Introduction a Akka

Reactive Programming & Acteurs

J.MOLIERE-jerome@javaxpert.com

Zenika

20140925

Ce jeu de slides

A propos de ces slides :

- produits avec LaTeXsous Emacs
- geres en version sous Git
- tout cela sous Debian...
- version de Scala :2.11
- version de JVM utilisee : 1.8.020

Acteurs

Les acteurs sont un des avatars du **Reactive Programming**. Un ensemble de **patterns** visant a construire des applications :

- tolerantes aux pannes,
- pouvant monter en charge,
- legeres en terme de ressources,
- multi threads sans gestion explicite de la concurrence.

Les origines

Initialement integre dans le langage, un module de gestion des acteurs.Ce module a grossi , deborde le stade du simple module pour devenir un produit a part entiere :

- support officiel par TypeSafe,
- outillage d'administration/supervision,
- nombreux algorithmes de **load balancing**,
- l'ancien framework integre au langage est dorenavant deprecie depuis la version 2.10.

But du framework Akka

Le framework Akka propose une solution couvrant l'ensemble des besoins pour une application critique :

1. distribution sur plusieurs machines
2. tolerance aux pannes
3. supervision
4. repartition de la charge

Acteurs : definition

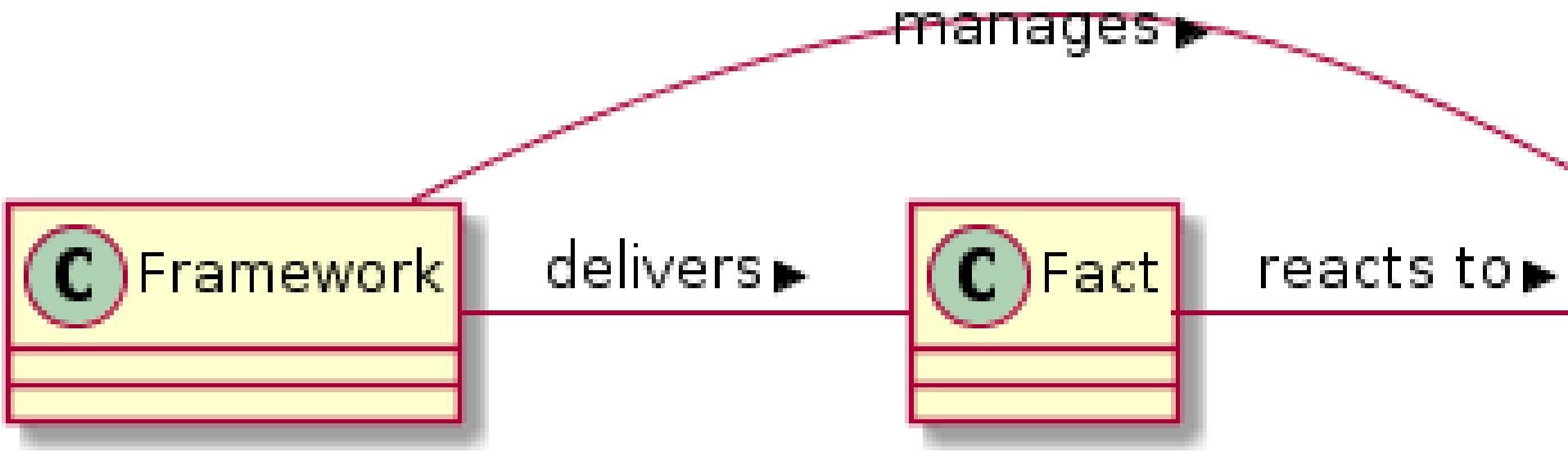
Definition

Un acteur est une unite de calcul encapsulant :

