Akka & Scala: distribution with no pain

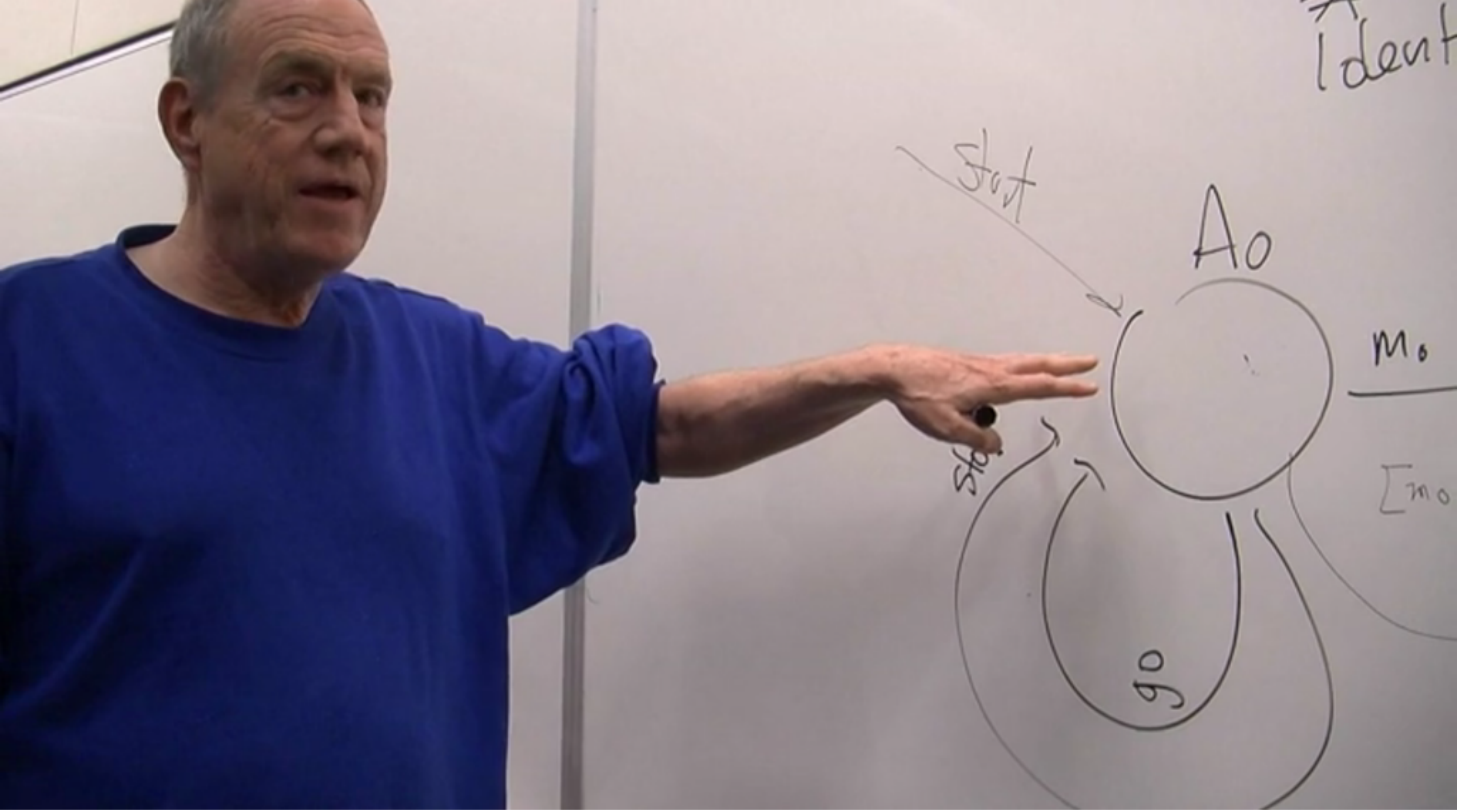
June 24th, 2015

Alexandre Masselot
Scala Romandie Meetup
http://alexandre.masselot@blogspot.ch
http://ch.linkedin.com/in/alexmass

