

Updated: 2021-04-01

# 5.1

## Actors and Akka

© 2015 Robin Hillyard



Northeastern  
University

# What is an Actor?

*All the world's a stage,  
And all the men and women merely players;  
They have their exits and their entrances,  
And one man in his time plays many parts,  
His acts being seven ages.*

Shakespeare, *As You Like It*.

- *Actor* is a mechanism that enables the separation of state from other state--and from a main (stateless) program--in both space and time;
- The concept was first introduced in Erlang;
- Actors form the foundation of the entire Akka system.

# Why Actors?

- We know that side-effects\* (including I/O) and mutable state don't support referential transparency, i.e. we cannot substitute expressions for identifiers when side-effects/mutable state are involved. We cannot provably compose functions (e.g. *for-comprehensions*) when the functions are not R.T.
  - But just about all complex systems involve some side-effects and mutable state. So, how can we make these "rogue" components safe for Functional Programming?
  - We can create concepts such as I/O Monads and state-preserving concepts such as our RNG class. By doing this, we can isolate all the non-pure code into very clear chunks, leaving all the rest of the logic to be *composable* in our FP way. BTW, I/O Monads are not easy and you can't find them in the standard Scala library.
  - What about *mutable* state? Another technique is to isolate mutable state in a way that other parts of the system can never observe a mutation. For instance, we want to implement *quicksort* by creating an *Array* of the elements and mutating their positions. Then we copy the array back to the original form. As long as the rest of the system can never substitute that array, we should be safe.

\* **side-effect** is basically any interaction with the world outside our program and, in particular, the thread it inhabits.

# Why Actors? (2)

- And, there is one other practical consideration. When we are dealing with side-effects, those actions typically take enormous amounts of time compared with “normal” programming. We typically want to interact with the world outside asynchronously, i.e. using a *Future*.
- There is one further golden rule of functional programming: let each method (or function) do one thing and do it well. [Actually, this should be the rule for all programming, period.] Remember our mantra:

simple, obvious, elegant.

- *Actors* embody all the properties so far mentioned (although they are admittedly not the only way to do so). But actors can do a lot more...
- Actors were introduced to the computer programming world by *Erlang* (“Erlang concurrency model”).
- Scala has its own actors but they are now deprecated in favor of...

# Akka: "Classic" actors

- *Akka* actors are:
  - **Message receivers and senders:**
    - That is, they can communicate with the rest of the system only via messages—this helps reduce *coupling*;
    - Each actor has its own “mailbox” into which messages are deposited by the actors system.
  - **Encapsulating:**
    - In the sense we talked about—the only handle that we, as programmers, have to an actor (except when testing) is an *ActorRef* which does not give access to any actor internals such as mutable state. Therefore the rest of the system cannot observe any side-effects or mutability directly and can therefore follow normal FP composition rules and patterns.
  - **Thread-safe:**
    - In the sense that the entire processing of any one message is completed before any other message from the mailbox can be read or processed.
  - **Untyped\*:**
    - That’s to say an actor has no intrinsic knowledge about the types of its properties—all actors are essentially the same in this regard. Information about typed objects is confined to the messages sent and received.
  - **Replicable:**
    - That’s to say that the actor system can make copies of an actor to improve performance. These copies can be on remote systems of course.
  - **Resilient:**
    - That’s to say that the actor system monitors the health of actors and will restart an actor if required, potentially bubbling up to the surface any exception.
  - **Lightweight:**
    - The boiler-plate overhead for an actor is only about 1k bytes so it’s practical to have millions of them in an application.

\* typed actors are here!.

# Akka Classic (2)

- Overall, these properties make *Akka* the perfect system for *reactive programming*!
- So, how do we get started? First go to <http://akka.io> for documentation, tutorials, patterns, etc.
- Then add the following to your *build.sbt*

```
val akkaGroup = "com.typesafe.akka"
val akkaVersion = "2.6.5"
libraryDependencies += Seq(
  akkaGroup %% "akka-actor" % akkaVersion,
  akkaGroup %% "akka-testkit" % akkaVersion % "test",
  akkaGroup %% "akka-slf4j" % akkaVersion,
  "com.typesafe" % "config" % "1.4.0",
  "ch.qos.logback" % "logback-classic" % "1.2.3" % "runtime"
)
```
- Let's take a look at an existing system: using Akka to solve Map-Reduce problems ([Majabigwaduce](#))

# Akka Classic (3)

- An actor receives *typed* messages via its *receive* method\*:

```
/**
 * The purpose of this mapper is to convert a sequence of objects into several sequences, each of
which is
 * associated with a key...
 * ...
 */
class Mapper[K1,V1,K2,W](f: (K1,V1)=>(K2,W)) extends MapReduceActor {
  override def receive = {
    case i: Incoming[K1,V1] =>
      log.info(s"received $i")
      val wk2ts = for ((k1,v1) <- i.m) yield Try(f(k1,v1))
      sender ! prepareReply(wk2ts)
    case q =>
      super.receive(q)
  }
}
case class Incoming[K, V](m: Seq[(K,V)]) {
  override def toString = s"Incoming: with ${m.size} elements"
}
```

- Typically, the actor will prepare a response and send it either to the sender or another actor. A logging actor is able to log the messages as they arrive.