- stockage
- traitement
- communication

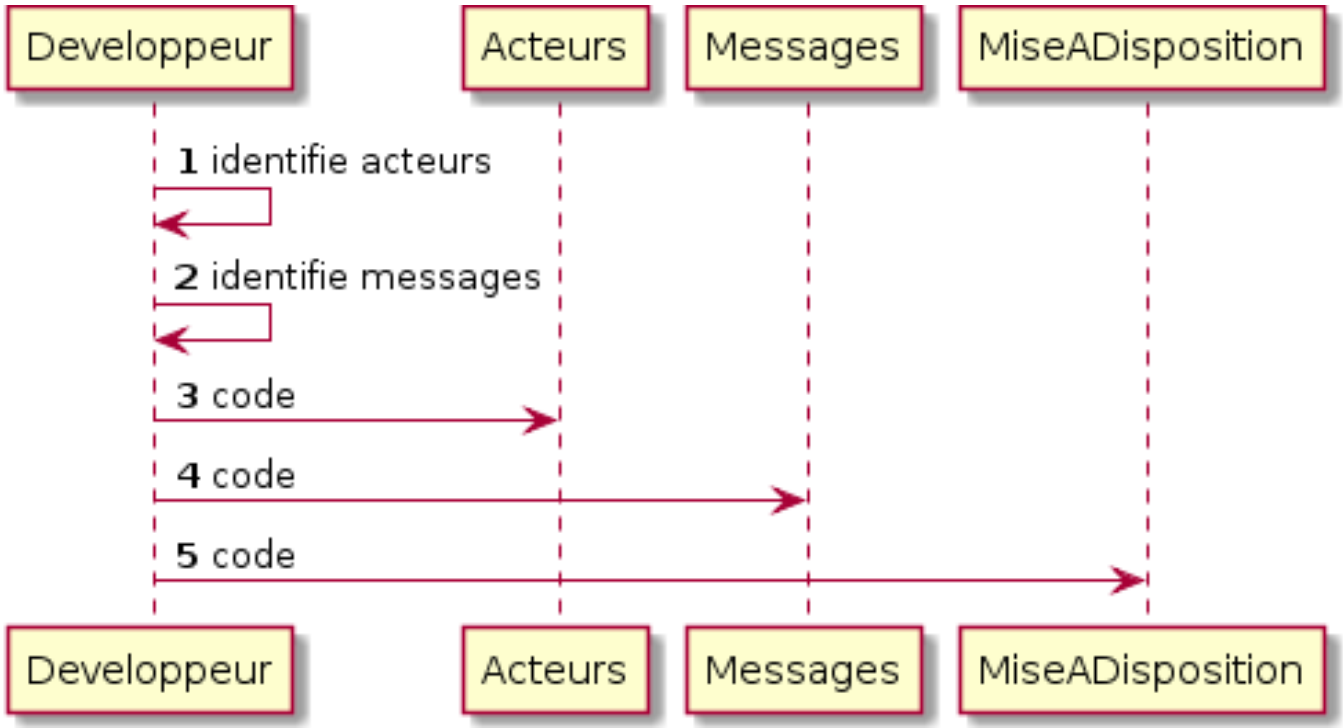
Entites en presence

Figure: Acteurs & entites



Cycle de developpement classique

Figure: Comment penser son application ?



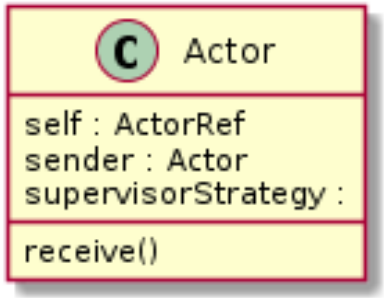
Acteurs et modelisation

Un acteur est une classe tres simple , reagissant a des evenements. Il convient de suivre quelques regles :

- heritage suivant le framework employe,
- redefinir les methodes idoines,
- tenter de conserver les acteurs *stateless*,
- tenter de faire en sorte qu'ils soient les plus simples possibles.

