Introducing akka

Jonas Bonér

CTO Typesafe

Typesafe

Introducing

Introducing

Akka (Áhkká)

The name comes from the goddess in the Sami (native swedes) mythology that represented all the wisdom and beauty in the world.

It is also the name of a beautiful mountain in Laponia in the north part of Sweden

Vision

Simpler

- Concurrency
- Scalability
- Fault-tolerance

Typesafe

akka

Vision

...with a single unified

- Programming Model
- Managed Runtime
- Open Source Distribution

Typesafe

akka

Manage System Overload

Typesafe

akka

Scale UP & Scale OUT

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.

Typesafe

akka

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

Typesafe

akka

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

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

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

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

Selection of Akka Production Users

TDC
Telefónica
CSC
htc quietly brilliant
hp
BLIZZARD ENTERTAINMENT
UBS
SIEMENS
amazon.com
KLOUT
W3C
HSBC
CISCO
HUAWEI
banksimple
svt
Autodesk
CREDIT SUISSE
IGN
Atos
O2
vmware
dialog Smart Stream Platform
SEVEN Networks
xerox
UNIBET
DRW TRADING GROUP
OOYALA
novus
BBC
navirec
T8 Webware
CARTOMAPIC
Moshi Monsters
JUNIPER NETWORKS
ngmoco:)
Maritime Poker
azavea
zeebox The best thing to happen to TV since TV
Answers.com The world's leading Q&A site

How

can we achieve this?

Typesafe

akka

Let’s use Actors

What is an Actor?

What is an Actor?

Typesafe

akka

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

Actors can be seen as
Virtual Machines (nodes)
in Cloud Computing

Encapsulated
and decoupled
black boxes...

Encapsulated
and decoupled
black boxes...

...that manages their own memory and behavior

...that manages their own memory and behavior

Communicates
with asynchronous
non-blocking
messages

Communicates
with asynchronous
non-blocking
messages

Elastic - grow and
shrink on demand

Elastic - grow and shrink on demand

Elastic - grow and shrink on demand

Elastic - grow and
shrink on demand

Hot deploy - change behavior at runtime

Hot deploy - change behavior at runtime

Now - if we replace

with

Actor

EVERYTHING will STILL HOLD for both LOCAL and DISTRIBUTED Actors

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);
      }
   }
}
```

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);
      }
   }
}
```

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
   public Greeting(Stri                    = 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
   public Greeting(Stri                    = 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("Hell  {}", ((Greeting) message).who);
      }
   }
}
```

Define the Actor class

Define the Actor's behavior

Typesafe

akka

Scala version

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

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

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

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

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

Typesafe

akka

Actors can form hierarchies

Guardian System Actor

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

Typesafe

akka

Actors can form hierarchies

Typesafe

akka

Actors can form hierarchies

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

Typesafe

akka

Actors can form hierarchies

Typesafe

akka

Actors can form hierarchies

Typesafe

akka

Name resolution - like a file-system

Typesafe

akka

Name resolution - like a file-system

Typesafe

akka

Name resolution - like a file-system

Typesafe

akka

Name resolution - like a file-system

Typesafe

akka

Name resolution - like a file-system

Typesafe

akka

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 happens Reactively

2. SEND

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

Typesafe

akka

2. SEND

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

2. SEND

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

Throughput on a single box

+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"));
```

Scala version

```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 tell Greeting("Charlie Parker")
```

Reply

```
class SomeActor extends UntypedActor {
  void onReceive(Object msg) {
    if (msg instanceof User) {
      User user = (User) msg;
      // reply to sender
      getSender().tell("Hi " + user.name);
    }
  }
}
```

Scala version

```
class SomeActor extends Actor {
  def receive = {
    case User(name) =>
      // reply to sender
      sender tell (“Hi ” + name)
  }
}
```

Typesafe

akka

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

Configure a Remote Provider

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

Remote deployment

Just feed the ActorSystem with this configuration

Configure a Remote Provider

For the Greeter actor

```
akka {
  [illegible] {
    [illegible]ider = akka.remote.RemoteActorRefProvider
    deployment {
      /Greeter {
        remote =
      }
    }
  }
}
```

Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
  [illegible] {
    [illegible]ider = akka.remote.RemoteActorRefProvider
    deployment {
      /Greeter {
        remote =
      }
    [illegible]
  }
```

Configure a Remote Provider

For the Greeter actor

Define Remote Path

Typesafe

akka

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
  [illegible] {
    [illegible]ider = akka.remote.RemoteActorRefProvider
    deployment {
      /Greeter {
        remote = akka://
      }
    [illegible]
  }
```

Configure a Remote Provider

For the Greeter actor

Define Remote Path

Protocol

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
  [illegible] {
    [illegible]ider = akka.remote.RemoteActorRefProvider
    deployment {
      /Greeter {
        remote = akka://MySystem
      }
    [illegible]
  }
```

Configure a Remote Provider

For the Greeter actor

Define Remote Path

Protocol

Actor System

Remote deployment

Just feed the ActorSystem with this configuration

```
akka {
  [illegible] {
    [illegible]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

Remote deployment

Just feed the ActorSystem with this configuration

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

Configure a Remote Provider

For the Greeter actor

Define Remote Path

Protocol

Actor System

Hostname

Port

Remote deployment

Just feed the ActorSystem with this configuration

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

Configure a Remote Provider

For the Greeter actor

Define Remote Path

Protocol

Actor System

Hostname

Port

Zero code changes

Typesafe

akka

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 himself into an Actor Pool or a Router

Why would I want to do that?

- Let a highly contended Actor adaptively transform himself 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 himself 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 himself 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 himself 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
- etc. use your imagination

Why would I want to do that?

- Let a highly contended Actor adaptively transform himself 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
- etc. 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;
      ...
    }
  }
});
```

Scala version

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

Load Balancing

Routers

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

Scala API

Router + Resizer

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

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

Scala API

Failure management in Java/C/C# etc.

Typesafe

Failure management in Java/C/C# etc.

- You are given a SINGLE thread of control

Typesafe

Failure management in Java/C/C# etc.

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

Typesafe

Failure management 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 management 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 management 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 management 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 management 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 management 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

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

...let's take a

standard OO

application

Typesafe
kka

Which components have

critically important state

and

explicit error handling?

Typesafe
kka

Fault-tolerant onion-layered Error Kernel

Error
Kernel
Typesafe
akka

Typesafe

akka

Typesafe

akka

Typesafe

akka

Typesafe

akka

Error Kernel

Typesafe

akka

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);
  }
}
```

Scala version

```
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
  }
}
```

Typesafe

akka

Scala version

```
class Supervisor extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy(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
  }
}
```

Typesafe

akka

Scala version

```
class Supervisor extends Actor {
  override val supervisorStrategy =
    AllForOneStrategy(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
  }
}
```

Typesafe

akka

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
  }
}
```

Scala API

Typesafe

akka

This was

Akka **2.x**

This was

Akka **2.x**

Well...it's a start...

...we have much much more

...we have much much more

TestKit

Cluster

FSM

IO

HTTP/REST

EventBus

Durable Mailboxes

Pub/Sub

Camel

Microkernel

TypedActor

SLF4J

AMQP

ZeroMQ

Dataflow

Transactors

Agents

Spring

Extensions

get it and learn more

http://akka.io

http://letitcrash.com

http://typesafe.com

EOF