Akka Made Our Day

JFokus 2014

Daniel Deogun & Daniel Sawano
Email: Daniel.deogun@Omegapoint.se, Daniel.Sawano@Omegapoint.se
Twitter: @DanielDeogun, @DanielSawano

omega point.

Who We Are

Daniel Deogun

Daniel Sawano

Omegapoint

Stockholm - Gothenburg - Malmoe - Umea - New York

omega point.

Agenda

- Akka in a nutshell
- Akka & Java
- Domain influences
- Lessons learned from building real systems with Akka

Akka in a Nutshell

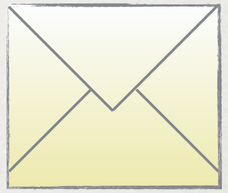
http://akka.io/

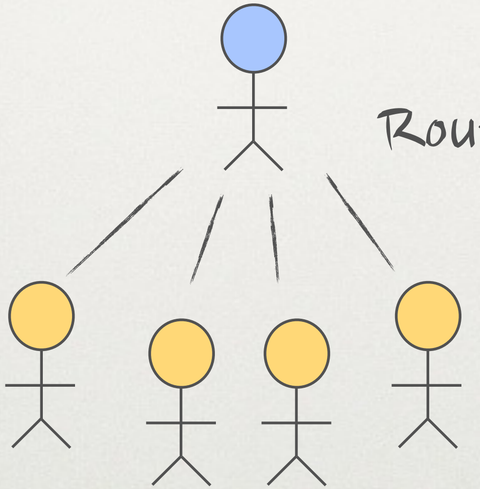
omega point.

Akka in a Nutshell

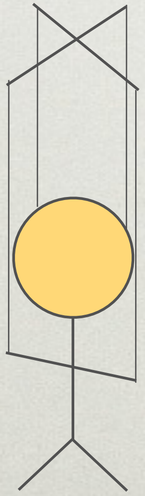
Actors

Messages





Routers



Actor System

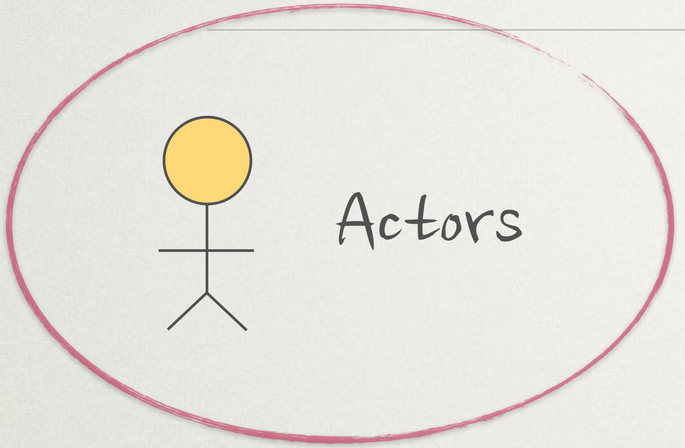
Mailbox

[1]

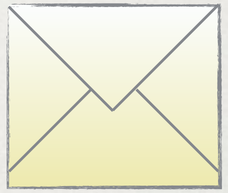
[1] By Dickelbers (Own work) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons
http://upload.wikimedia.org/wikipedia/commons/0/09/Sweden_postbox.JPG

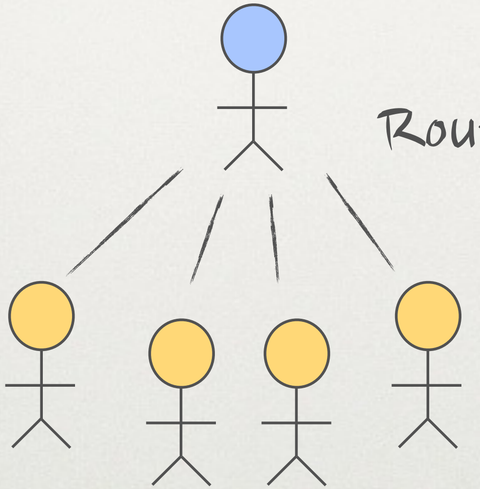
omega point.

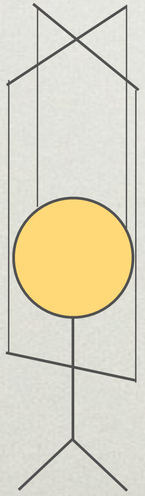
Akka in a Nutshell



Messages







Mailbox

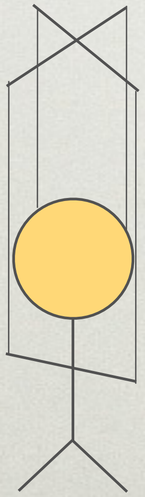
[1]

[1] By Dickelbers (Own work) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons
http://upload.wikimedia.org/wikipedia/commons/0/09/Sweden_postbox.JPG

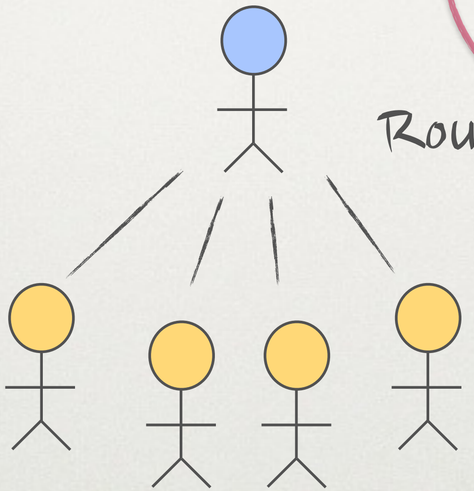
omega point.

Akka in a Nutshell

Actors

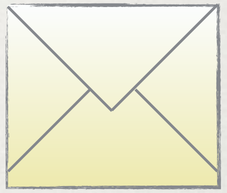


Actor System



Routers

Messages



Mailbox

[1]

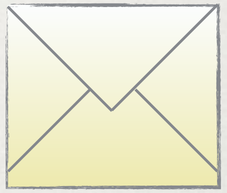
[1] By Dickelbers (Own work) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons
http://upload.wikimedia.org/wikipedia/commons/0/09/Sweden_postbox.JPG

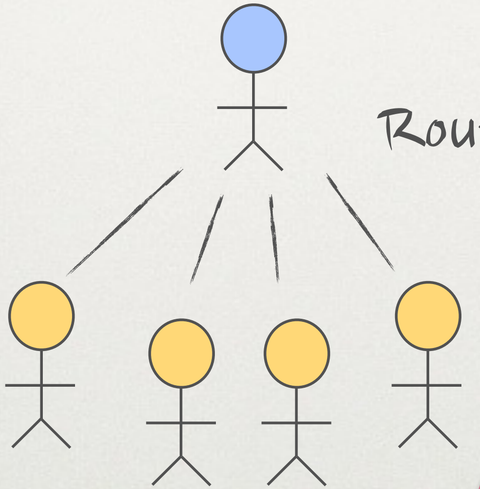
omega point.

Akka in a Nutshell

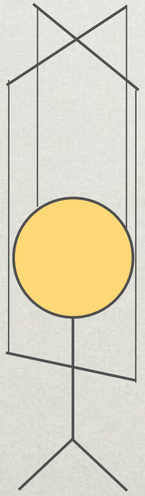
Actors

Messages





Routers



Actor System

Mailbox

[1]

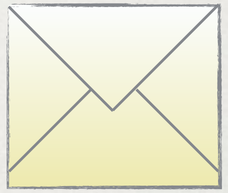
[1] By Dickelbers (Own work) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons
http://upload.wikimedia.org/wikipedia/commons/0/09/Sweden_postbox.JPG

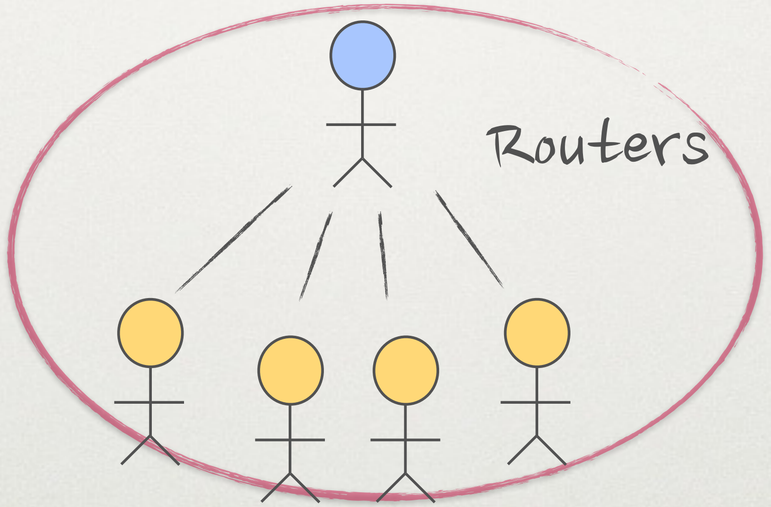
omega point.

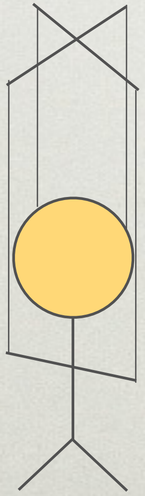
Akka in a Nutshell

Actors

Messages







Actor System

Mailbox

[1]

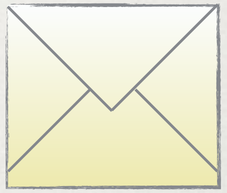
[1] By Dickelbers (Own work) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons
http://upload.wikimedia.org/wikipedia/commons/0/09/Sweden_postbox.JPG

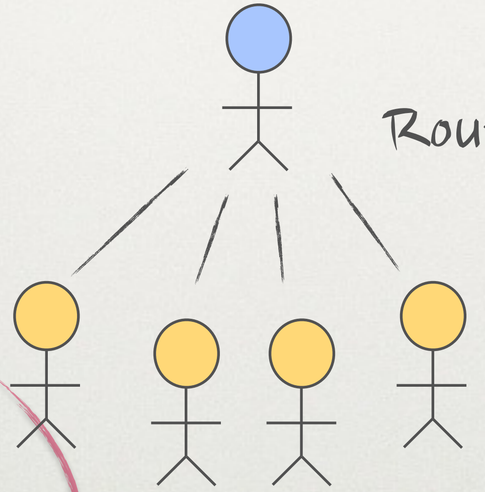
omega point.

Akka in a Nutshell

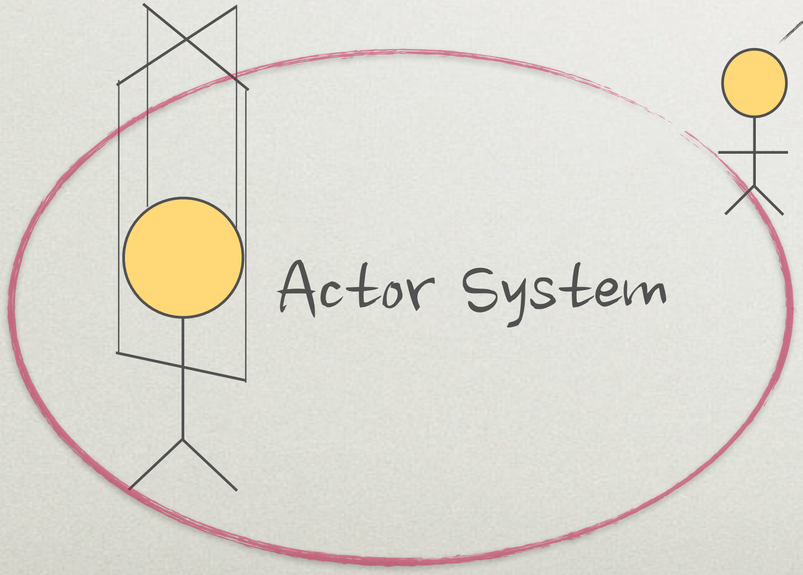
Actors

Messages





Routers



Actor System

Mailbox

[1]

[1] By Dickelbers (Own work) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons
http://upload.wikimedia.org/wikipedia/commons/0/09/Sweden_postbox.JPG

omega point.

Our definition of Legacy Code

Legacy \ˈle-gə-sē\

“: something that happened in the past or that comes from someone in the past”

- Merriam-Webster

omega point.