la classe Actor

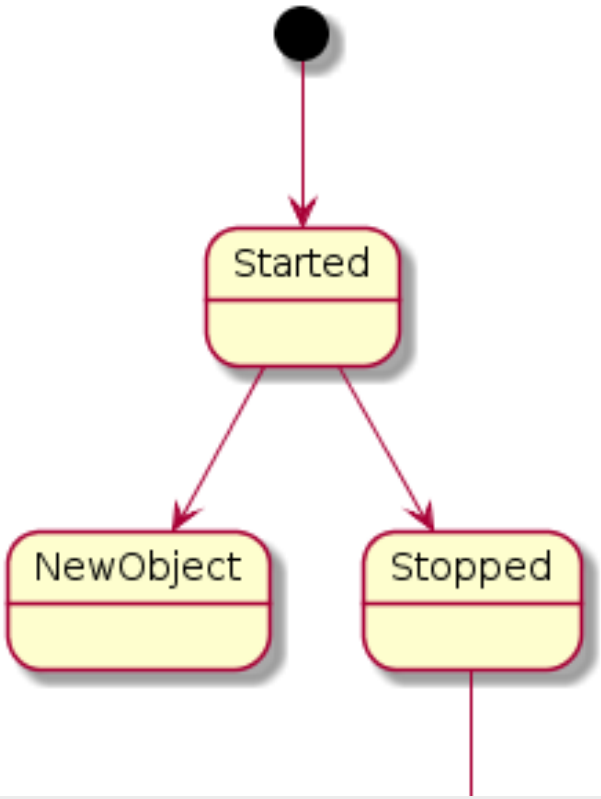
Figure: La classe Actor vue statique



Cycle de vie des acteurs

De nombreuses methodes se trouvent dans l'API permettant un controle total sur les transitions entre etats des acteurs. Methodes pre[StateName]post[StateName].

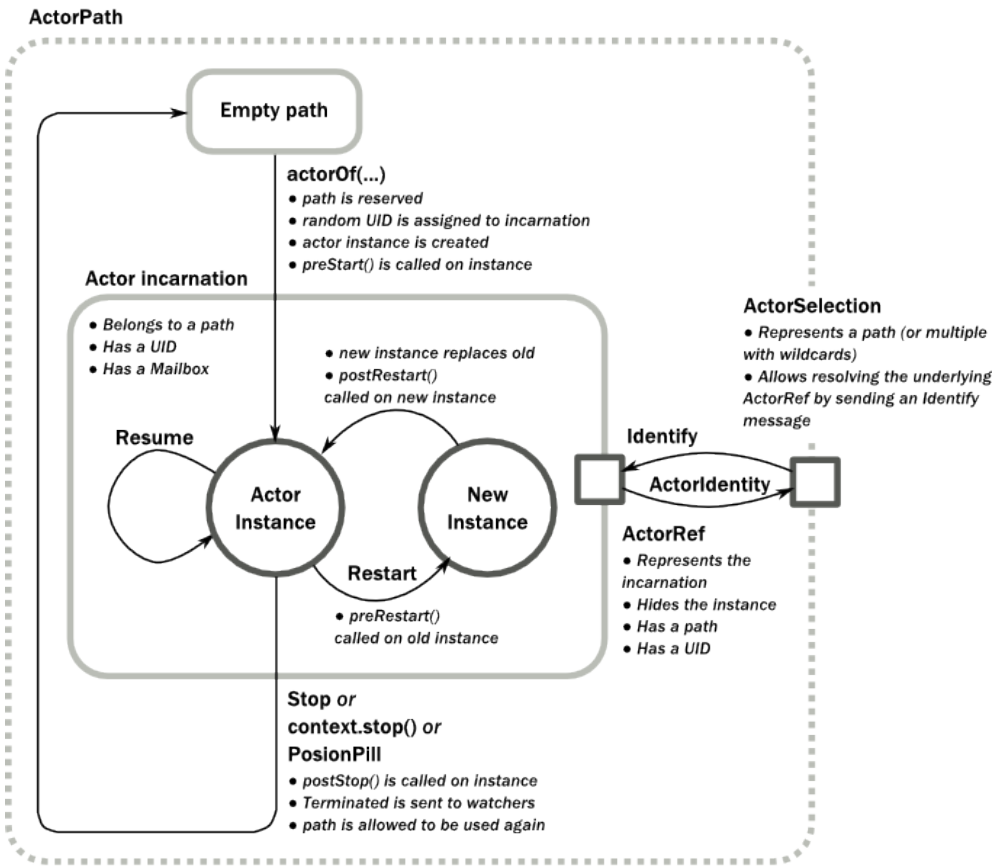
Figure: Cycle de vie



Cycle de vie des acteurs..Suite

Diagramme officiel extrait des documentations Akka officielles.