\* This is taken from [Majabigwaduce](#)

# Akka Classic (4)

```
import akka.actor.{ Actor, ActorLogging, ActorSystem, Props }
import scala.concurrent.duration._
import scala.concurrent._
import akka.util.Timeout
import akka.pattern.ask
import scala.util._

import ExecutionContext.Implicits.global
case class QuickSort() extends Actor with ActorLogging {
  override def receive = {
    case input: Seq[Int] =>
      log.info(s"received $input")
      val array = input.toArray
      Sorting.quickSort(array)
      val response = array.toSeq
      sender ! response
    case _ => println("unknown message")
  }
}

object QuickSort extends App {
  implicit val timeout: Timeout = 10.seconds
  implicit val system = ActorSystem("QuickSort")
  val quickSort = system.actorOf(Props.create(classOf[QuickSort]), "sorter")
  val ints = args map {_.toInt}
  val f = quickSort ? ints.toSeq
  f.onComplete { x => x match {
    case Success(s) => println(s"received response of sorted sequence: $s")
    case Failure(e) => System.err.println(s"exception: $e")
  }
}
Await.ready(f, 10.second)
system.stop(quickSort)
system.shutdown
}
```

Quicksort is most efficient for large  $N$  when it is operating on a (mutable) array. Encapsulating array inside an *Actor* is *Referentially Transparent*.

Quicksort is one of the best-known and most efficient sorting methods: it is generally  $O(N \log N)$  but in worst case is  $O(N^2)$ .



# Akka Classic (5)

- Our repository has several places that use actors.
- First, let's take a look at lab-actors:
  - lab-actors is based on a previous project called HedgeFund, which keeps track of stock and option prices and makes recommendations for puts and calls (“options” or “derivatives”).
  - lab-actors uses Scala 2.11.9 because the HTTP client it uses (“spray-can”) has been deprecated and replaced by AkkaHTTP.

# Akka Classic: lab-actors uses “blackboard” pattern

```
package edu.neu.coe.csye7200.actors

import akka.actor.{Actor, ActorLogging, ActorRef, Props}

/**
 * @author robinhillyard
 */
class Blackboard(forwardMap: Map[Class[_ <: Any], String], actors: Map[String, Class[_ <: BlackboardActor]]) extends Actor with ActorLogging {

  val actorMap: Map[String, ActorRef] = actors map {
    case (k, v) => k -> context.actorOf(Props.create(v, self), k)
  }

  // To encode specific, non-forwarding behavior, override this method
  override def receive: PartialFunction[Any, Unit] = {
    case message =>
      forwardMap.get(message.getClass) match {
        case Some(s) => actorMap.get(s) match {
          case Some(k) => k forward message
          case _ => log.warning(s"no actor established for key $s")
        }
        case _ => log.warning(s"no forward mapping established for message class ${message.getClass}")
      }
  }
}
```

# Blackboard (classic) actor example

```
package edu.neu.coe.csye7200.actors
import akka.actor.{ActorRef, Props}
import spray.http._

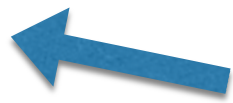
class HttpReader(blackboard: ActorRef) extends BlackboardActor(blackboard) {
  val entityParser: ActorRef = context.actorOf(Props.create(classOf[EntityParser],
    blackboard), "EntityParser")

  override def receive: PartialFunction[Any, Unit] = {
    case HttpResult(queryProtocol, request, HttpResponse(status, entity, headers, protocol))
=>
      log.info("request sent: {}; protocol: {}; response status: {}", request, protocol,
status)
      if (status.isSuccess)
        processResponse(entity, headers, queryProtocol)
      else
        log.error("HTTP transaction error: {}", status.reason)
    case m => super.receive(m)
  }
  def processResponse(entity: HttpEntity, headers: List[HttpHeader], protocol: String): Unit
= {
    log.debug("response headers: {}; entity: {}", headers, entity)
    entityParser ! EntityMessage(protocol, entity)
  }
}
case class EntityMessage(protocol: String, entity: HttpEntity)
```

# Map-Reduce

- Map-reduce is a pattern for parallel processing:
  - it is based on the notion that you can divide a problem into  $N$  tasks:
    - provided that each of the tasks is truly independent;
    - where you have a “map” function which yields a “key” for each element (sub-division) of the problem;
    - and where you have a “reduce(k)” function which evaluates all elements with key “k” in parallel with (and independent of) all other elements;
    - once all of the  $N$  values are available, they are collected together and combined for the final report/result;
  - it is the pattern on which *Hadoop* and *Spark* are based;
  - it is very amenable to functional programming.