Our definition of Legacy Code

Legacy \ˈle-gə-sē\

": something that happened in the past or that comes from someone in the past"

- Merriam-Webster

Legacy Code \ˈle-gə-sē\ \ˈkōd\

": code that does not satisfy the characteristics of a reactive system"

- Deogun-Sawano

omega point.

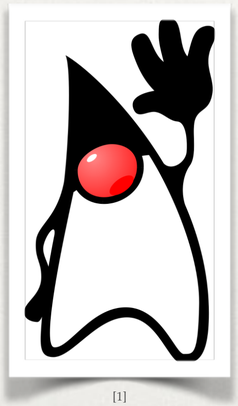
What is Legacy Code?

Characteristics of a reactive system, as defined by the reactive manifesto:

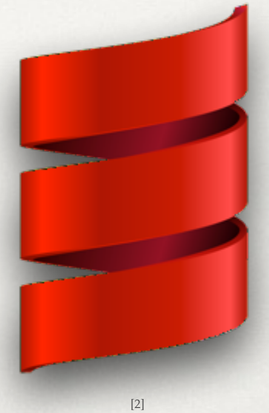
- responsive
- scalable
- resilient
- event-driven

reactivemanifesto.org

Java or Scala



[1]



[2]

I want to build an application with Akka, should I use Java or Scala?

Well, it depends...

[1] https://duke.kenai.com/wave/_Midsize/Wave.png.png
[2] http://www.scala-lang.org/

omega point.

Java or Scala

Assume we want to build a machine M to solve a problem P where,

- Efficiency is imperative
- Sequential computations shall be independent
- Implementation of M shall be platform independent
- Complexity and boilerplate code shall be reduced
- M's behavior shall be verifiable
- Time to Market is essential and risks minimized

omega point.

Java or Scala
Pros & Cons

Assume we want to build a machine M to solve a problem P where,

- ☐ Efficiency is imperative
- ☐ Sequential computations shall be independent
- ☐ Implementation of M shall be platform independent
- ☐ Complexity and boilerplate code shall be reduced
- ☐ M's behavior shall be verifiable
- ☐ Time to Market is essential and risks shall be minimized

Scoreboard

Java:

Scala:

omega point.

Java or Scala
Pros & Cons

Assume we want to build a machine M to solve a problem P where,

- [x] Efficiency is imperative
- [ ] Sequential computations shall be independent
- [ ] Implementation of M shall be platform independent
- [ ] Complexity and boilerplate code shall be reduced
- [ ] M's behavior shall be verifiable
- [ ] Time to Market is essential and risks shall be minimized

Scoreboard

Java: 1

Scala: 1

omega point.

Java or Scala
Pros & Cons

Assume we want to build a machine M to solve a problem P where,

- [x] Efficiency is imperative
- [x] Sequential computations shall be independent
- [ ] Implementation of M shall be platform independent
- [ ] Complexity and boilerplate code shall be reduced
- [ ] M's behavior shall be verifiable
- [ ] Time to Market is essential and risks shall be minimized

Scoreboard

Java: 11

Scala: 11

omega point.

Java or Scala
Pros & Cons

Assume we want to build a machine M to solve a problem P where,

- [x] Efficiency is imperative
- [x] Sequential computations shall be independent
- [x] Implementation of M shall be platform independent
- [ ] Complexity and boilerplate code shall be reduced
- [ ] M's behavior shall be verifiable
- [ ] Time to Market is essential and risks shall be minimized

Scoreboard

Java: III

Scala: III

omega point.

Java or Scala
Pros & Cons

Assume we want to build a machine M to solve a problem P where,

- [x] Efficiency is imperative
- [x] Sequential computations shall be independent
- [x] Implementation of M shall be platform independent
- [x] Complexity and boilerplate code shall be reduced
- [ ] M's behavior shall be verifiable
- [ ] Time to Market is essential and risks shall be minimized

Scoreboard

Java: III

Scala: IIII

omega point.

Java or Scala
Pros & Cons

Assume we want to build a machine M to solve a problem P where,

- [x] Efficiency is imperative
- [x] Sequential computations shall be independent
- [x] Implementation of M shall be platform independent
- [x] Complexity and boilerplate code shall be reduced
- [x] M's behavior shall be verifiable
- [ ] Time to Market is essential and risks shall be minimized

Scoreboard

Java: IIII

Scala: ~~IIII~~

omega point.

Java or Scala
Pros & Cons

Assume we want to build a machine M to solve a problem P where,

- [x] Efficiency is imperative
- [x] Sequential computations shall be independent
- [x] Implementation of M shall be platform independent
- [x] Complexity and boilerplate code shall be reduced
- [x] M's behavior shall be verifiable
- [x] Time to Market is essential and risks shall be minimized

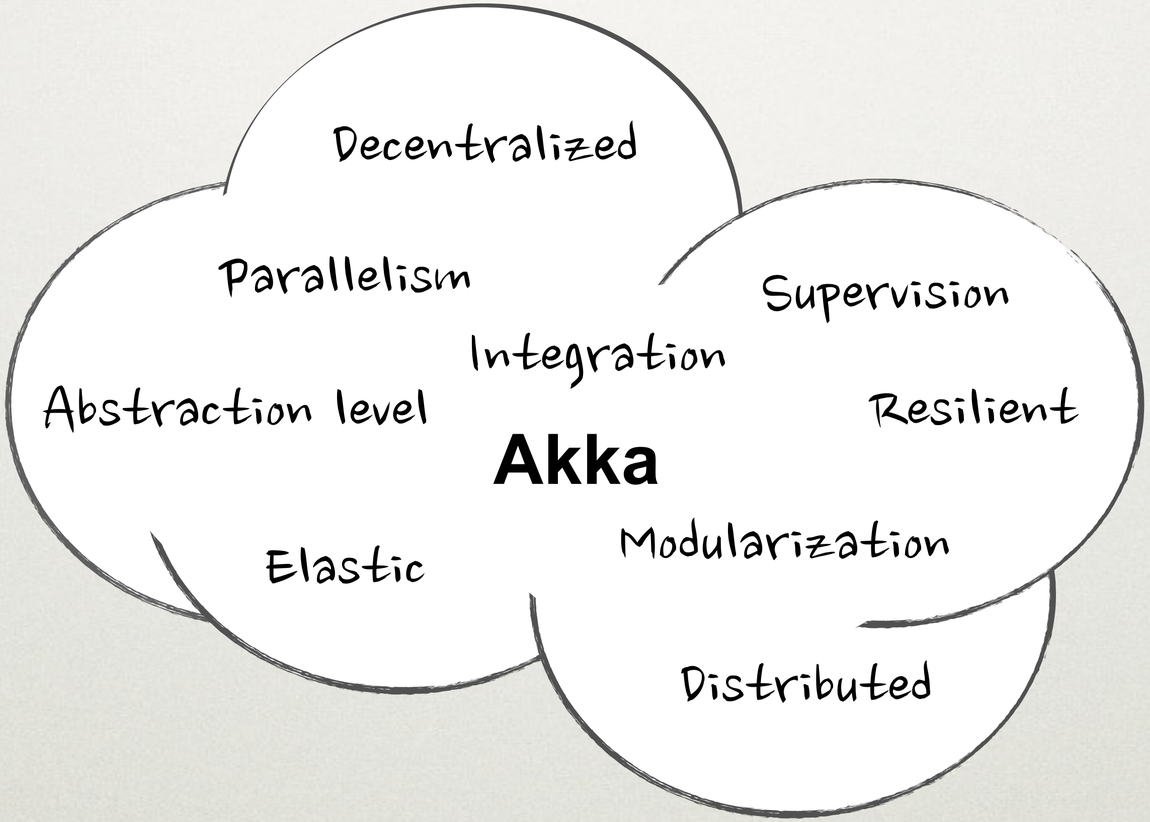
omega point.

Java or Scala Conclusion

- Both Java and Scala works well with Akka
- Choose the language that makes most sense
- Don't add unnecessary risk

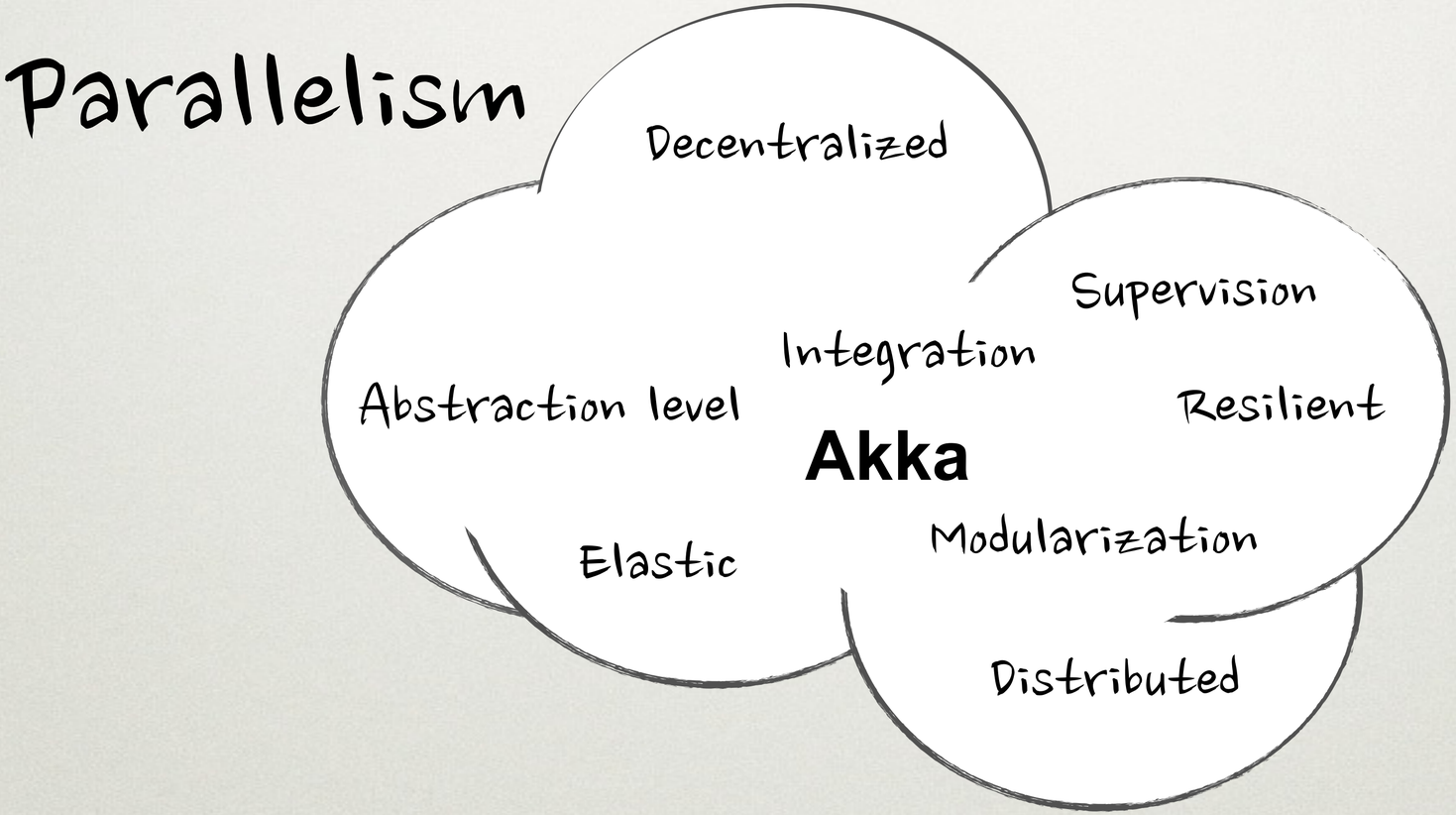
omega point.

Akka
All or nothing?