Figure: Cycle de vie



Acteurs et communication

Les acteurs adoptent d'office un mode de communication asynchrone :

1. jamais de blocage du systeme,
2. ideal en terme de montee en charge, deployer plus d'acteurs sur plus de machines induit une augmentation du throughput,
3. plus simple a gerer pour la supervision, les *timeouts* sont vite assimiles a des defauts applicatifss.

Akka & Composants

Akka est un ensemble de composants parmi lesquels :

1. une API
2. une console de supervision
3. un DSL

La partie client-serveur utilise Netty, le projet JBoss de serveur TCP NIO.

Droit au code

Apres cette introduction generale, regardons la mise en place d'unacteur avec Akka :

1. codage d'un acteur,
2. codage d'un message,
3. classe principale (manipulation du framework),
4. definition generale du projet SBT,
5. mise en place sur une machine.

Rien de tel qu'un bon vieux Hello World...

Un acteur basique

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
class HelloActor extends Actor {
  def receive = {
    case "hello" => println("hello")
    case _       => println("Sorry!!")
  }
}
```

Mise a disposition de l'acteur

```
object Main extends App {
val system = ActorSystem("HelloSystem")
// default Actor constructor
val helloActor = system.actorOf(
Props[HelloActor], name = "helloactor")
helloActor ! "hello"
helloActor ! "buenos␣dias"
}
```

Script SBT du projet

```
name := "Actor basic #1"

version := "1.0"

scalaVersion := "2.11.0"

resolvers += "Typesafe Repository" \
at "http://repo.typesafe.com/typesafe/releases/"

libraryDependencies += "com.typesafe.akka" % "akka-acto
% "2.3.6"
```

Ce qu'il faut retenir...

Points essentiels mis en evidence par l'exemple :

1. heriter du trait *Actor*,
2. redefinir la methode *receive*, indiquant la reaction adaptee a chaque type de message,
3. envoi de messages entre acteurs via !

Introduction

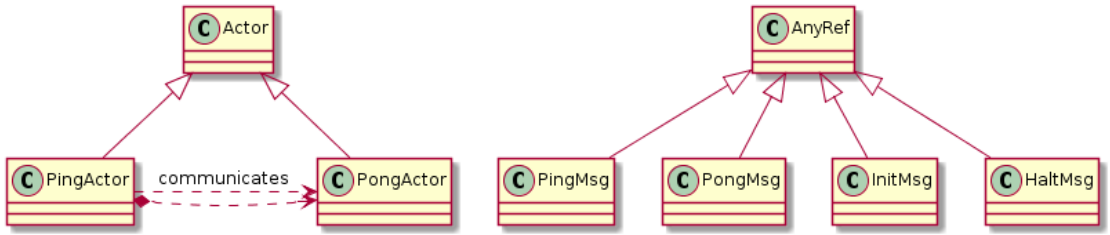
Nous allons mettre en place un ping-pong avec Akka. Les entites en lice sont :

- un acteur **Ping** reagissant aux messages **Pong** et **Halt**
- un acteur **Pong** reagissant aux messages **Ping** et **Init**
- une classe mettant a disposition les objets et initialisant le dialogue

Au bout de *n* echanges le dialogue est rompu.

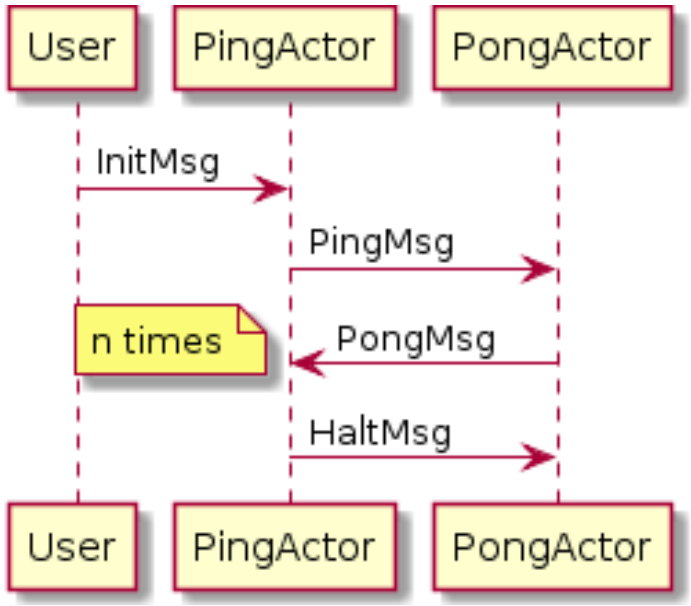
Modelisation

Figure: Acteurs & entites



Dialogue

Figure: Dialogue entre acteurs



Les messages