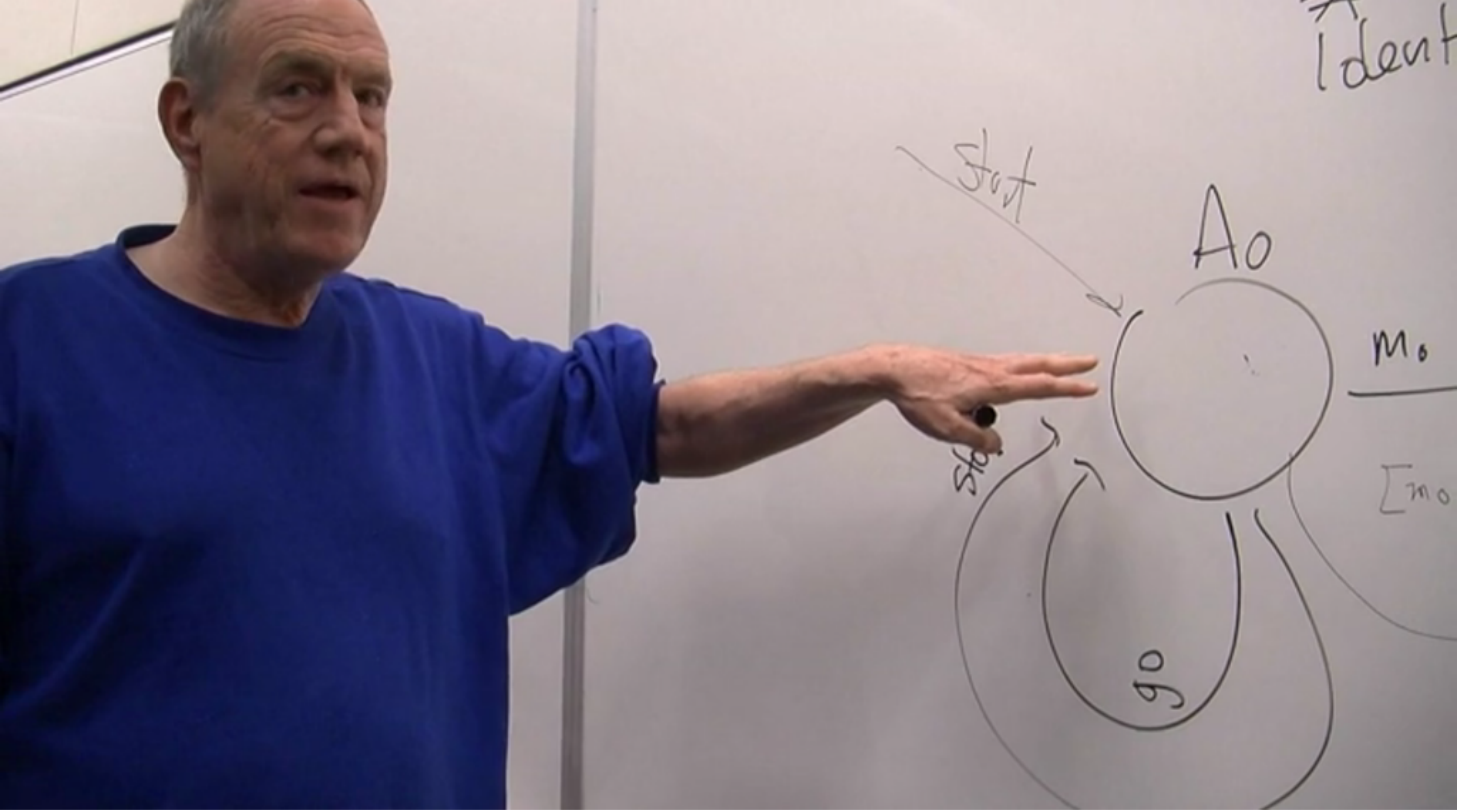
Scala & Akka: the perfect match

Scala & Akka:
beauty & efficiency

What is an Actor?