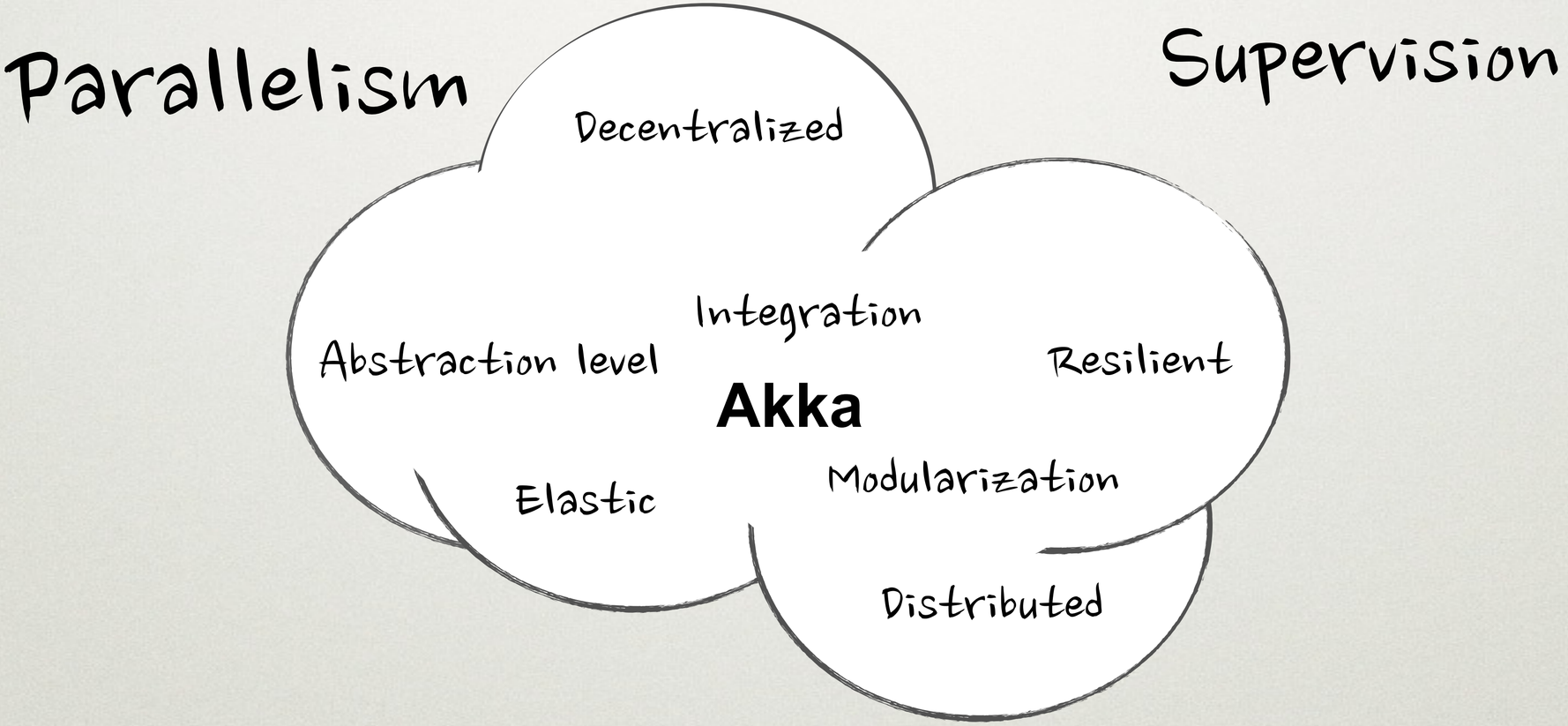
omega point.

Akka
All or nothing?



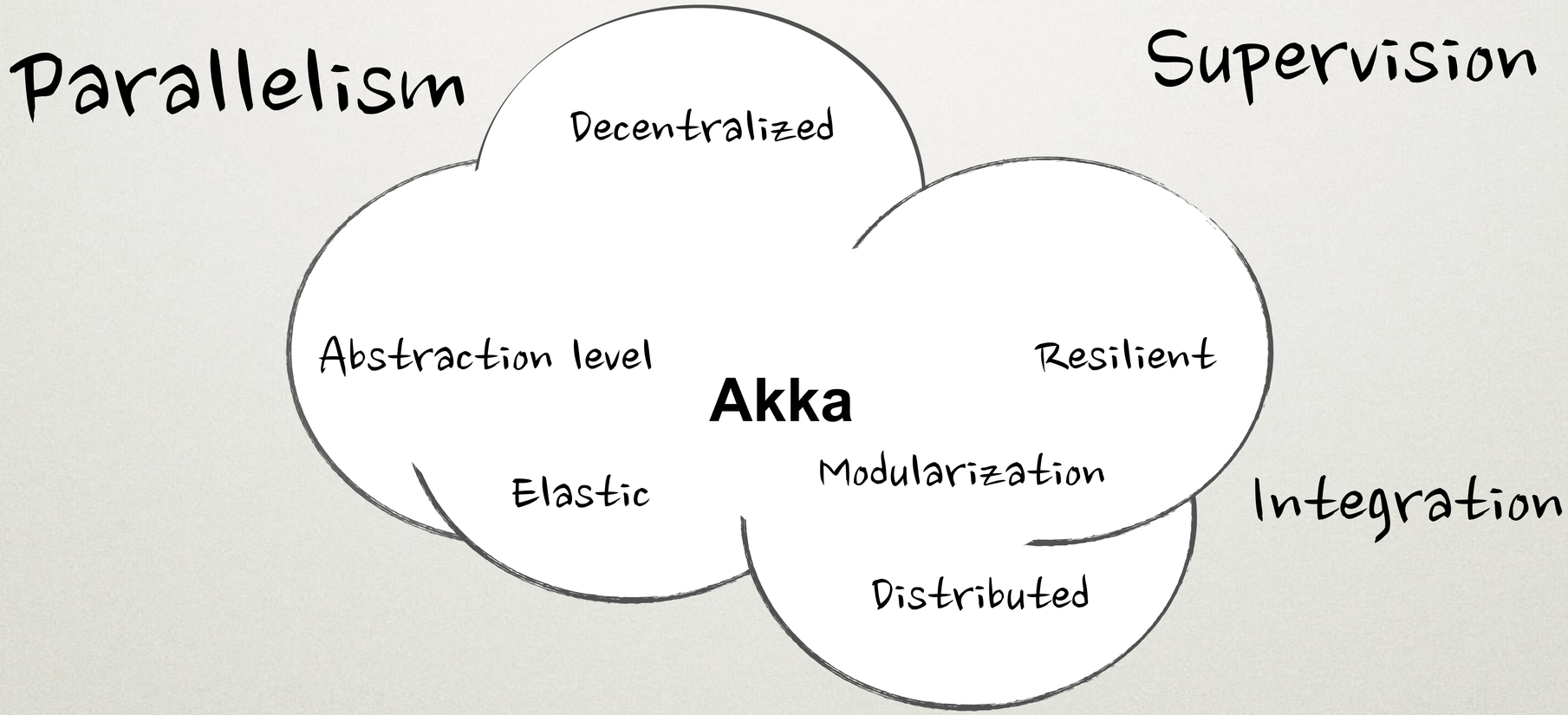
omega point.

Akka
All or nothing?



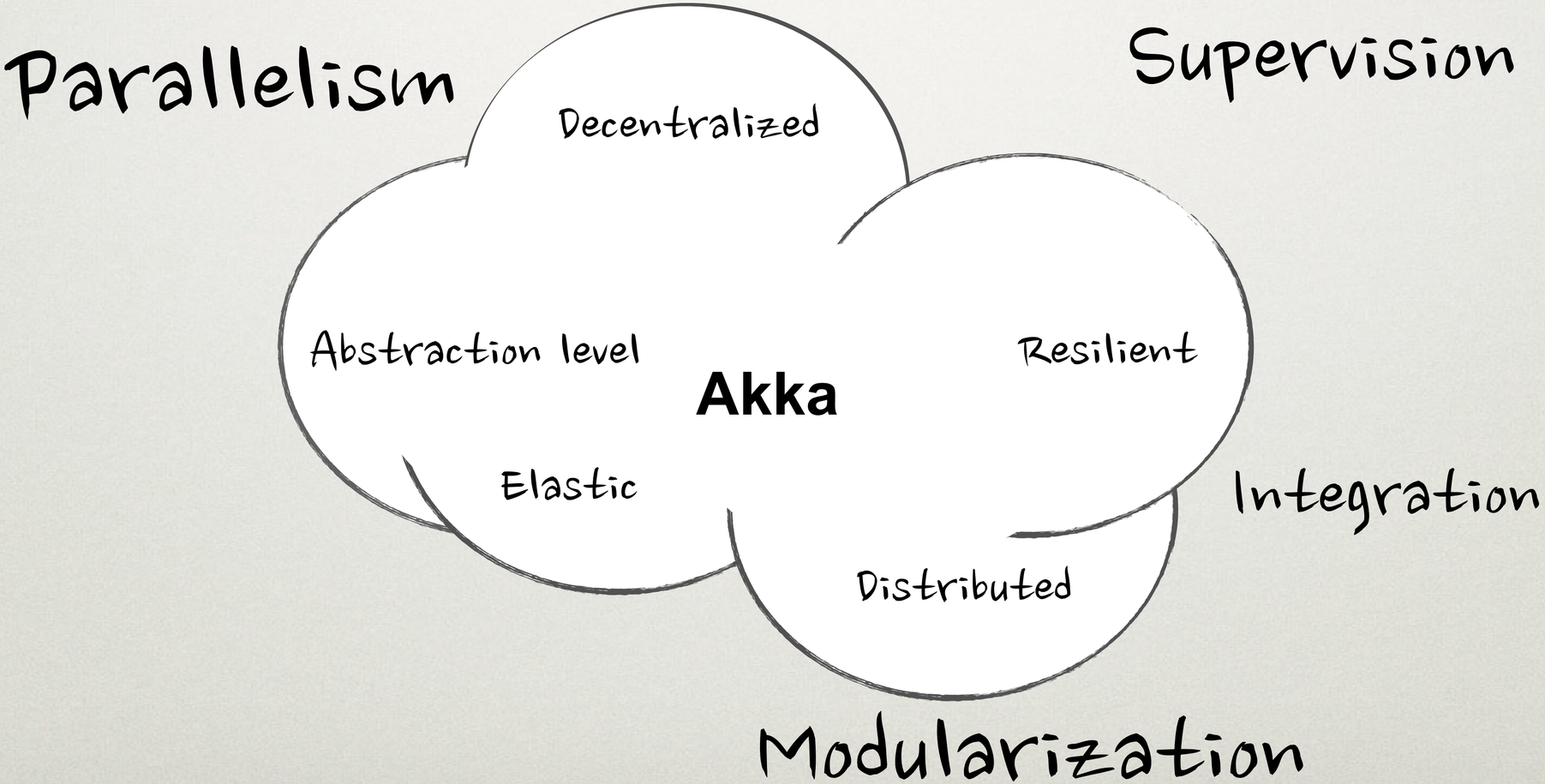
omega point.

Akka
All or nothing?



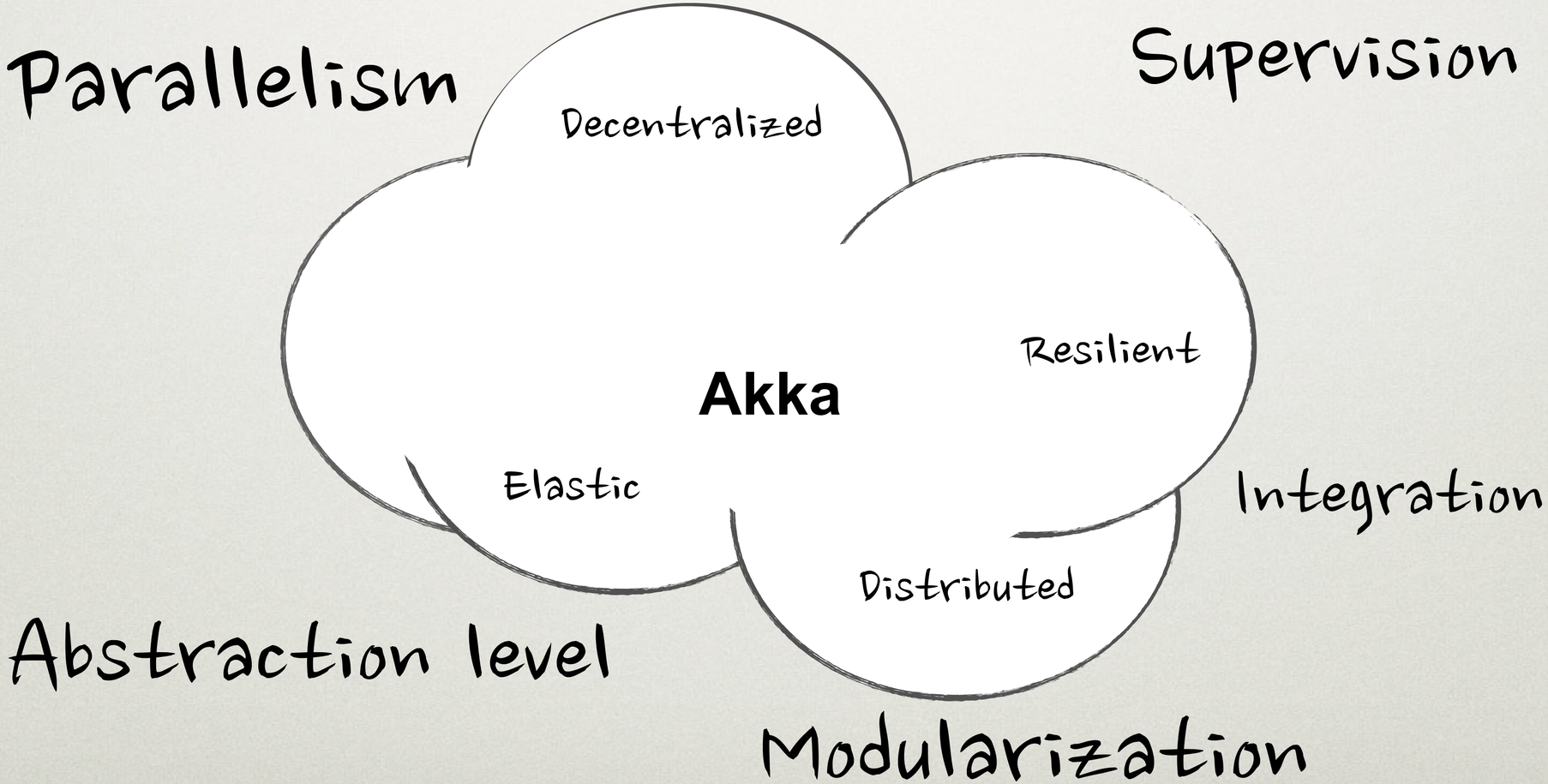
omega point.

