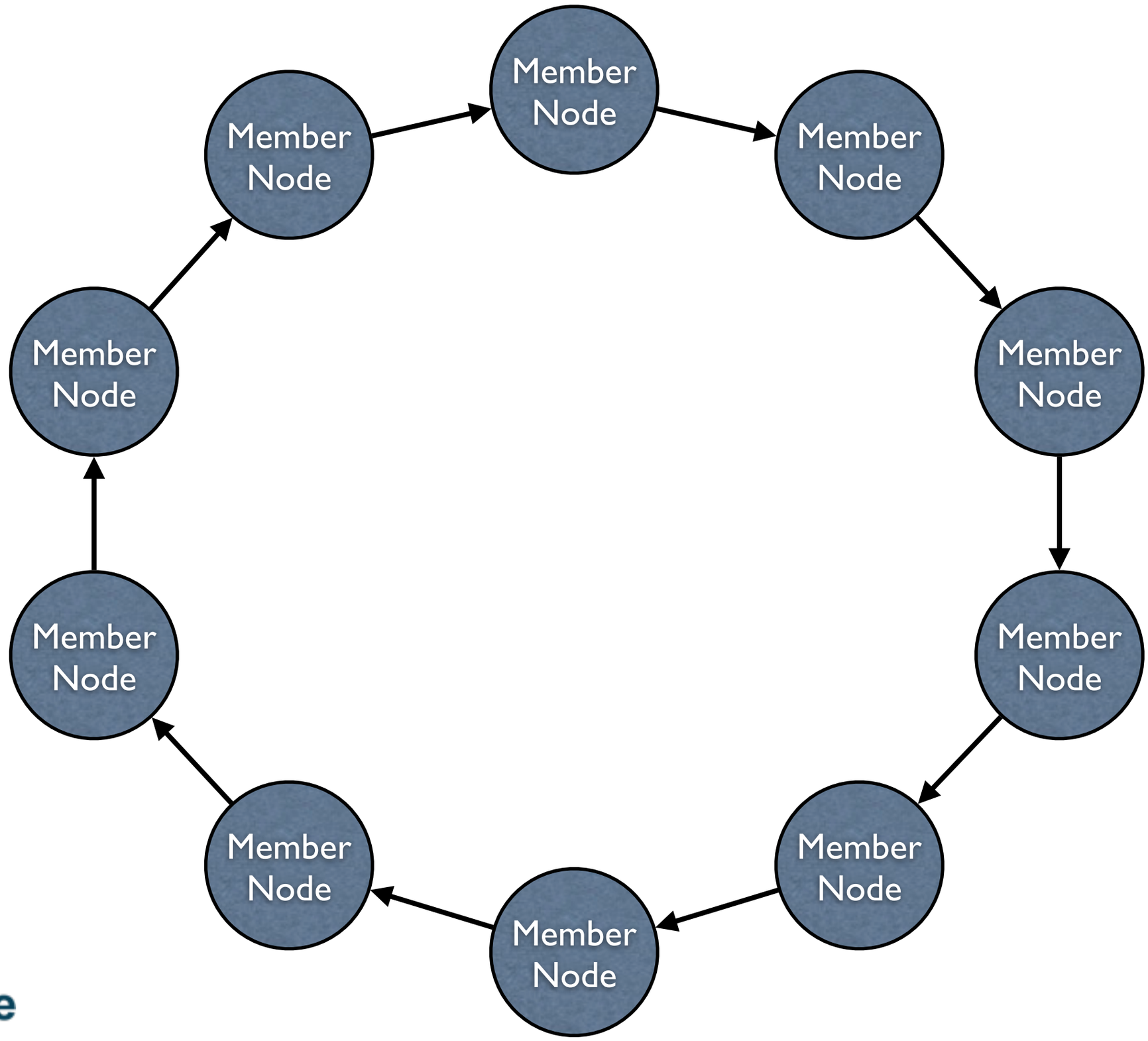
Node ring with gossiping members



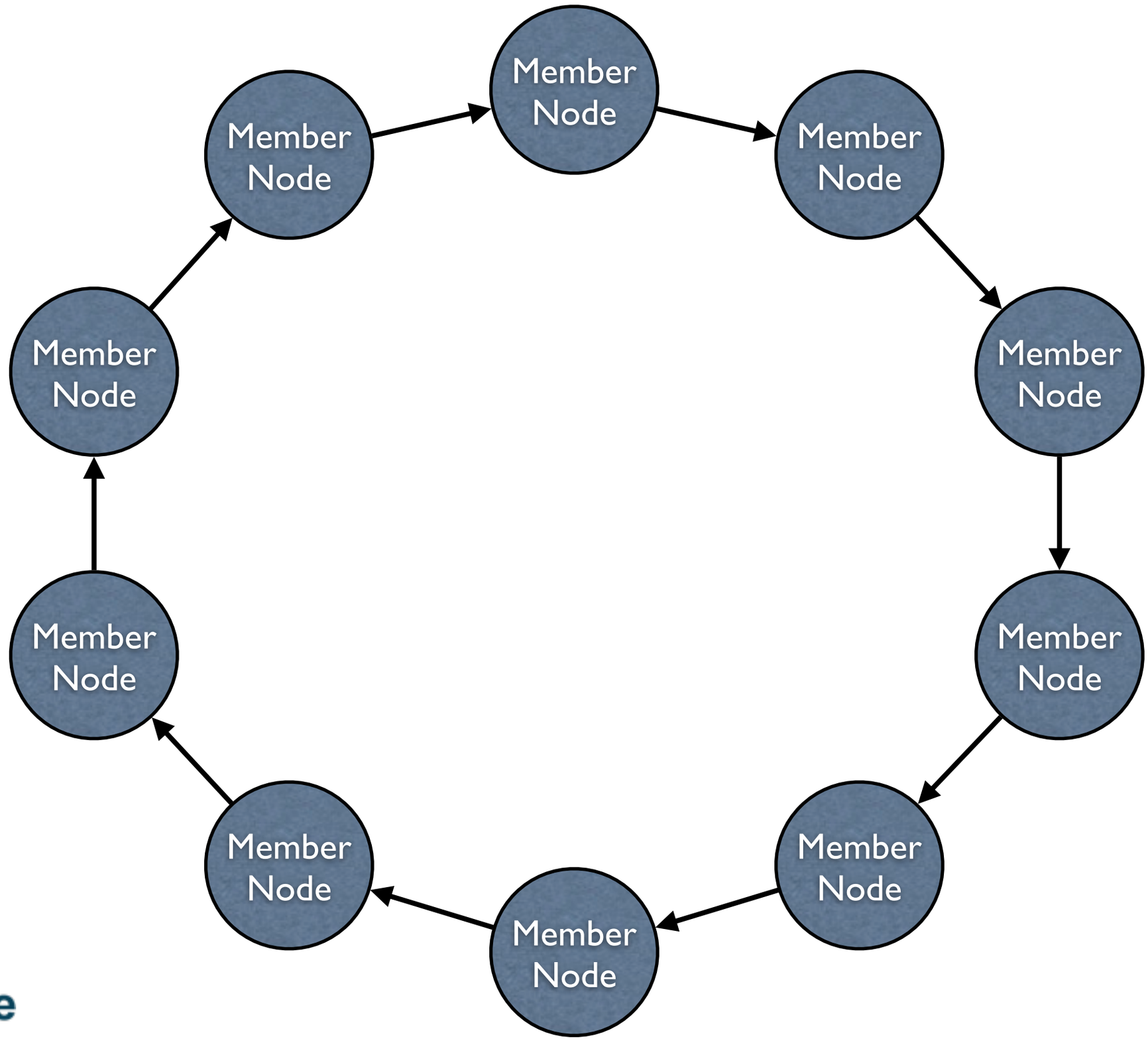
Typesafe

akka

Node ring with gossiping members



Node ring with gossiping members



Gossiping Protocol

Gossiping Protocol

- Variation of the Riak (Dynamo) protocol

Gossiping Protocol

- Variation of the Riak (Dynamo) protocol
- Vector Clocks

Gossiping Protocol

- Variation of the Riak (Dynamo) protocol
- Vector Clocks
- Eventually Consistent (CAP)

Gossiping Protocol

- Variation of the Riak (Dynamo) protocol
- Vector Clocks
- Eventually Consistent (CAP)
- Convergence