Carl Hewitt, Peter Bishop *et* Richard Steiger
"A Universal Modular Actor Formalism for Artificial Intelligence"
IJCAI 1973



Carl Hewitt, Peter Bishop *et* Richard Steiger
"A Universal Modular Actor Formalism for Artificial Intelligence"
IJCAI **1973**

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

An actor embodies:

- Processing
- Storage
- Communications

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

When it receives a message, it can:

- Create new actors
- Send messages to actors it knows
- Designate how it show handle
  the next message it will receive

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

The message mailbox

“One actor is no actor.
They come in systems”

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

An actor system
carries indeterminism

Akka

Created by Jonas Bonér

“Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.”

http://akka.io

```
#build.sbt


libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.3.11",
  "com.typesafe.akka" %% "akka-slf4j" % “2.3.11"
)
```

~2.5 million actors per GB of heap

50 million msg/sec
on a single machine

Load balancing, resiliency local & remote

And it’s cool to use

Akka actors in action

https://github.com/alexmasselot/scala-romandie-akka-primes

#1 - just print it

```
class PingActor extends Actor with ActorLogging {
  def receive = {
    case name:String =>
      log.info(s"hello, my name is $name")
 }
}
```

```
class PingActor extends Actor with ActorLogging {
  def receive = {
    case name:String =>
      log.info(s"hello, my name is $name")
 }
}

object HelloApp extends App{
  val system = ActorSystem("MyActorSystem")
  val pingActor =
          system.actorOf(Props[PingActor], "pingActor")




}
```

```
class PingActor extends Actor with ActorLogging {
  def receive = {
    case name:String =>
      log.info(s"hello, my name is $name")
 }
}

object HelloApp extends App{
  val system = ActorSystem("MyActorSystem")
  val pingActor =
          system.actorOf(Props[PingActor], "pingActor")

  pingActor ! "Bond"

}
```

```
[INFO] [23:20:17.881] … hello, my name is Bond
```

#2 - with multiple messages

```scala
case class Name(value:String) extends AnyVal
object Tcho

class PingActor extends Actor with ActorLogging {
  def receive = {
    case Name(name) =>
      log.info(s"hello, my name is $name")
    case Tcho =>
      log.info("shutting down")
      context.system.shutdown()
  }
}
```