Akka
All or nothing?



omega point.

Akka
All or nothing?



omega point.

Domain Specific Requirements

- Akka is more or less a perfect match for all parallelizable domains

- But what about
  - reactive domains with blocking parts?
  - legacy domains with reactive parts?

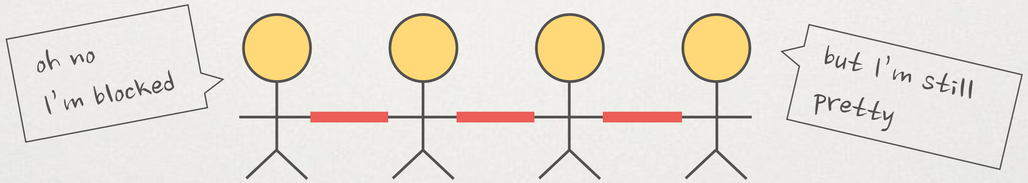
[1]

[1] By Renesis (Own work) [CC-BY-SA-2.5 (http://creativecommons.org/licenses/by-sa/2.5)], via Wikimedia Commons. http://upload.wikimedia.org/wikipedia/commons/1/19/Stop2.png

omega point.

Blocking in a Reactive Environment

- If we choose Akka, we need to block actors, yuck!



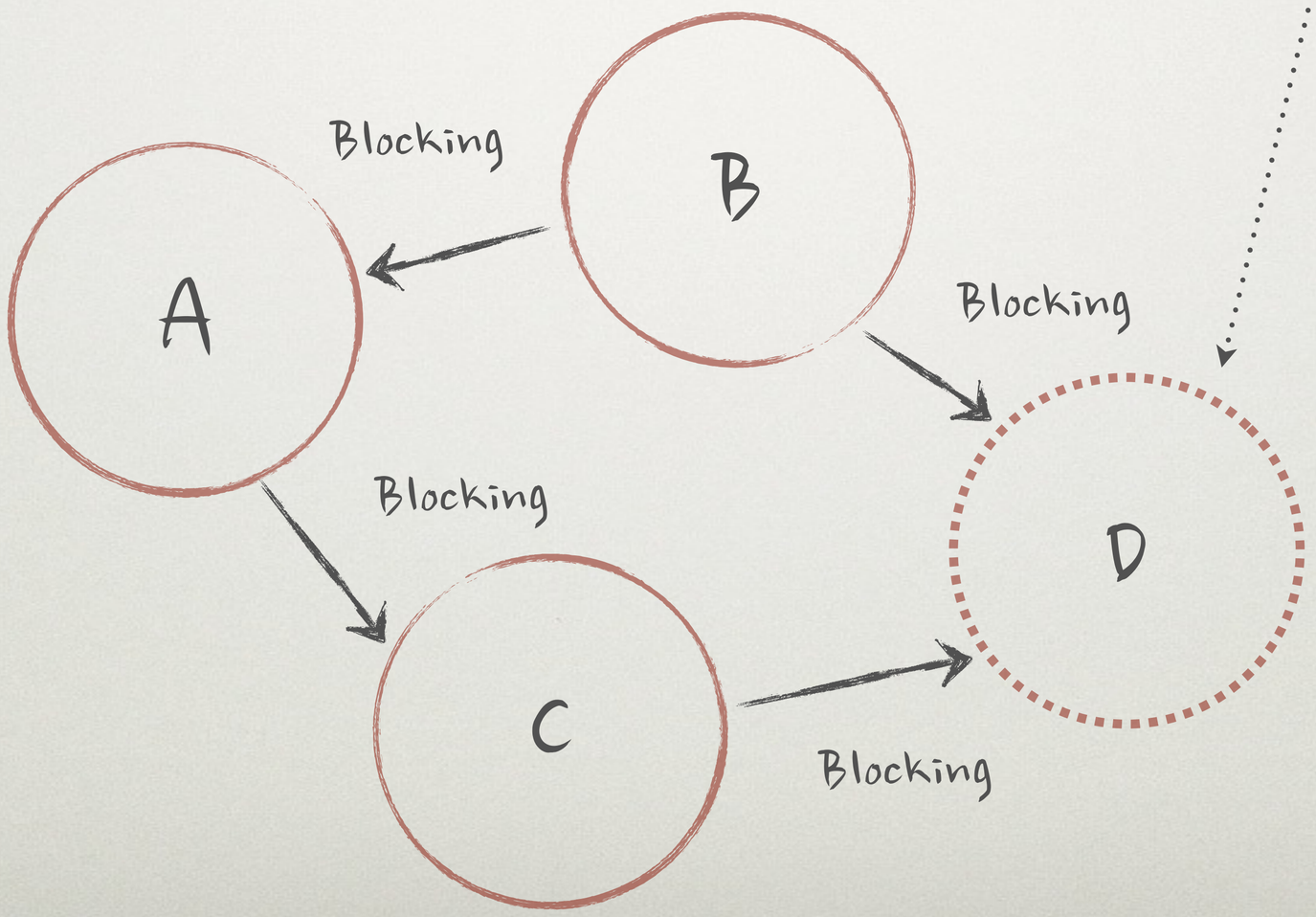
- The main advantage is that we can reuse actors from parallelizable parts but are there any downsides?

- The other option is to use legacy design

omega point.

Reactive in a Legacy Environment?

Is it possible to replace D by a component implemented with Akka?



omega point.

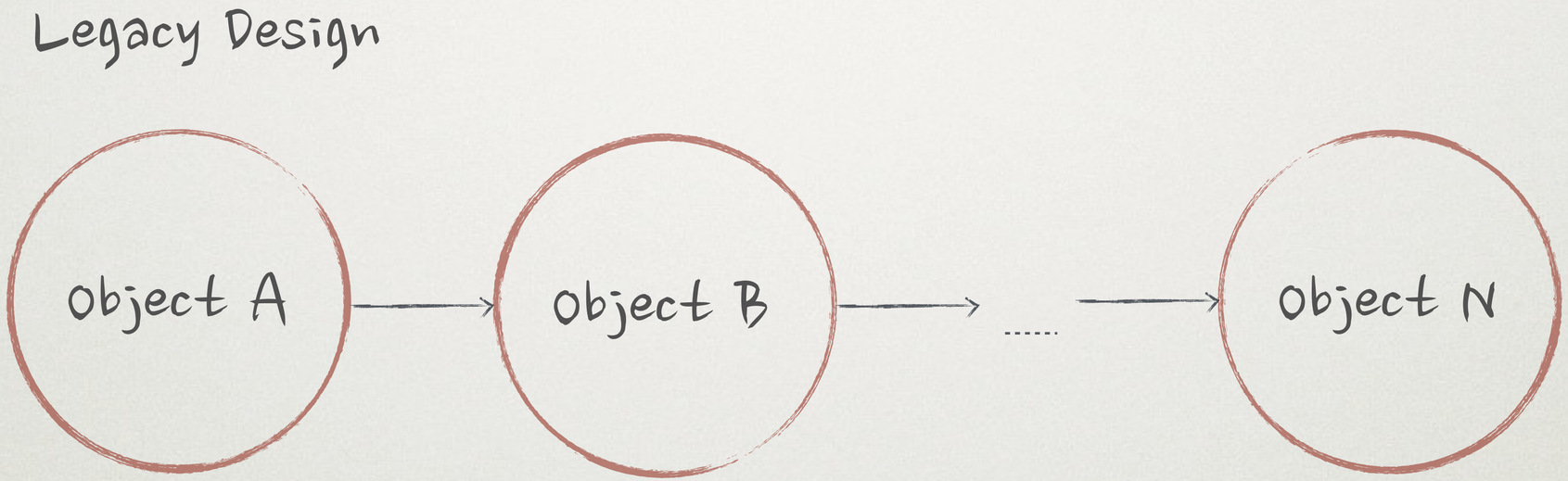
Debugger
Friend or Foe?

We often get the question:

"The asynchrony in Akka makes it very hard to use the debugger, Am I doing it wrong?"

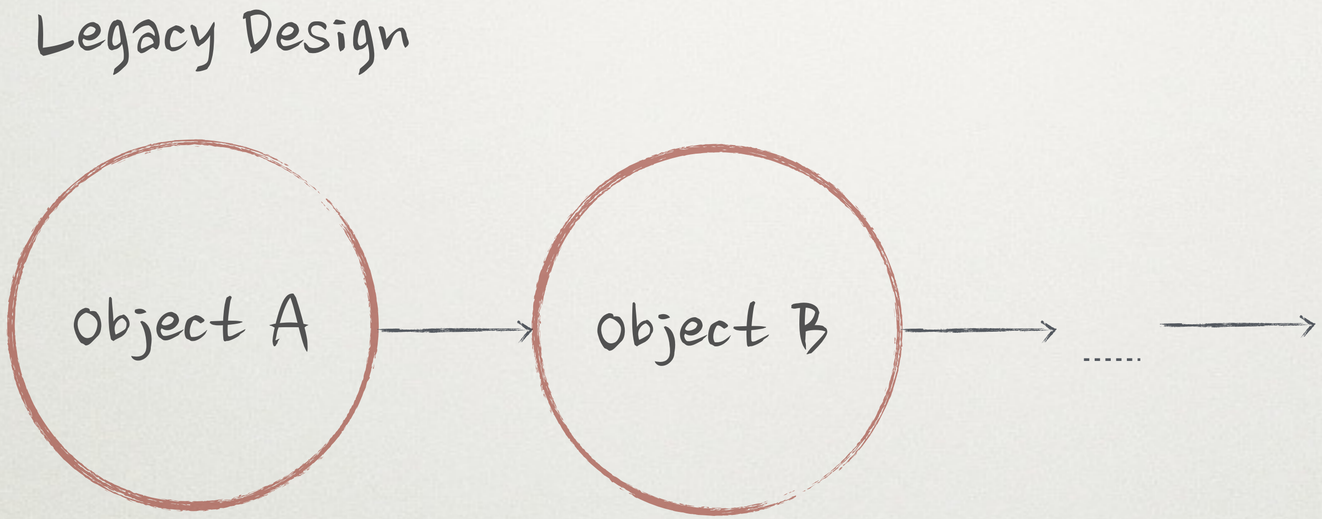
omega point.

Debugger in Legacy Design



omega point.