```
package pingpong.msg
case object InitMsg
case object PongMsg
case object HaltMsg
case object PingMsg
```

Acteur Pong

```
package pingpong.actors
import akka.actor.*
import pingpong.msg.*
class PongActor extends Actor {
  var counter = 0
  def newPing (sender : ActorRef) = {
    counter+=1
    println("received a new Ping")
    if(counter<3)
      sender ! PongMsg
    else{
      println("Max value reached ..!")
      sender ! HaltMsg} }
```

Acteur Pong..suite

```
  def receive = {
    case PingMsg =>
      newPing(sender)
    case _ => println("oops!!!")
  }
}
```

Acteur Ping..

```
class PingActor(pong : ActorRef) extends Actor {
  var started = false
  def receive = {
    case InitMsg =>
      started = true
      println( "Ok starting with a Ping to Pong")
      pong ! PingMsg
    case PongMsg =>
      println("received Pong messag")
      if(!started)
        println("Discarded not yet started !!")
      else{
        sender ! PingMsg
        println ( "replying to Pong with Ping")}
```

```
case HaltMsg =>
      started = false
      println ( "HaltMsg received")
      //context.stop(self)
      case _ => println( )}}
```

Programme principal

```
// imports omis
object Main extends App {
  println("Starting␣PingPong")
  val system = ActorSystem("PingPongSystem")
  val pong = system.actorOf(Props[PongActor]
  , name = "pong")
  val ping = system.actorOf(Props
  (new PingActor(pong)), name = "ping")
  println("Sending␣InitMsg")
  ping ! InitMsg
}
```

Execution du programme

```
Starting PingPong
Got the actors references ...Sending InitMsg
Ok starting with a Ping to Pong
received a new Ping
received Pong messag
received a new Ping
replying to Pong with Ping
received Pong messag
replying to Pong with Ping
received a new Ping
Max value reached ..!!!Leaving...
HaltMsg received - toggled flag
```

Ce qu'il faut retenir

Points a souligner :

- les messages sont simplistes et ne contiennent pas d'information d'ou utilisation de **case object**,
- possibilite de stopper le programme en supprimant la reference d'acteur via *context.stop(self)*,
- la reference sur l'objet courant self,
- le fonctionnement proche de RMI JNDI (**annuaire**),

Introduction

Akka est *extrememement configurable* :

- utilise TypeSafe Config le framework de TypeSafe :
  - syntaxe proche de JSON et de l'EDN Clojure
  - donc proche de JSON
  - couples cles & valeurs hierarchisees
- permet de specifier de nombreux parametres :
  - pooling (ExecutorService)
  - routage
  - specification du typage des messages achemines sur les acteurs
  - deploiement
  - supervision

Configurer : dans quels cas ?

Pour des besoins avances ou pour tester des cas specifiques :

- developpement sur plusieurs noeuds,
- diverses instances sur la meme machine (developpement & recette),
- pour ameliorer la robustesse de votre solution.

Par defaut, nul besoin de configuration explicite.

Configuration exemple