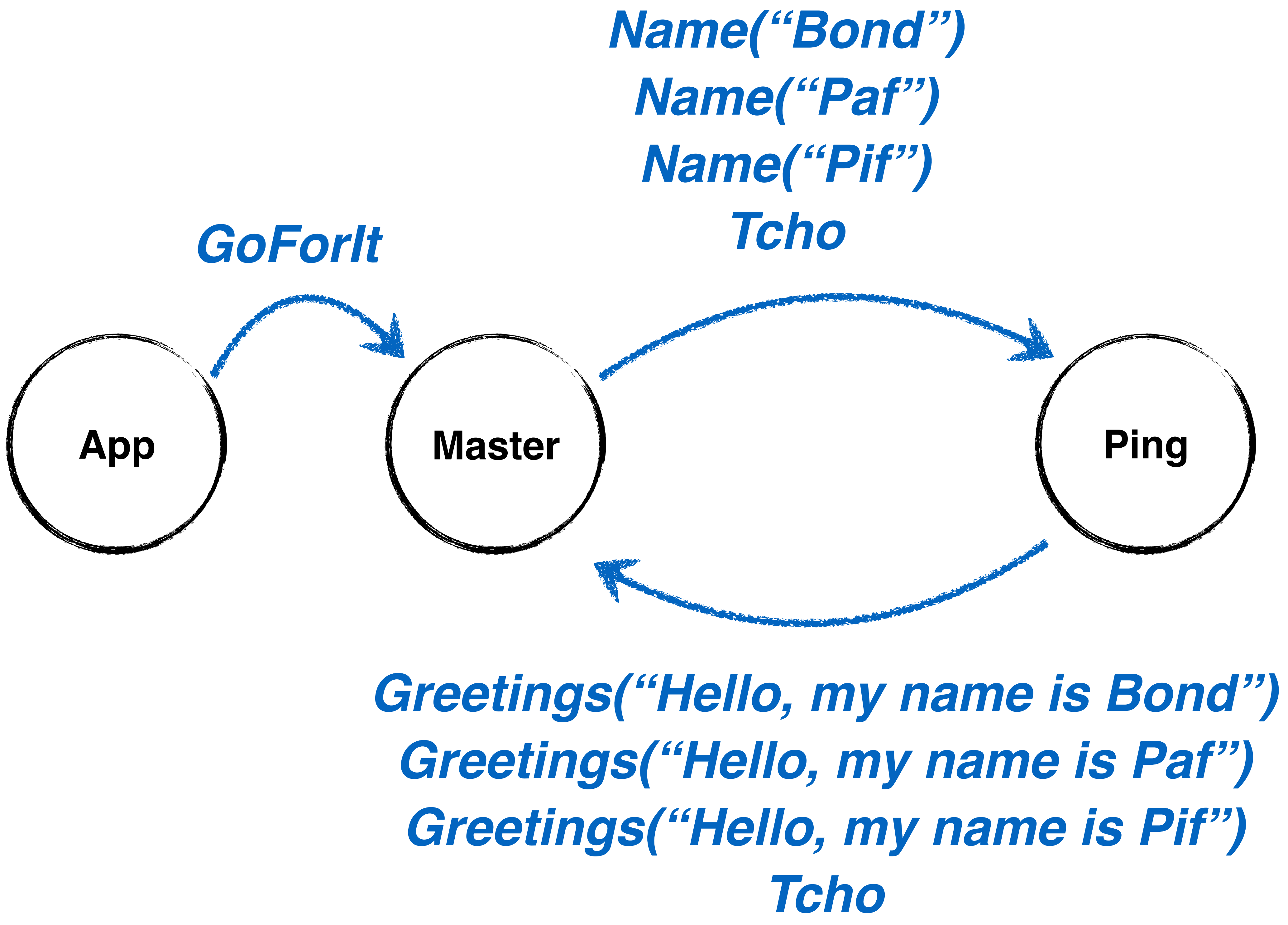
```scala
object HelloMultiMessagesApp extends App{
  val system = ActorSystem("MyActorSystem")
  val pingActor = system.actorOf(Props[PingActor],
                              "pingActor")

  pingActor ! Name("Bond")
  pingActor ! Name("Paf")
  pingActor ! Name("Pif")
  pingActor ! Tcho

  system.awaitTermination()
}
```

```
[23:24:20.171] … hello, my name is Bond
[23:24:20.172] … hello, my name is Paf
[23:24:20.172] … hello, my name is Pif
[23:24:20.172] … shutting down
```

#3 - sending messages back and forth


Name("Bond")
Name("Paf")
Name("Pif")
Tcho
GoForIt
App
Master
Ping
Greetings("Hello, my name is Bond")
Greetings("Hello, my name is Paf")
Greetings("Hello, my name is Pif")
Tcho

```scala
case class Name(value: String)
case class Greetings(value: String)
object Tcho
object GoForIt

class PingActor extends Actor with ActorLogging {
  def receive = {
    case Name(name) =>
      log.info(s"received [$name]")
      sender ! Greetings(s"hello, my name is $name")
    case Tcho => sender ! tcho
  }
}
```

```
class MasterActor extends Actor with ActorLogging {
  val pingActor = context.actorOf(Props[PingActor],
                                   "pingActor")
  override def receive: Receive = {
    case GoForIt =>
      List(Name(“Bond"),
           Name(“Paf"),
           Name(“Pif")).foreach(pingActor !)
      pingActor ! Tcho
    case Greetings(message) =>
      log.info(s"received greeting [$message]")
    case Tcho => context.system.shutdown()
  }
}
```

```scala
object HelloMultiActorsApp extends App {
  val system = ActorSystem("MyActorSystem")

  val masterActor = system.actorOf(Props[MasterActor],
                                   "master")

  masterActor ! GoForIt
  system.awaitTermination()
}
```