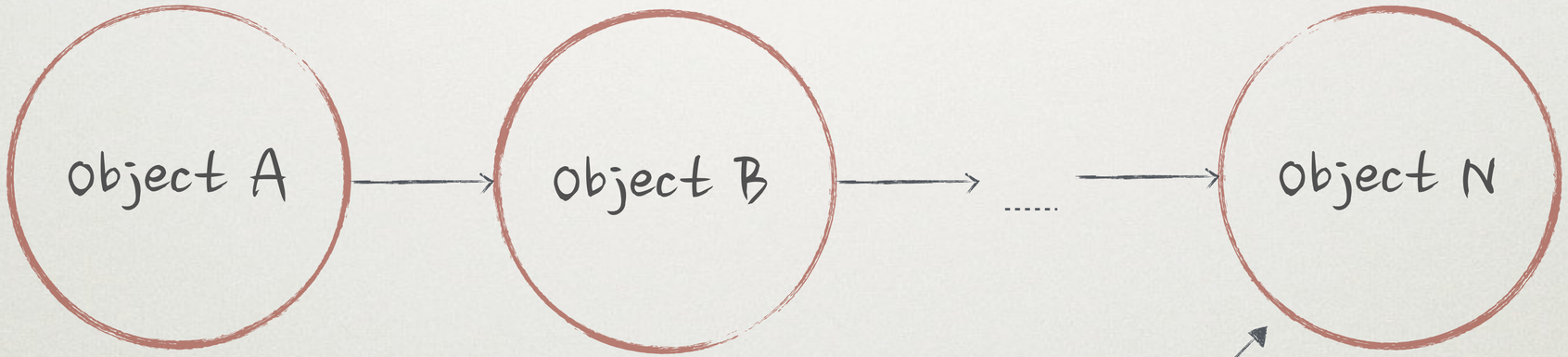
Debugger in Legacy Design



omega point.

Debugger in Legacy Design

Legacy Design

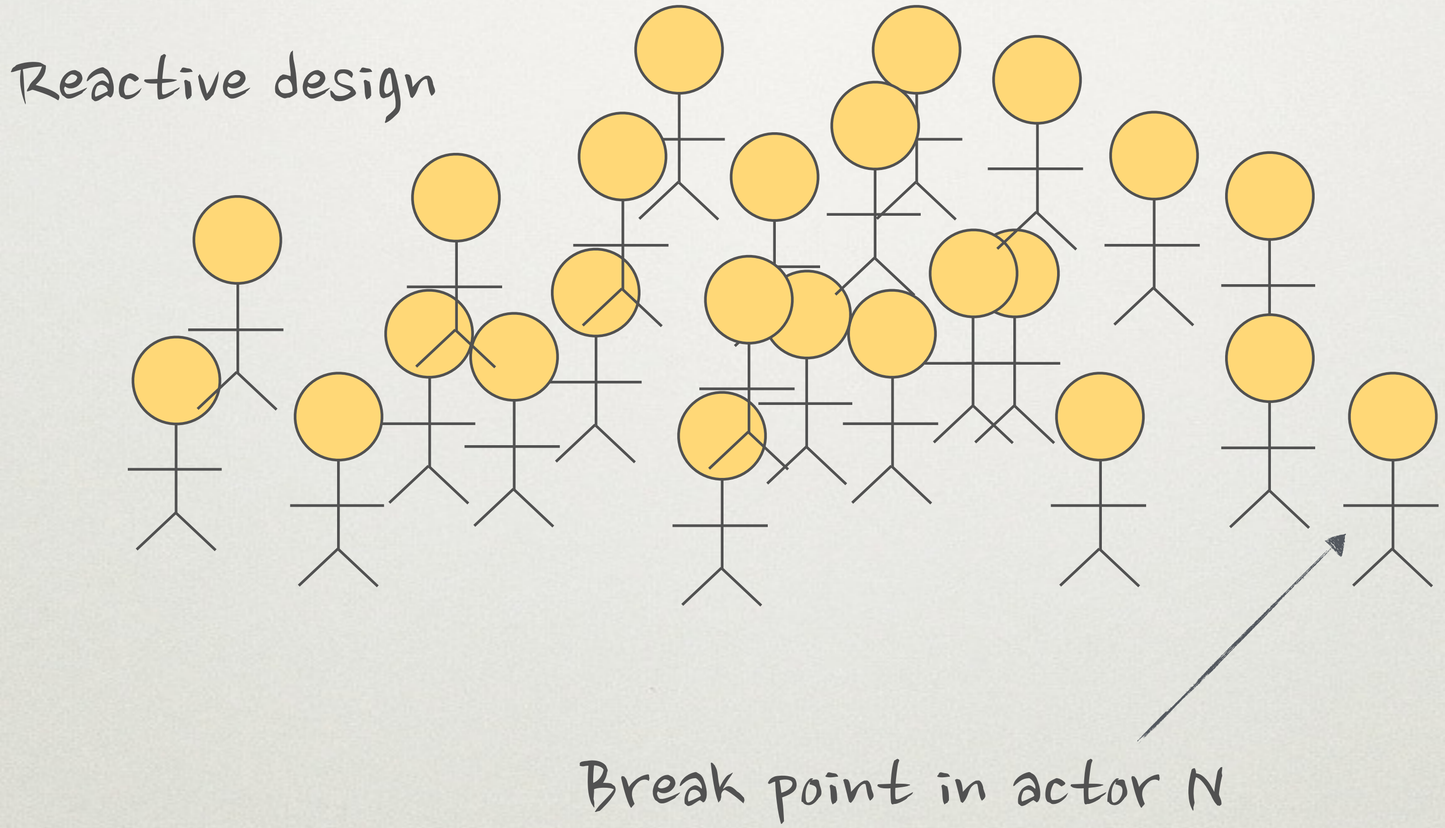


Place a break point in N to find out

- Which value caused the crash?
- Who created it?

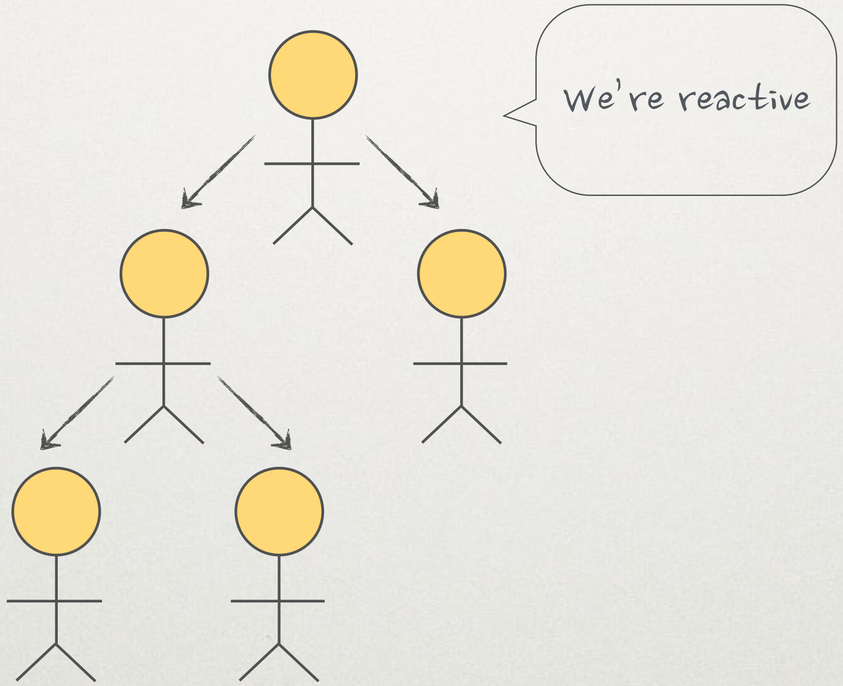
omega point.

Debugger in Reactive Design



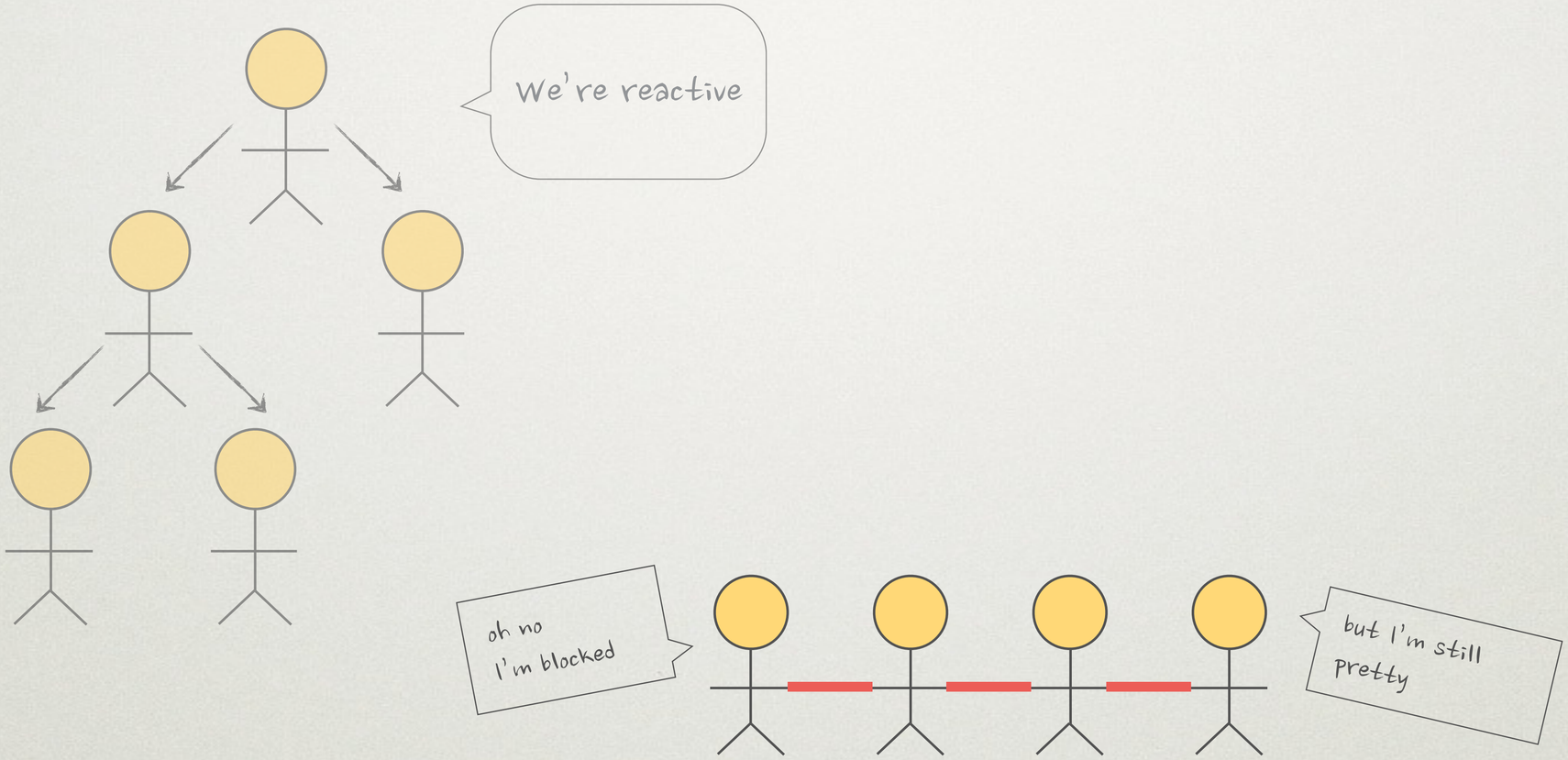
omega point.