Gossiping Protocol

- Variation of the Riak (Dynamo) protocol
- Vector Clocks
- Eventually Consistent (CAP)
- Convergence
- Leader can be deterministically determined

Gossiping Protocol

- Variation of the Riak (Dynamo) protocol
- Vector Clocks
- Eventually Consistent (CAP)
- Convergence
- Leader can be deterministically determined
- Accrual Failure Detector

Gossiping Protocol

- Variation of the Riak (Dynamo) protocol
- Vector Clocks
- Eventually Consistent (CAP)
- Convergence
- Leader can be deterministically determined
- Accrual Failure Detector
- Push/Pull Gossip

Gossiping Protocol

- Variation of the Riak (Dynamo) protocol
- Vector Clocks
- Eventually Consistent (CAP)
- Convergence
- Leader can be deterministically determined
- Accrual Failure Detector
- Push/Pull Gossip
- Seed Nodes

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
    }
  }
}
```

Cluster Specification

doc.akka.io/docs/akka/snapshot/cluster/cluster.html

Cluster User Guide

doc.akka.io/docs/akka/snapshot/cluster/cluster-usage.html

Cluster Code

github.com/akka/akka/tree/master/akka-cluster

Typesafe **Console**

free for developers later in the fall

Typesafe | Documentation | Demo | demo@typesafe.com LOGOUT

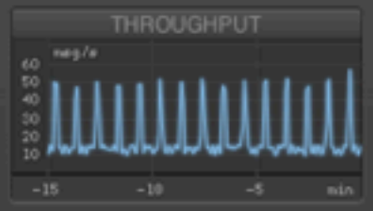
Search (or 'help')

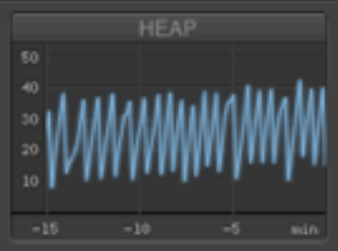
NODES | DISPATCHERS | ACTORS | TAGS

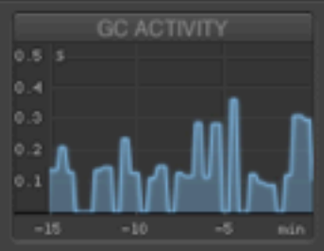
NODES
Nodes in your system

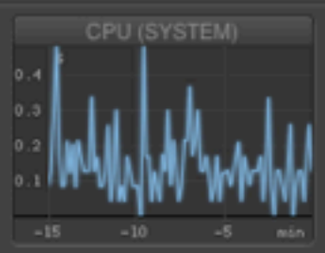
node1
7 actors | 2 dispatchers

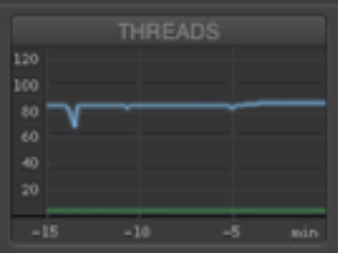
MAX MAILBOX: 68
MEAN MAILBOX: 12.51
PEAK RATES: 35.41 · 26.11
REMOTE ERR.: 0











MACHINE

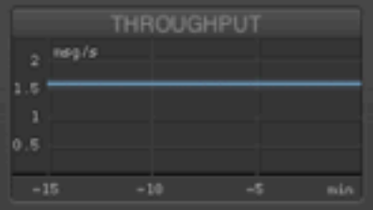
Max Heap:	8105 MB
Used Heap:	1232 MB
Memory Usage:	76.70 %
Running Actors:	6
Processors:	4
Up Time:	13832 mins

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