PSP:
let’s fake some computations

A *PSP* number is prime
and the sum of its digit is prime

```scala
def isPrime(i: Int): Boolean = {
  (2 to Math.sqrt(i).toInt).forall(i % _ != 0)
}
```

```
def isPrime(i: Int): Boolean = {
  (2 to Math.sqrt(i).toInt).forall(i % _ != 0)
}

def sumDigit(i: Int): Int = {
  @tailrec
  def sumDigiHandler (acc:Int, i:Int):Int = i match{
    case x if x < 10 => acc+x
    case x => sumDigiHandler(acc+ (x % 10), x /10)
  }
  sumDigiHandler(0, i)
}
```

```
def isPrime(i: Int): Boolean = {
  (2 to Math.sqrt(i).toInt).forall(i % _ != 0)
}

def sumDigit(i: Int): Int = {
  @tailrec
  def sumDigiHandler (acc:Int, i:Int):Int = i match{
    case x if x < 10 => acc+x
    case x => sumDigiHandler(acc+ (x % 10), x /10)
  }
  sumDigiHandler(0, i)
}

def isPSP(i:Int) = isPrime(i) && isPrime(sumDigit(i))
```

```scala
object PrimeSumPrime {
  def nextPSP(i:Int):Int = (i until Int.MaxValue)
    .find(isPSP) match {
      case Some(j)=> j
      case None => throw new OutOfBoundException(i)
    }

  def allPSP(start:Int, end:Int):Seq[Int] =
    (start until end).filter(isPSP)

  def streamPSP(start:Int, end:Int):Stream[Int] =
    (start until end).toStream.filter(isPSP)
}
```

#4 - bean, list, stream & ask

(get the flow back)

```
class PSPActor extends Actor with ActorLogging {
  override def receive: Receive = {
    case FindNextPSP(i) =>
      log.info(s"FindNextPSP($i)")
      sender ! PSPSingle(PrimeSumPrime.nextPSP(i))









  }
}
```

```
class PSPActor extends Actor with ActorLogging {
  override def receive: Receive = {
    case FindNextPSP(i) =>
      log.info(s"FindNextPSP($i)")
      sender ! PSPSingle(PrimeSumPrime.nextPSP(i))
    case FindListPSP(start, end) =>
      log.info(s"FindListPSP($start, $end)")
      sender ! PSPList(PrimeSumPrime.allPSP(start, end))







  }
}
```

```scala
class PSPActor extends Actor with ActorLogging {
  override def receive: Receive = {
    case FindNextPSP(i) =>
      log.info(s"FindNextPSP($i)")
      sender ! PSPSingle(PrimeSumPrime.nextPSP(i))
    case FindListPSP(start, end) =>
      log.info(s"FindListPSP($start, $end)")
      sender ! PSPList(PrimeSumPrime.allPSP(start, end))
    case FindStreamPSP(start, end) =>
      log.info(s"FindStreamPSP($start, $end)")
      PrimeSumPrime.streamPSP(start, end).foreach({
        x => sender ! PSPSingle(x)
      })
  }
}
```

```scala
class MasterActor extends Actor with ActorLogging {
  val pspActor = context.actorOf(Props[PSPActor],
                                 "pspActor")







  override def receive: Receive = {
    case PSPSingle(i) =>
      log.info(s"received PSP [$i]")
    case al: PSPList =>
      log.info(s"received PSP $al")
  }
}
```

```
class MasterActor extends Actor with ActorLogging {
  val pspActor = context.actorOf(Props[PSPActor],
                                 "pspActor")

  pspActor ! FindNextPSP(1000)
```

```
case FindNextPSP(i) =>
  sender ! PSPSingle(PrimeSumPrime.nextPSP(i))
```

```
  override def receive: Receive = {
    case PSPSingle(i) =>
      log.info(s"received PSP [$i]")
    case al: PSPList =>
      log.info(s"received PSP $al")
  }
}
```

```
class MasterActor extends Actor with ActorLogging {
  val pspActor = context.actorOf(Props[PSPActor],
                                  "pspActor")

 pspActor ! FindListPSP(1000000000, 1000010000)


           case FindListPSP(start, end) =>
               sender ! PSPList(PrimeSumPrime.allPSP(start, end))

  override def receive: Receive = {
    case PSPSingle(i) =>
      log.info(s"received PSP [$i]")
    case al: PSPList =>
      log.info(s"received PSP $al")
  }
}
```

```scala
class MasterActor extends Actor with ActorLogging {
  val pspActor = context.actorOf(Props[PSPActor],
                                "pspActor")


 pspActor ! FindStreamPSP(1000000000, 1000000200)
```

```scala
case FindStreamPSP(start, end) =>
    PrimeSumPrime.streamPSP(start, end).foreach({
      x => sender ! PSPSingle(x)
     })
```

```scala
  override def receive: Receive = {
    case PSPSingle(i) =>
      log.info(s"received PSP [$i]")
    case al: PSPList =>
      log.info(s"received PSP $al")
  }
}
```

```
class MasterActor extends Actor with ActorLogging {
  val pspActor = context.actorOf(Props[PSPActor],
                                  "pspActor")
  implicit val timeout = Timeout(100.days)

  (pspActor ? FindListPSP(2000000000, 2000100000))
     .mapTo[PSPList]
     .map(a => log.info(s"received from ask $a"))

  override def receive: Receive = {
    case PSPSingle(i) =>
      log.info(s"received PSP [$i]")
    case al: PSPList =>
      log.info(s"received PSP $al")
  }
}
```