Supervision



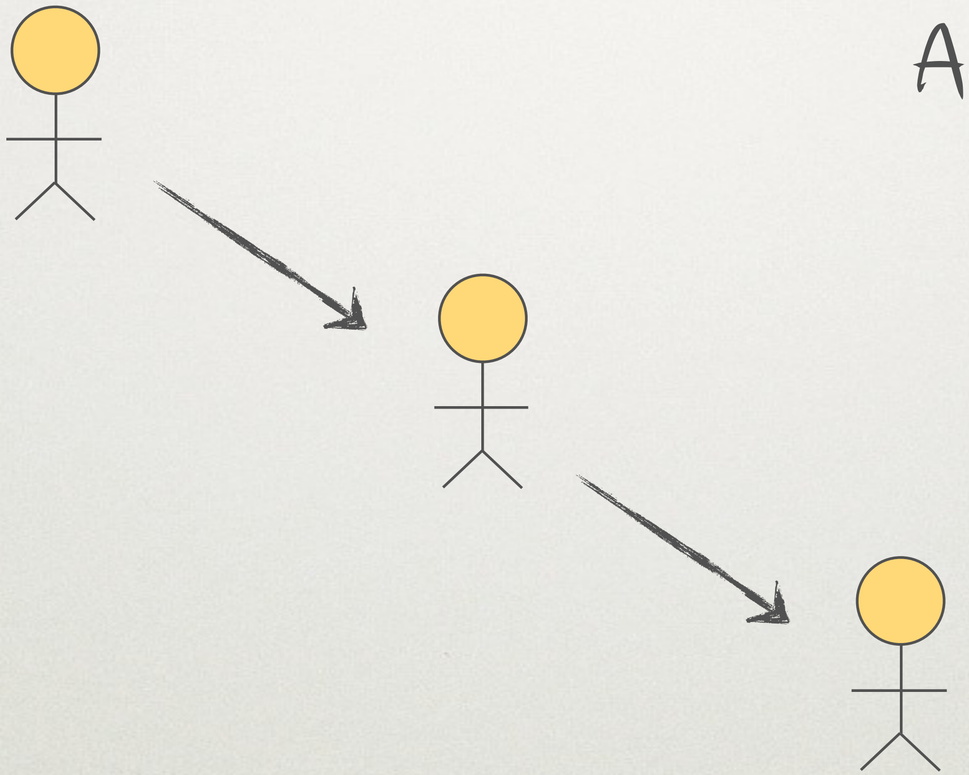
omega point.

Supervision



omega point.

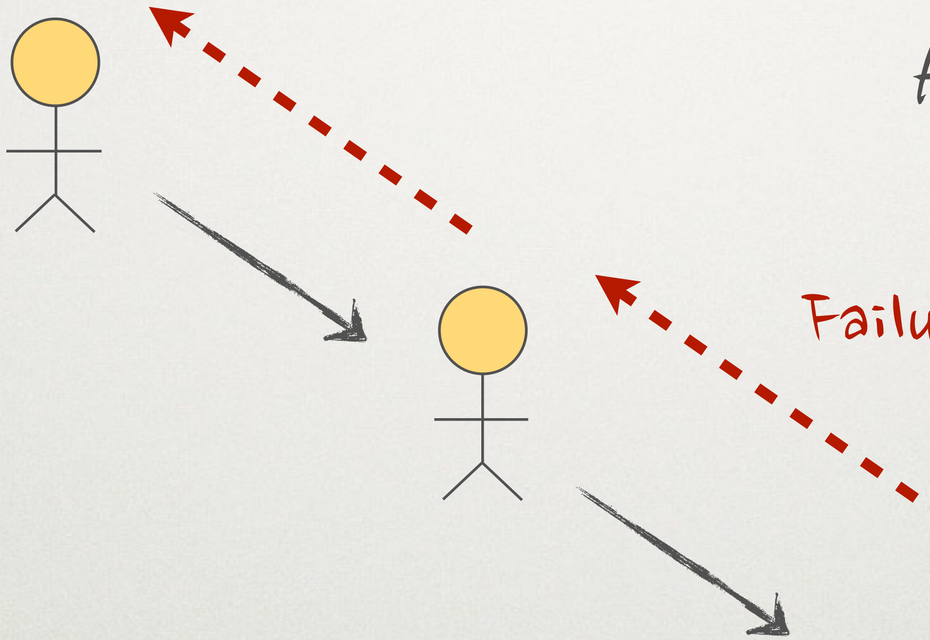
Supervision



Asynchronous

omega point.

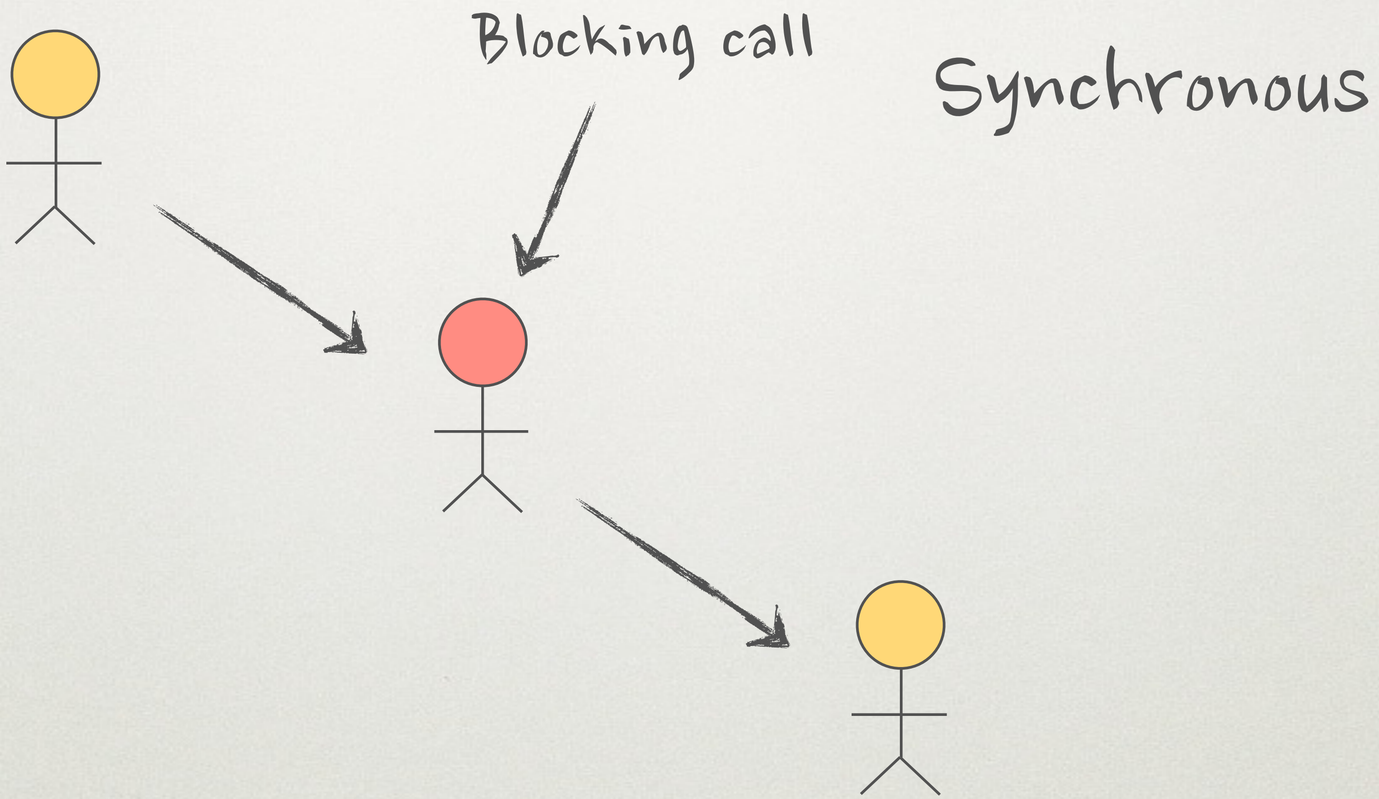
Supervision



Asynchronous

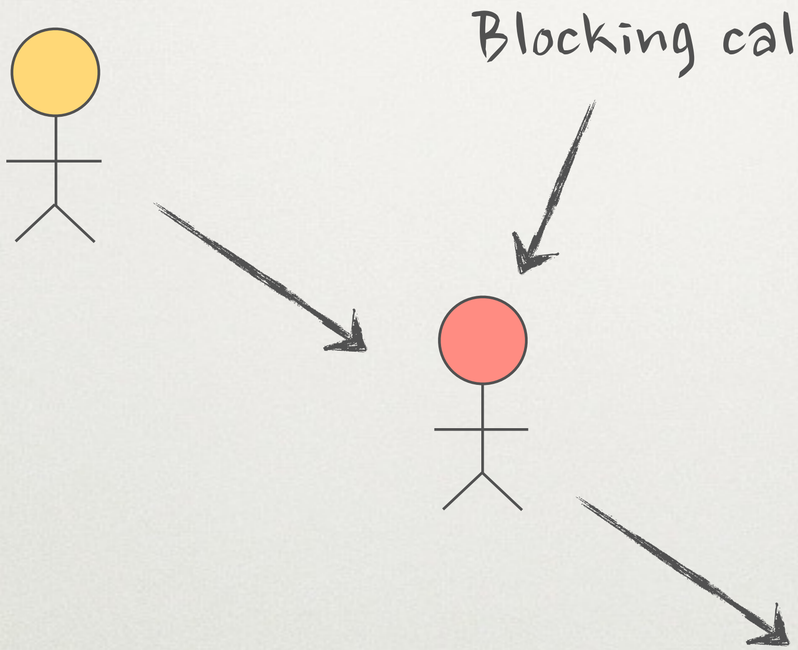
omega point.

Supervision



omega point.

