



Lifecycle Monitoring and The Error Kernel

Principles of Functional Programming

Roland Kuhn

Lifecycle Monitoring

The only observable transition occurs when stopping an actor:

- ▶ having an ActorRef implies liveness (at some earlier point)
- ▶ restarts are not externally visible
- ▶ after stop there will be no more responses

Lifecycle Monitoring

The only observable transition occurs when stopping an actor:

- ▶ having an ActorRef implies liveness (at some earlier point)
- ▶ restarts are not externally visible
- ▶ after stop there will be no more responses

No replies could also be due to communication failure, therefore Akka supports Lifecycle Monitoring a.k.a. DeathWatch.

- ▶ an Actor registers its interest using `context.watch(target)`
- ▶ it will receive a `Terminated(target)` message when target stops
- ▶ it will not receive any direct messages from target thereafter

The DeathWatch API

```
trait ActorContext {  
  def watch(target: ActorRef): ActorRef  
  def unwatch(target: ActorRef): ActorRef  
  ...  
}
```

```
case class Terminated private[akka] (actor: ActorRef)  
  (val existenceConfirmed: Boolean, val addressTerminated: Boolean)  
  extends AutoReceiveMessage with PossiblyHarmful
```

Applying DeathWatch to Controller & Getter (1)

```
class Getter(url: String, depth: Int) extends Actor {  
  ...  
  def receive = {  
    case body: String =>  
      for (link <- findLinks(body))  
        context.parent ! Controller.Check(link, depth)  
      context.stop(self)  
    case _: Status.Failure => context.stop(self)  
  }  
}
```

Simply terminating when done uses DeathWatch as End-Of-Conversation.

The Children List

Each actor maintains a list of the actors it created:

- ▶ the child has been entered when `context.actorOf` returns
- ▶ the child has been removed when `Terminated` is received
- ▶ an actor name is available IFF there is no such child

```
trait ActorContext {  
  def children: Iterable[ActorRef]  
  def child(name: String): Option[ActorRef]  
  ...  
}
```

Applying DeathWatch to Controller & Getter (2)

```
class Controller extends Actor with ActorLogging {
  override val supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 5) {
    case _: Exception => SupervisorStrategy.Restart
  }
  def receive = {
    case Check(url, depth) =>
      if (!cache(url) && depth > 0)
        context.watch(context.actorOf(getterProps(url, depth - 1)))
      cache += url
    case Terminated(_) =>
      if (context.children.isEmpty) context.parent ! Result(cache)
    case ReceiveTimeout => context.children foreach context.stop
  }
  ...
}
```

Lifecycle Monitoring for Fail-Over

```
class Manager extends Actor {  
  def prime(): Receive = {  
    val db = context.actorOf(Props[DBActor], "db")  
    context.watch(db)  
  
    {  
      case Terminated('db') => context.become(backup())  
    }  
  }  
  def backup(): Receive = { ... }  
  def receive = prime()  
}
```

The Error Kernel

Keep important data near the root, delegate risk to the leaves.

- ▶ restarts are recursive (supervised actors are part of the state)
- ▶ restarts are more frequent near the leaves
- ▶ avoid restarting Actors with important state

Application to Receptionist (1)

- ▶ Always stop Controller if it has a problem.
- ▶ React to Terminated to catch cases where no Result was sent.
- ▶ Discard Terminated after Result was sent.

Application to Receptionist (1)

- ▶ Always stop Controller if it has a problem.
- ▶ React to Terminated to catch cases where no Result was sent.
- ▶ Discard Terminated after Result was sent.

```
class Receptionist extends Actor {  
  
  override def supervisorStrategy = SupervisorStrategy.stoppingStrategy  
  
  ...  
}
```

Application to Receptionist (2)

```
class Receptionist extends Actor {  
  ...  
  def runNext(queue: Vector[Job]): Receive = {  
    reqNo += 1  
    if (queue.isEmpty) waiting  
    else {  
      val controller = context.actorOf(controllerProps, s"c$reqNo")  
      context.watch(controller)  
      controller ! Controller.Check(queue.head.url, 2)  
      running(queue)  
    }  
  }  
}
```

Application to Receptionist (3)

```
def running(queue: Vector[Job]): Receive = {  
  case Controller.Result(links) =>  
    val job = queue.head  
    job.client ! Result(job.url, links)  
    context.stop(context.unwatch(sender()))  
    context.become(runNext(queue.tail))  
  case Terminated(_) =>  
    val job = queue.head  
    job.client ! Failed(job.url)  
    context.become(runNext(queue.tail))  
  case Get(url) =>  
    context.become(enqueueJob(queue, Job(sender(), url)))  
}
```

Interjection: the EventStream (1)

Actors can direct messages only at known addresses.

The EventStream allows publication of messages to an unknown audience.

Every actor can optionally subscribe to (parts of) the EventStream.

```
trait EventStream {  
  def subscribe(subscriber: ActorRef, topic: Class[_]): Boolean  
  def unsubscribe(subscriber: ActorRef, topic: Class[_]): Boolean  
  def unsubscribe(subscriber: ActorRef): Unit  
  def publish(event: AnyRef): Unit  
}
```

Interjection: the EventStream (2)

```
class Listener extends Actor {  
  context.system.eventStream.subscribe(self, classOf[LogEvent])  
  def receive = {  
    case e: LogEvent => ...  
  }  
  override def postStop(): Unit = {  
    context.system.eventStream.unsubscribe(self)  
  }  
}
```

Where do Unhandled Messages Go?

Actor.Receive is a partial function, the behavior may not apply.

Unhandled messages are passed into the unhandled method:

```
trait Actor {  
  ...  
  def unhandled(message: Any): Unit = message match {  
    case Terminated(target) => throw new DeathPactException(target)  
    case msg =>  
      context.system.eventStream.publish(UnhandledMessage(msg, sender(), self))  
  }  
}
```