#5 - scale with routers

```
class MasterActor extends Actor with ActorLogging {

  val pspActor = context.actorOf(Props[PSPActor],
                                 "pspActor")



  pspActor ! FindListPSP(2000000000, 2000100000)
  pspActor ! FindListPSP(2000100000, 2000200000)
  pspActor ! FindListPSP(2000200000, 2000300000)
  pspActor ! FindListPSP(2000300000, 2000400000)

  override def receive: Receive = {
    case al: PSPList =>
      log.info(s"received PSP $al")
  }
}
```

```
[00:25:41.659] [akka://MyActorSystem/user/master/pspActor]
           FindListPSP(2000000000, 2000100000)
[00:25:44.672] [akka://MyActorSystem/user/master/pspActor]
           FindListPSP(2000100000, 2000200000)
[00:25:44.672] [akka://MyActorSystem/user/master]
           received PSP len=1817 (2000000063..2000099957)
[00:25:47.491] [akka://MyActorSystem/user/master/pspActor]
           FindListPSP(2000200000, 2000300000)
[00:25:47.491] [akka://MyActorSystem/user/master]
           received PSP len=1848 (2000100097..2000199967)
[00:25:50.262] [akka://MyActorSystem/user/master]
           received PSP len=1761 (2000200003..2000299997)
[00:25:50.262] [akka://MyActorSystem/user/master/pspActor]
           FindListPSP(2000300000, 2000400000)
[00:25:53.063][akka://MyActorSystem/user/master]
           received PSP len=1809 (200300033..200399983)
```

```
class MasterActor extends Actor with ActorLogging {

  val pspActor = context.actorOf(Props[PSPActor],
                                 "pspActor")



  pspActor ! FindListPSP(2000000000, 2000100000)
  pspActor ! FindListPSP(2000100000, 2000200000)
  pspActor ! FindListPSP(2000200000, 2000300000)
  pspActor ! FindListPSP(2000300000, 2000400000)

  override def receive: Receive = {
    case al: PSPList =>
      log.info(s"received PSP $al")
  }
}
```

```
class MasterActor extends Actor with ActorLogging {

  val pspActor = context.actorOf(
           RoundRobinPool(5).props(Props[PSPActor]),
           "pspactor-router")

  pspActor ! FindListPSP(2000000000, 2000100000)
  pspActor ! FindListPSP(2000100000, 2000200000)
  pspActor ! FindListPSP(2000200000, 2000300000)
  pspActor ! FindListPSP(2000300000, 2000400000)

  override def receive: Receive = {
    case al: PSPList =>
      log.info(s"received PSP $al")
  }
}
```

```
[00:32:19.682] [akka://MyActorSystem/user/master/
pspactor-router/$a] FindListPSP(2000000000, 2000100000)
[00:32:19.682] [akka://MyActorSystem/user/master/
pspactor-router/$b] FindListPSP(2000100000, 2000200000)
[00:32:19.682] [akka://MyActorSystem/user/master/
pspactor-router/$c] FindListPSP(2000200000, 2000300000)
[00:32:19.682] [akka://MyActorSystem/user/master/]
FindListPSP(2000300000, 2000400000)
[00:32:22.957] [akka://MyActorSystem/user/master]
received PSP len=1761 (2000200003..2000299997)
[00:32:22.991] [akka://MyActorSystem/user/master]
received PSP len=1848 (2000100097..2000199967)
[00:32:23.028] [akka://MyActorSystem/user/master]
received PSP len=1817 (2000000063..2000099957)
[00:32:23.031] [akka://MyActorSystem/user/master]
received PSP len=1809 (2000300033..2000399983)
```