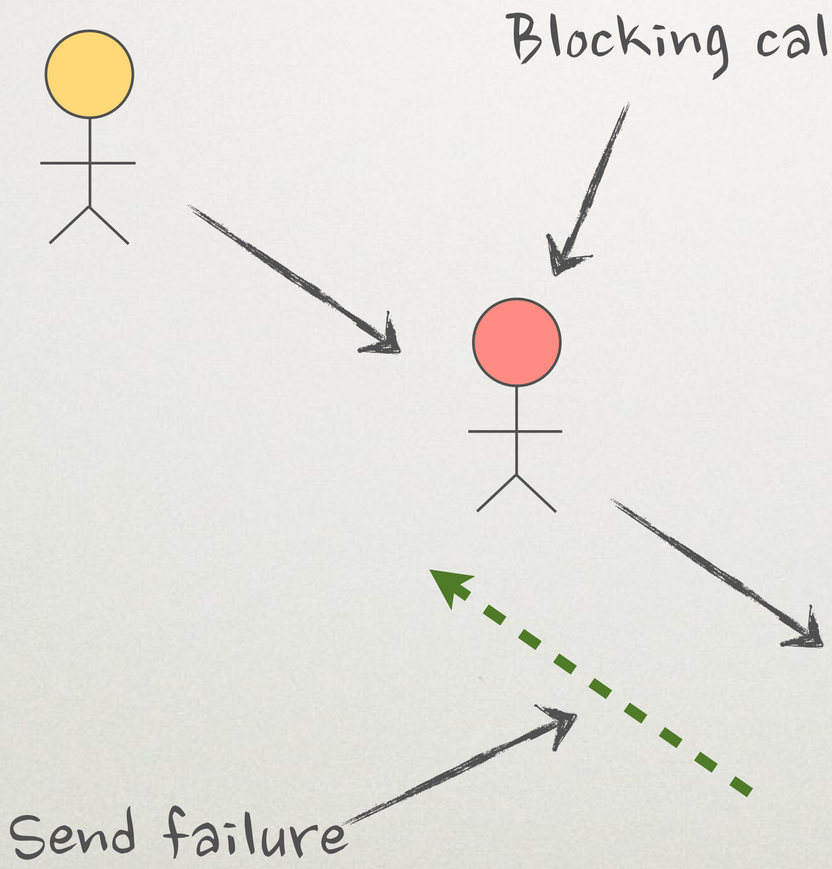
Supervision



Synchronous

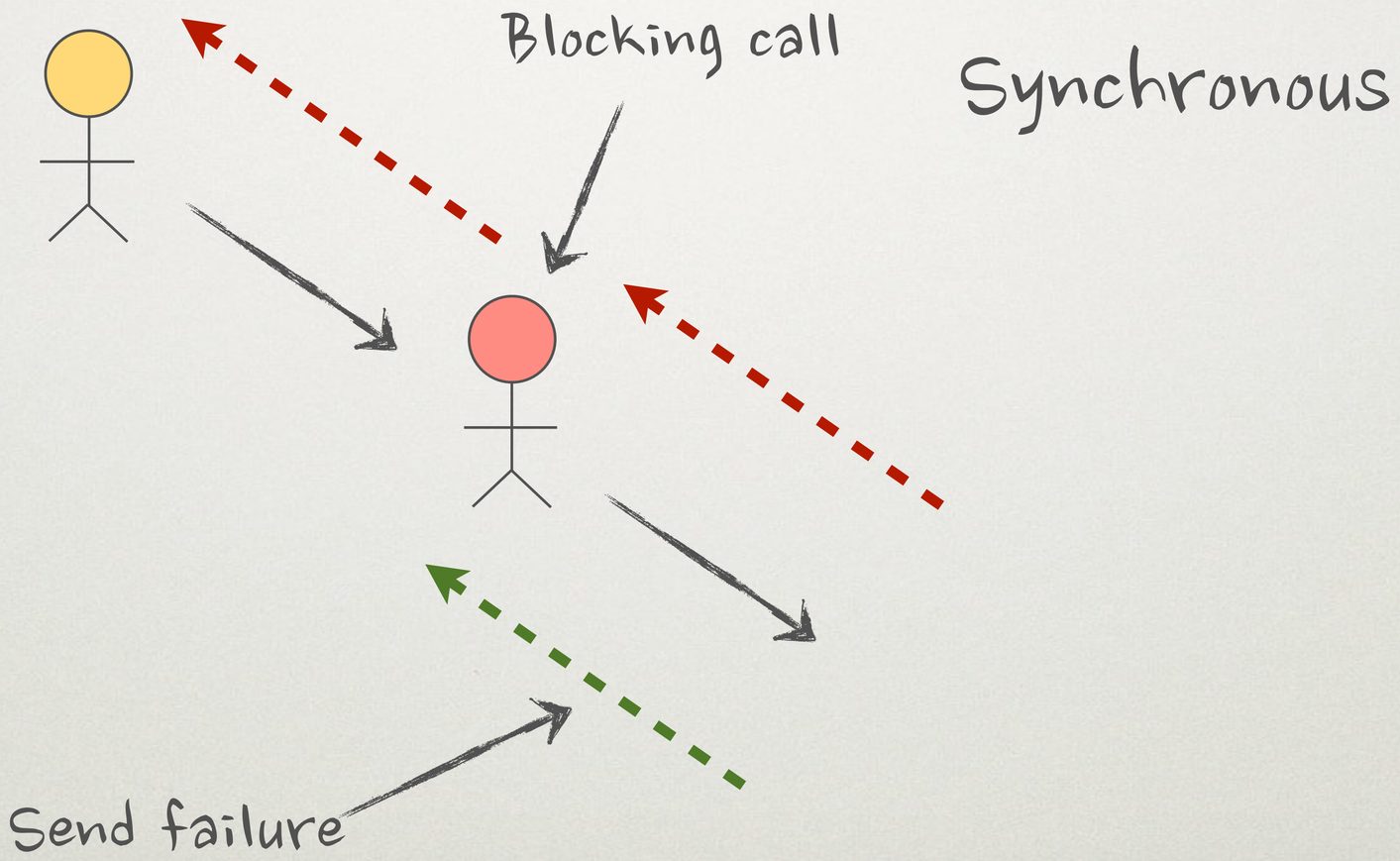
omega point.

Supervision



omega point.

Supervision



omega point.

Supervision

```
    private ActorRef targetActor;
    private ActorRef caller;
    private AskParam askParam;
    private Cancellable timeoutMessage;

    @Override
    public SupervisorStrategy supervisorStrategy() {
        return new OneForOneStrategy(0, Duration.Zero(), new Function<Throwable, SupervisorStrategy.Directive>() {
            public SupervisorStrategy.Directive apply(Throwable cause) {
                caller.tell(new Failure(cause), self());
                return SupervisorStrategy.stop();
            }
        });
    }

    @Override
    public void onReceive(final Object message) throws Exception {
        if (message instanceof AskParam) {
            askParam = (AskParam) message;
            caller = sender();
            targetActor = context().actorOf(askParam.props);
            context().watch(targetActor);
            targetActor.forward(askParam.message, context());
            final Scheduler scheduler = context().system().scheduler();
            timeoutMessage = scheduler.scheduleOnce(askParam.timeout.duration(), self(), new AskTimeout(), context().dispatcher(),
null);
        }
        else if (message instanceof Terminated) {
            sendFailureToCaller(new ActorKilledException("Target actor terminated."));
            timeoutMessage.cancel();
            context().stop(self());
        }
        else if (message instanceof AskTimeout) {
            sendFailureToCaller(new TimeoutException("Target actor timed out after " + askParam.timeout.toString()));
            context().stop(self());
        }
        else {
            unhandled(message);
        }
    }

    private void sendFailureToCaller(final Throwable t) {
        caller.tell(new Failure(t), self());
    }
```

omega point.

Supervision

- Write legacy code
- Use Akka actors

omega point.

Supervision

- Write legacy code
- Use Akka actors

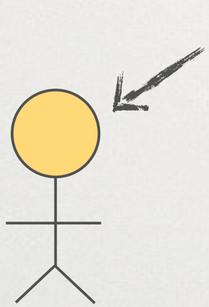
Sequential does not imply synchronicity

The Shared Mutable State Trap

Keeping state between messages in an actor is extremely dangerous because it may cause a shared mutable state

omega point.

The Shared Mutable State Trap



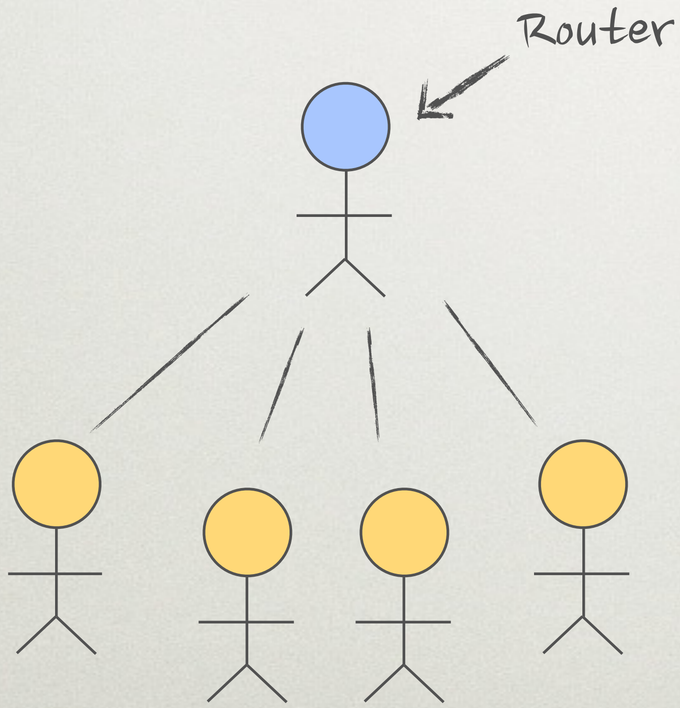
Properties

- One message queue
- Only one message is processed at a time
- Messages are processed in order received
- An actor may choose to divide and conquer a task by calling other actors

omega point.

The Shared Mutable State Trap

Actors may be grouped in a router



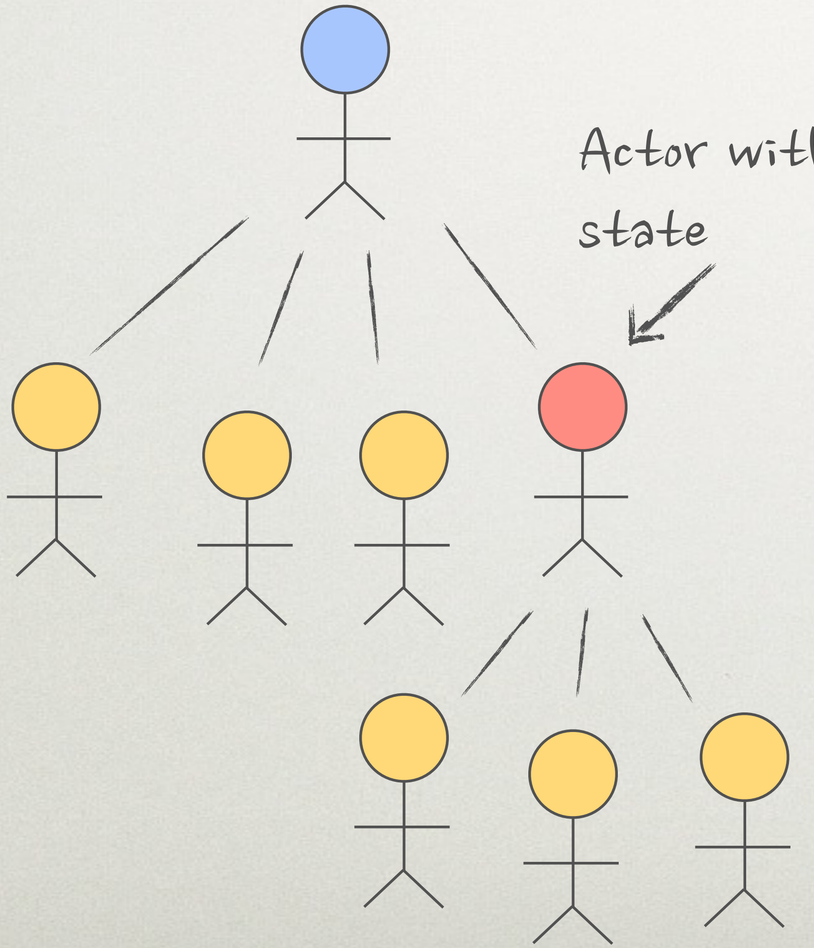
Properties

- Messages are distributed among actors according to some message delivery scheme, eg Round Robin or Smallest Mailbox

- A router may receive a lot of messages

omega point.

The Shared Mutable State Trap



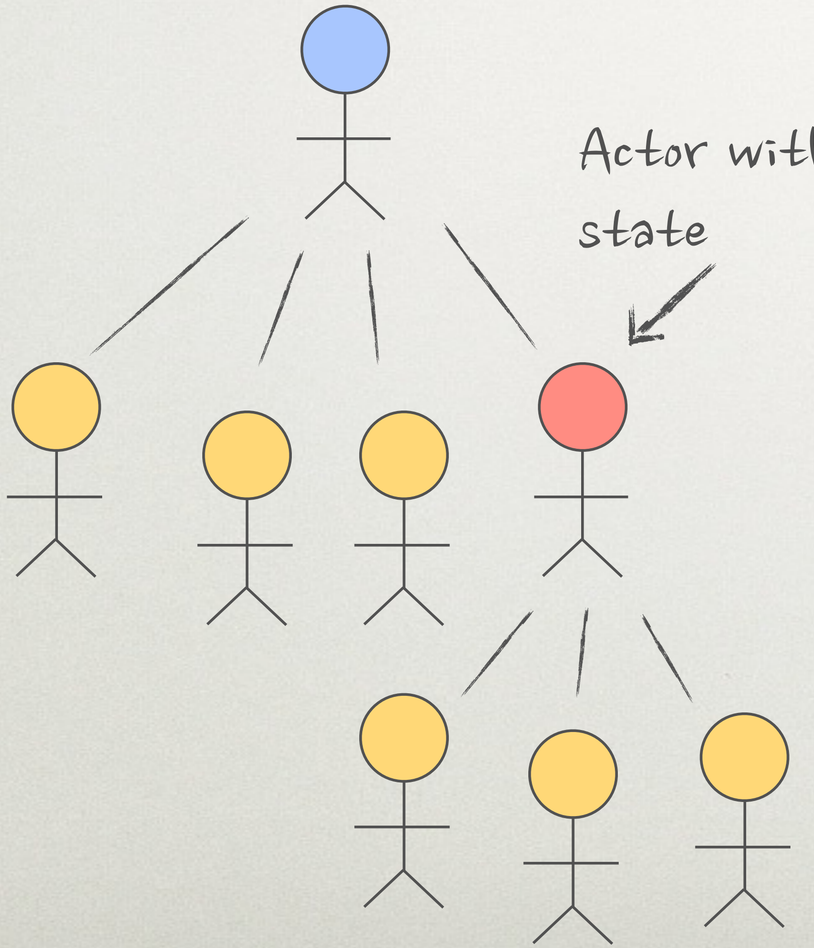
Scenario

- A stateful actor is part of a router

- It uses divide and conquer to solve its task

omega point.