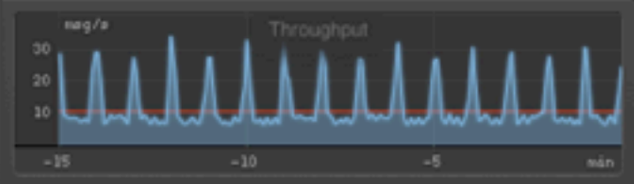
MAX MAILBOX: 1
MEAN MAILBOX: 0
PEAK RATES: - - -
REMOTE ERR.: 0



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

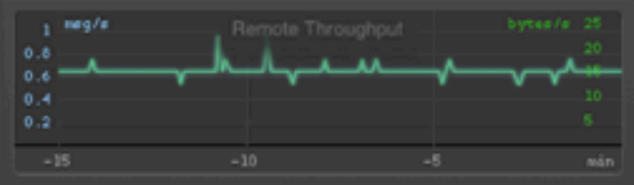
Errors:	30
Warnings:	15
Deadlocks:	20
Dead letters:	0

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

Typesafe | Documentation | Demo | demo@typesafe.com LOGOUT

d:*

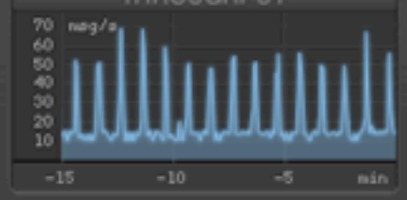
NODES | DISPATCHERS | ACTORS | TAGS

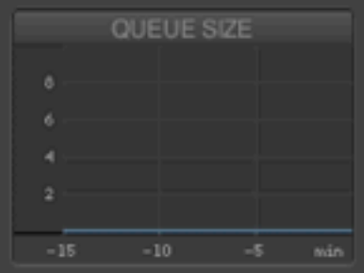
DISPATCHERS
Dispatchers for your query

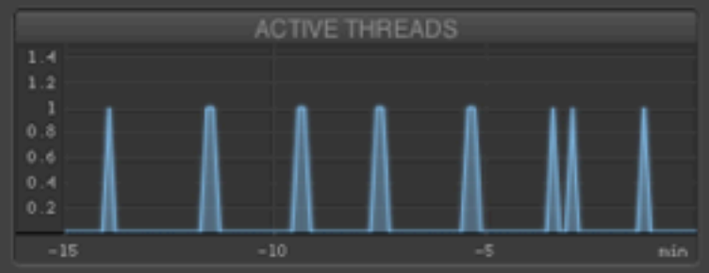
akka.actor.default-dispatcher

NODE: node1

TYPE: Dispatcher-FJ







ACTOR COUNTS

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

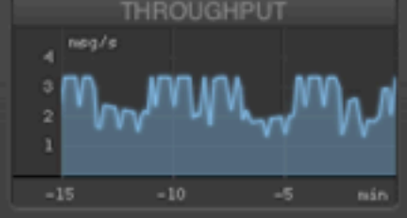
ACTOR RATES

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

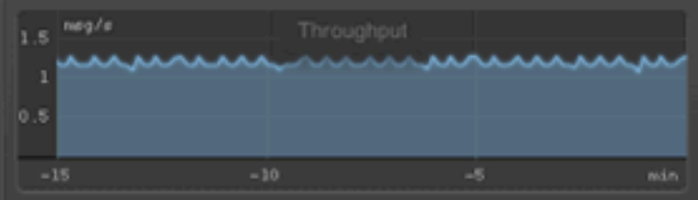
TYPE: Dispatcher-TP

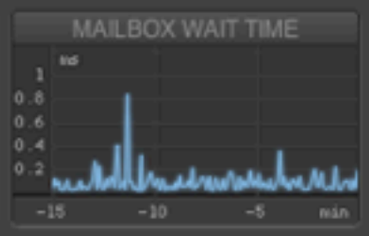


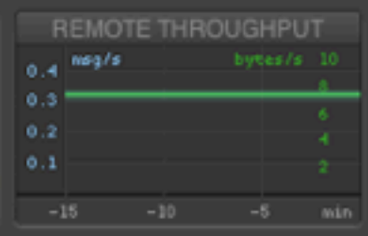
ACTOR
Actor for your query

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

Lots of statistics

- message rates and peaks
- individual trace events linked as trace trees or spans
- latency between predefined or user defined points
- latency distributions
- actor supervisor hierarchies

- actor mailbox queue size and delays
- actor message dispatcher status
- status of remoting and system errors
- metadata about the system
- JVM and OS health

live demo

http://console-demo.typesafe.com

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

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

Routing

TestKit

FSM

Durable Mailboxes

Pub/Sub

EventBus

Camel

IO

Pooling

Microkernel

TypedActor

SLF4J

ZeroMQ

Dataflow

Transactors

Agents

Extensions

get it and learn more

http://akka.io

http://letitcrash.com

http://typesafe.com

EØF