#6 - reference actor by name

```
class PSPActorLogger extends Actor with ActorLogging {
  val writer = new FileWriter("/tmp/psp.log")
  override def receive: Receive = {
    case PSPSingle(i) =>
      writer.write(s"$i\n")
  }
}
```

```scala
class PSPActorLogger extends Actor with ActorLogging {
  val writer = new FileWriter("/tmp/psp.log")
  override def receive: Receive = {
    case PSPSingle(i) =>
      writer.write(s"$i\n")
  }
}
object PSProuterWithLoggerApp extends App {
  val system = ActorSystem("MyActorSystem")
  val loggerActor  =
        system.actorOf(Props[PSPActorLogger],
                       "psp-logger")
  val masterActor =
        system.actorOf(Props[MasterActorWithLogger],
                       "master")
}
```

```
class PSPActorWithLogger extends Actor with
ActorLogging {
  var actorLogger =
         context.actorSelection("/user/psp-logger")

  override def receive: Receive = {
    case FindListPSP(start, end) =>
      log.info(s"FindListPSP($start, $end)")
      val l = PrimeSumPrime.allPSP(start, end)
      l.foreach(x => actorLogger ! PSPSingle(x))
      sender ! PSPList(l)
  }
}
```

#7 - using configuration

(that’s the last one)

```
#conf/akka_01.conf
actor {
  deployment {
    /master/router1 {
      router = round-robin-pool
      nr-of-instances = 5
    }
  }
}
```

```
#conf/akka_01.conf
actor {
  deployment {
    /master/router1 {
      router = round-robin-pool
      nr-of-instances = 5
    }
  }
}

val config = ConfigFactory
            .parseFile(new File("conf/akka_01.conf"))
val system = ActorSystem("MyActorSystem", config)
val masterActor = system.actorOf(Props[MasterActor],
                                  "master")
```

```
#conf/akka_01.conf
 actor {
   deployment {
     /master/router1 {
       router = round-robin-pool
       nr-of-instances = 5
     }
   }
 }

 val config = ConfigFactory
              .parseFile(new File("conf/akka_01.conf"))
 val system = ActorSystem("MyActorSystem", config)
 val masterActor = system.actorOf(Props[MasterActor],
                                   "master")

 //within MasterActor
 val pspActor = context.actorOf(
          FromConfig.props(Props[PSPActor]),
          "router1")
```

And there is way more

Way more routers

- RoundRobinRoutingLogic
- RandomRoutingLogic
- SmallestMailboxRoutingLogic
- BroadcastRoutingLogic
- ScatterGatherFirstCompletedRoutingLogic
- TailChoppingRoutingLogic
- ConsistentHashingRoutingLogic

Way more load balancing

```
akka.actor.deployment {
  /parent/router29 {
    router = round-robin-pool
    resizer {
      lower-bound = 2
      upper-bound = 15
      messages-per-resize = 100
    }
  }
}
```

Way more special messages: `PoisonPill`, `Broadcast`…

Way more resiliency with supervising strategies

Way more scale out with remote actors & akka-cluster

And way more fun!