The Shared Mutable State Trap



Scenario

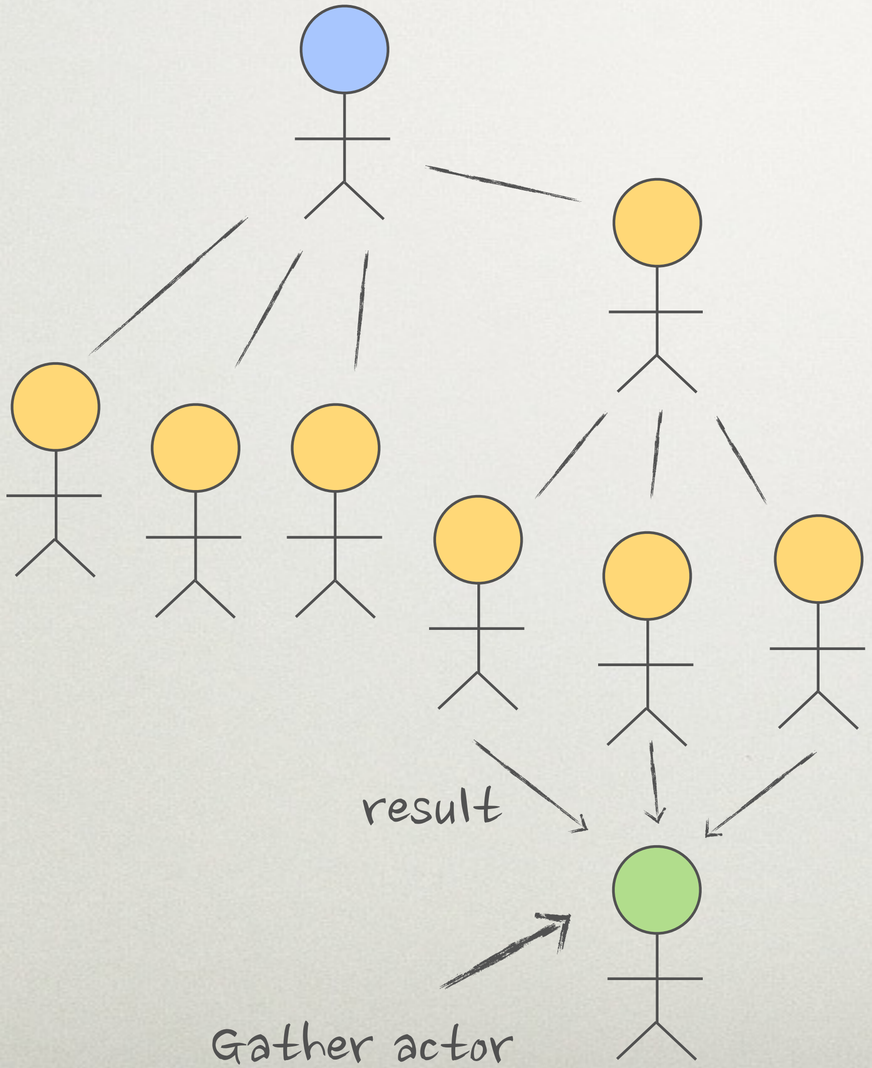
- Each message recived from the router resets the state

- New messages are inter mixed with child responses

- Hence, we have a shared mutable state

omega point.

The Shared Mutable State Trap



Solution

- Each request sent by the router results in a "Gather" actor

- The Gather actor is responsible for collecting the result

- A Gather actor is NEVER reused

omega point.

Key Take-Aways

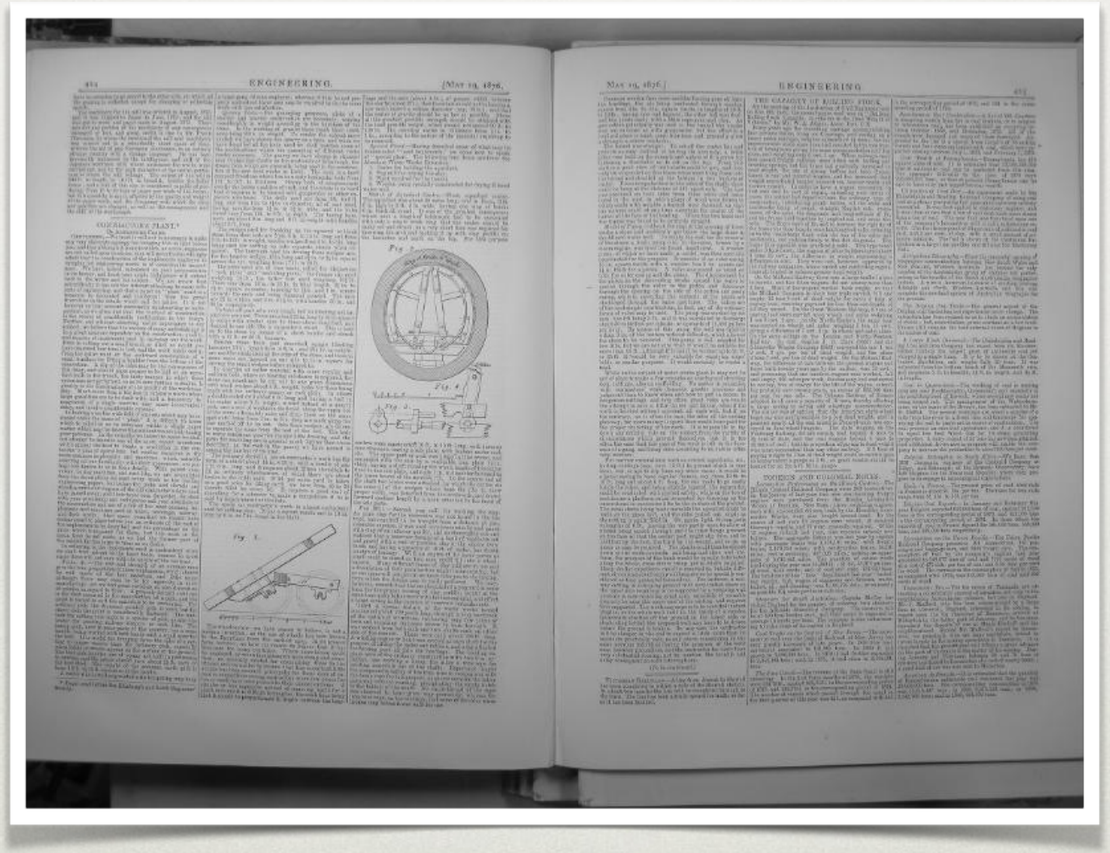
- It's very easy to accidentally fall back to sequential "thinking" with state

- Often one does not realize this until the system is placed under heavy load

- Try to avoid state!

omega point.

Readability



By Various. Edited by: W H Maw and J Dredge (Abandoned library clearance. Self photographed.) [Public domain], via Wikimedia Commons

omega point.

Implications of untyped actors

Akka actors are untyped – by design

```
public void onReceive(Object message)
```

Implications of untyped actors

Akka actors are untyped – by design

```
public void onReceive(Object message)
```

- Readability
- No support from compiler/IDE

omega point.

Message Handling V 1.0

```
@Override
public void onReceive(Object message) {
    if (message instanceof SomeMessage) {
        doStuff();
    }
    else {
        unhandled(message);
    }
}
```

omega point.

V 1.0 - If-else contd.

```
@Override
public void onReceive(Object message) {
    if (message instanceof SomeMessage) {
        doStuff();
    }
    else if (message instanceof SomeOtherMessage) {
        doSomeOtherStuff();
    }
    else {
        unhandled(message);
    }
}
```

omega point.

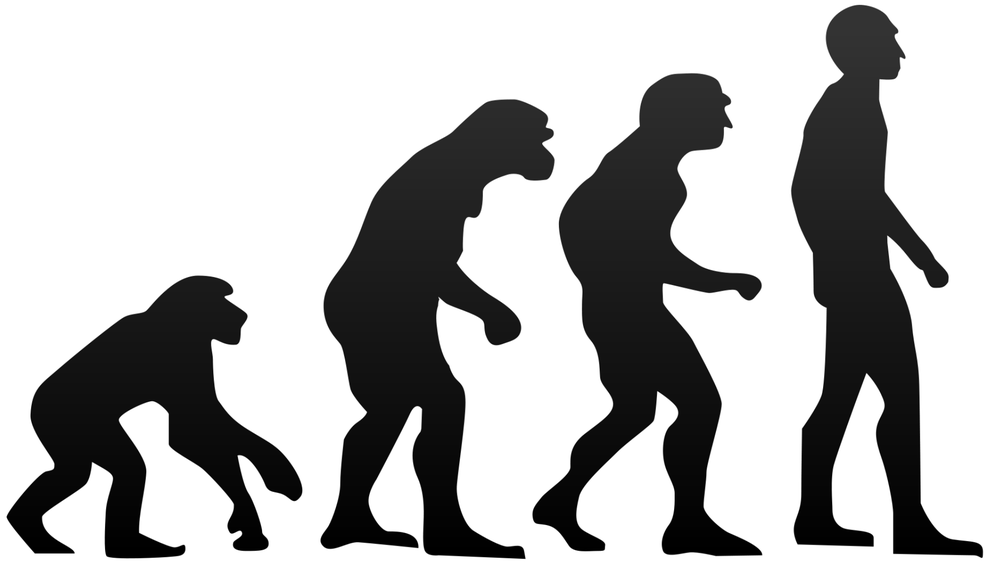
V 1.0 - If-else mess

```
@Override
public void onReceive(Object message) {
    if (message instanceof SomeMessage) {
        doStuff();
    }
    else if (message instanceof SomeOtherMessage) {
        doSomeOtherStuff();
    }
    else if (message instanceof YetAnotherMessage) {
        doEvenMoreStuff();
    }
    else {
        unhandled(message);
    }
}
```

omega point.

Evolution of message handling

```
@Override
public void onReceive(Object message) {
    ...
}
```



http://upload.wikimedia.org/wikipedia/commons/thumb/6/69/Human_evolution.svg/2000px-Human_evolution.svg.png

omega point.

V 2.0 - Overloading

```
public void onMessage(SomeMessage message) {...}

public void onMessage(SomeOtherMessage message) {...}
```

omega point.

V 2.0 - Overloading

```
public void onMessage(SomeMessage message) {...}

public void onMessage(SomeOtherMessage message) {...}
```

```
@Override
public void onReceive(Object message) {
    ...
    methods.get(message.getClass()).invoke(this, message);
    ...
}
```

omega point.

V 2.1 - Annotations

```
class Worker extends BaseActor {

    @Response
    public void onMessage(SomeMessage message) {...}


    public void onMessage(SomeOtherMessage message) {...}
}
```

omega point.

V 3.0 - Contract

```
interface Messages {
    void doSomething(SomeMessage message);
    void doSomethingElse(SomeOtherMessage message);
}
```

omega point.

V 3.0 - Contract

```
interface Messages {
    void doSomething(SomeMessage message);
    void doSomethingElse(SomeOtherMessage message);
}
```

```
SomeActor extends BaseActor implements Messages {...}
```

omega point.

V 3.0 - Contract

```
class Worker extends BaseActor implements Messages {

   public void handleResponse(SomeMessage message) {...}

   public void handleRequest(SomeOtherMessage message) {...}
}
```

omega point.

V 3.0 - Contract

```
BaseActor extends UntypedActor {

    BaseActor() {
        methodDelegate = new MethodDelegate(this);
    }

    @Override
    public void onReceive(Object message) {
        if (methodDelegate.onReceive(message)) {
            return;
        }
        unhandled(message);
    }
}
```

omega point.

Before

```
class Worker extends BaseActor implements Messages {
    @Override
    public void onReceive(Object message) {
        if (message instanceof SomeMessage) {
            doStuff();
        }
        else if (message instanceof SomeOtherMessage) {
            doSomeOtherStuff();
        }
        else if (message instanceof YetAnotherMessage) {
            doEvenMoreStuff();
        }
        else {
            unhandled(message);
        }
```

After

```
class Worker extends BaseActor implements Messages {

   public void handleResponse(SomeMessage message) {
      doStuff();
   }

   public void handleRequest(SomeOtherMessage message) {
      doSomeOtherStuff();
   }

   public void process(YetAnotherMessage message) {
      doEvenMoreStuff();
   }
}
```

omega point.

Conclusions

- Contracts to explicitly define behavior works pretty well
- Can be a useful tool in your toolbox
- Scala can give similar support via traits & match

Testing Akka Code

Interact by sending messages

BDD Scenarios

Integration tests

omega point.

Scenarios

Case A1: Sort any sequence of numbers in increasing order

Given a sequence of numbers in decreasing order

When applying the sort algorithm

Then the resulting sequence of numbers is in increasing order

The Test

```
@Test
public void caseA1() {
    given(sequence(9,8,7,6,5,4,3,2,1,0));

    whenSorting();

    thenResultIs(sequence(0,1,2,3,4,5,6,7,8,9));
}
```

omega point.

Key Take-Aways

- It's possible to use Akka in legacy code
- Akka is Scala & Java compliant
- Akka is a toolkit with a lot of goodies
- Stop writing legacy code

Thank you

@DanielDeogun @DanielSawano

omega point.