# Map-Reduce (2)

- The  $n^{\text{th}}$  stage of the process looks like this:
  - $Map[K_{n-1}, V_{n-1}] \xrightarrow{\text{"mapper/groupBy"}} Map[K_n, Seq[W_n]] \xrightarrow{\text{"reducer(s)"}} Map[K_n, V_n]$
  - But, typically, the input data to the first stage has no natural key so we use the fact that  $Map[K_0, V_0]$  can be transformed directly to  $Seq[(K_0, V_0)]$  where  $Seq[(\emptyset, V_0)]$  in turn is the equivalent of  $Seq[V_0]$ .  But we can't *generally* go in the reverse direction. Why not?
- Assuming, then, that each ( $n^{\text{th}}$ ) stage of the pipeline works as expected, then overall, we can transform:
  - $Seq[V_0] \rightarrow Map[K_n, V_n]$

# Map-Reduce (3)

- For example: WebCrawler application...
  - $Map[\emptyset, String] \rightarrow Map[URI, Seq[URI]] \rightarrow Map[URI, Seq[String]]$   
 $\rightarrow Map[URI, Seq[String]] \rightarrow Map[Seq[String], Seq[String]]$   
 $\rightarrow Seq[String]$

# CountWords app

```
/**
 * CountWords: an example application of the MapReduce framework.
 * This application is a three-stage map-reduce process (the final stage is a pure reduce process).
 * Stage 1 takes a list of Strings representing URIs, converts to URIs, opens each as a stream, reading the contents
 * and finally returns a map of URI->Seq[String]
 * where the key is the URI of a server, and the Strings are the contents of each of the documents retrieved from that
 * server.
 * Stage 2 takes the map of URI->Seq[String] resulting from stage 1 and adds the lengths of the documents (in words) to
 * each other. The final result is a map of
 * URI->Int where the value is the total number of words read from the server represented by the key.
 * Stage 3 then sums these values together to yield a grand total.
 */
case class CountWords(resourceFunc: String => Resource)(implicit system: ActorSystem, config: Config, timeout: Timeout,
ec: ExecutionContext) extends (Seq[String] => Future[Int]) {
  override def apply(v1: Seq[String]): Future[Int] = {
    def init = Seq[String]()

    val stage1: MapReduce[String, URI, Seq[String]] = MapReduceFirstFold(
      { w: String => val u = resourceFunc(w); system.log.debug(s"stage1 map: $w"); (u.getServer, u.getContent) }, { (a:
Seq[String], v: String) => a :+ v },
      init _
    )
    val stage2: MapReduce[(URI, Seq[String]), URI, Int] = MapReducePipe(
      { (w: URI, gs: Seq[String]) => (w, (for (g <- gs) yield g.split("""\s+""").length) reduce (_ + _)) }, { (x: Int,
y: Int) => x + y },
      1
    )
    val stage3 = Reduce[Int, Int](_ + _)
    val countWords = stage1 | stage2 | stage3
    countWords.apply(v1)
  }
}
```

# CountWords app

```
object CountWords {
  def apply(hc: HttpClient, args: Array[String]): Future[Int] = {
    val configRoot = ConfigFactory.load
    implicit val config: Config = configRoot.getConfig("CountWords")
    implicit val system: ActorSystem = ActorSystem(config.getString("name"))
    implicit val timeout: Timeout = getTimeout(config.getString("timeout"))
    import ExecutionContext.Implicits.global

    val ws = if (args.length > 0) args.toSeq else Seq("http://www.bbc.com/doc1",
"http://www.cnn.com/doc2", "http://default/doc3", "http://www.bbc.com/doc2", "http://www.bbc.com/doc3")
    CountWords(hc.getResource).apply(ws)
  }

  // TODO try to combine this with the same method in MapReduceActor
  def getTimeout(t: String): Timeout = {
    val durationR = """"(\d+)\s*(\w+)""".r
    t match {
      case durationR(n, s) => new Timeout(FiniteDuration(n.toLong, s))
      case _ => Timeout(10 seconds)
    }
  }
}
```