```
akka {
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  loglevel = "DEBUG"
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
    default-dispatcher {
      throughput = 10
    }
  }
  remote {
    netty.tcp.port = 4711
  }
}
```

Introduction

La console Akka permet de superviser son application et de monitorer le fonctionnement de celle-ci en :

- echantillonant les reactions de l'application dans le cas d'envoi de messages,
- stockant les resultats de cet echantillonnage dans une base de donnees,
- en offrant une restitution visuelle.

Concepts

L'echantillonnage realise par la console n'est que du code insere directement dans votre **bytecode** par un framework d'**AOP**.

Consequence

Il faudra donc activer et configurer explicitement les acteurs concernes par ces mesures et la frequence de celle-ci.

Configuration sur une application

Il faudra rajouter dans votre *application.conf* un bloc du type.

```
        atmos {
        trace {
          enabled = true
          event-handlers =
          ["akka.atmos.trace.store.mongo.
␣␣␣␣␣␣␣␣MongoTraceEventListener"]
          mongo {
            db-name = "atmos-monitoring"
            db-connection-uri = "mongodb://localhost"
          }
        }
    }
```

On aurait pu parler de

Voila pour cet examen detaille du framework qui n'aborde pas tout l'ensemble des possibilites offertes par Akka car on aurait pu aussi :

- parler de la personnalisation du framework avec les possibilites d'implementations de divers traits offerts par le framework,
- se pencher plus sur la personnalisation,
- entrer dans la mecanique de reprise sur erreur,
- aborder la performance, l'audit ...
- etc...

Questions & Commentaires

Remarques bienvenues... Des questions ?

Skype en Scala - Partie 1

Le sujet de ce TP est de construire les briques d'un moteur de messagerie instantanee. Pour cela il sera bon de :

- lister les acteurs en lice,
- lister les messages echanges,
- bref suivre la methode vue dans cette section.

Implementer et tester cette premiere partie dans la VM.

Skype en Scala - Partie 2

Ici nous allons passer a un deploiement en reseau des acteurs. Pour tester cette partie il faudra modifier la configuration de la VM :

1. stopper la VM
2. cliquer bouton droit dans VirtualBox
3. aller dans preferences/settings
4. selectionner le menu reseau
5. ajouter une seconde carte reseau (une par defaut em mode NAT)

Si la configuration de la salle le permet on devrait pouvoir communiquer entre VMs dans la salle via une configuration en mode **bridge**. Si celle-ci ne fonctionne pas et que les machines sont assez puissantes on peut tenter de lancer 2 VMs sur la meme machine hote en mode **reseau prive**.

Skype en Scala - Partie 3

Pour ceux motives pour arriver ici, nous allons ajouter ici :

- supervision avec la console Akka,
- tests de diverses strategies,

Demarrage

1. lancer VirtualBox
2. importer la VM (.ova)
3. une fois le processus termine, cliquer sur la VM apparue dans la liste et cliquer sur start,
4. apres quelques secondes, un ecran de connexion apparait
5. utiliser **user** et testp pour vous connecter

Travailler avec la VM

Il s'agit d'un Linux base sur du Debian stable (wheezy). L'ensemble des documents, des exemples et des binaires lies a Scala sont rassembles dans le dossier **platform** dans le compte utilisateur.

```
cd ~
cd platform
ls -la
```

Installer des paquets

Si certains logiciels manquent a l'appel vous pouvez en installer :

- dans un terminal par **apt-get** ou **aptitude**
- graphiquement par **synaptic** disponible dans les menus (bouton droit sur le bureau)

```
>aptitude search scala
>sudo aptitude install scala
```