# CountWords unit test

```
class CountWordsSpec extends FlatSpec with Matchers with Futures with ScalaFutures with Inside with
MockFactory {
  "CountWords" should "work for http://www.bbc.com/ http://www.cnn.com/ http://default/" in {
    val wBBC = "http://www.bbc.com/"
    val wCNN = "http://www.cnn.com/"
    val wDef = "http://default/"
    val uBBC = new URI(wBBC)
    val uCNN = new URI(wCNN)
    val uDef = new URI(wDef)
    val hc = mock[HttpClient]
    val rBBC = mock[Resource]
    (rBBC.getServer _).expects().returning(uBBC)
    rBBC.getContent _ expects() returning CountWordsSpec.bbcText
    val rCNN = mock[Resource]
    rCNN.getServer _ expects() returning uCNN
    rCNN.getContent _ expects() returning CountWordsSpec.cnnText
    val rDef = mock[Resource]
    rDef.getServer _ expects() returning uDef
    rDef.getContent _ expects() returning CountWordsSpec.defaultText
    hc.getResource _ expects wBBC returning rBBC
    hc.getResource _ expects wCNN returning rCNN
    hc.getResource _ expects wDef returning rDef
    val nf = CountWords(hc, Array(wBBC, wCNN, wDef))
    whenReady(nf, timeout(Span(6, Seconds))) {
      case i => assert(i == 556)
    }
  }
}
```

# Akka: typed actors

- *Akka* typed actors are:
  - **Message receivers and senders:**
    - As before.
  - **Encapsulating:**
    - As before.
  - **Thread-safe:**
    - As before.
  - **Typed:**
    - Typed actors have a very different API. See next slide.
  - **Replicable:**
    - As before.
  - **Resilient:**
    - As before.
  - **Lightweight:**
    - As before.

# Akka: Typed Actors (2)

- Essentials of the API:
  - Actors are created through factory methods (unlike untyped actors which were constructed);
  - An actor has (typed) *behavior*;
  - The actor's receive method returns a (potentially different) behavior after doing any of the following:
    - creating and messaging other actors;
    - changing internal state

# Exercise:

- Go to <https://developer.lightbend.com/start/?group=akka&project=akka-quickstart-scala>
  - Click CREATE A PROJECT FOR ME
- Make a New project in your IDE using the existing source directory that you just downloaded;
  - When offered a choice of project type, use SBT (or BSP);
  - You will probably have to specify a JDK (use whatever you normally use);
  - Run *AkkaQuickStart* (e.g. right-click on the source file in the Project window and select Run).
- You've just created and used actors!

```

package com.example
import akka.actor.typed._
import akka.actor.typed.scaladsl.Behaviors
import com.example.GreeterMain.SayHello
object Greeter {
  final case class Greet(whom: String, replyTo: ActorRef[Greeted])
  final case class Greeted(whom: String, from: ActorRef[Greet])
  def apply(): Behavior[Greet] = Behaviors.receive { (context, message) =>
    context.log.info("Hello {}!", message.whom)
    message.replyTo ! Greeted(message.whom, context.self)
    Behaviors.same
  }
}
object GreeterBot {
  def apply(max: Int): Behavior[Greeter.Greeted] = { bot(0, max) }
  private def bot(greetingCounter: Int, max: Int): Behavior[Greeter.Greeted] = Behaviors.receive {
(context, message) =>
    val n = greetingCounter + 1
    context.log.info("Greeting {} for {}", n, message.whom)
    if (n == max) Behaviors.stopped
    else { message.from ! Greeter.Greet(message.whom, context.self); bot(n, max) }
  }
}
object GreeterMain {
  final case class SayHello(name: String)
  def apply(): Behavior[SayHello] = Behaviors.setup { context =>
    val greeter = context.spawn(Greeter(), "greeter")
    Behaviors.receiveMessage { message =>
      val replyTo = context.spawn(GreeterBot(max = 3), message.name)
      greeter ! Greeter.Greet(message.name, replyTo)
      Behaviors.same
    }
  }
}
object AkkaQuickstart extends App {
  val greeterMain: ActorSystem[GreeterMain.SayHello] = ActorSystem(GreeterMain(), "AkkaQuickStart")
  greeterMain ! SayHello("Charles")
}

```

# Akka

The industry's leading cloud-native frameworks and runtimes.



Full suite of reactive microservices frameworks and runtimes  
for building cloud-native applications



The Power of Akka optimized for the cloud



Akka Streams, Spark, Flink and everything you need to rapidly  
build and operate streaming data applications on Kubernetes



Next generation, stateful serverless computing,  
powered by Akka



<https://www.lightbend.com>