Sorbonne Université
Paradigmes de Programmation Concurrente 5I553

Cours 9 - Le Modèle Actor
Akka

Carlos Agon

3 décembre 2018

Exemple 1 : l'acteur de la semaine

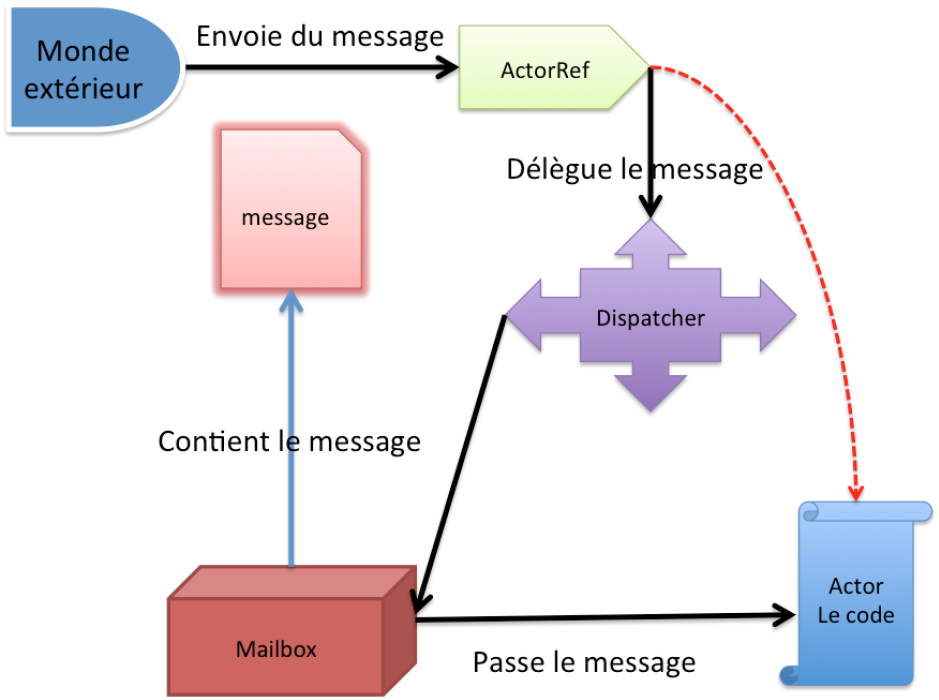
Quelques remarques :

- On n'est pas obligé de communiquer uniquement avec des strings
- Notre `Actors` ne fait rien. C'est vrai qu'ils font peu, mais ici il ne fait rien !
- `Thread.sleep`.
- `Terminaison`.

Propriétés des acteurs

- ils sont réactifs,
- ils ne font qu'une chose à la fois,
- on ne peut voir rien à l'intérieur,
- ils sont disponibles et on peut les trouver facilement.

Components d'un acteur



- ActorRef : on ne communique jamais avec l'actor directement. Il contacte le dispatcher pour mettre le message dans la mailbox.
- Dispatcher : Met le message dans un thread
- Mailbox : A l'execution, il enlève un ou plusieurs mails et les envoie à l'actor

Communication : messages

Le message hérite de ANY

```
case class Gamma(g: String)
case class Beta(b: String, g: Gamma)
case class Alpha(b1: Beta, b2: Beta)

class MyActor extends Actor{
 def receive = {
        case "Hello" =>
                println("Hi")

        case 42 =>
                println("I don't know the question.")

        case s: String =>
                println(s"You sent me a string: $s")

        case Alpha(Beta(b1, Gamma(g1)),  Beta(b2, Gamma(g2))) =>
                println(s"beta1: $b1, beta2: $b2, gamma1: $g1, gamma2: $g2")

        case _ =>
                println("Huh?")
 }
}
```

Communication : send

```
object MySequencedObject {
        def doSomething(withThis: String){
                // ... calcule  une heure ...
        }
}

MySequencedObject.doSomething("Hello !")
println("Hi !")
```

Traditionnellement on ne voit "Hi !" que quand `doSomething` se termine (1h après).

Ce n'est pas le cas pour les Actors

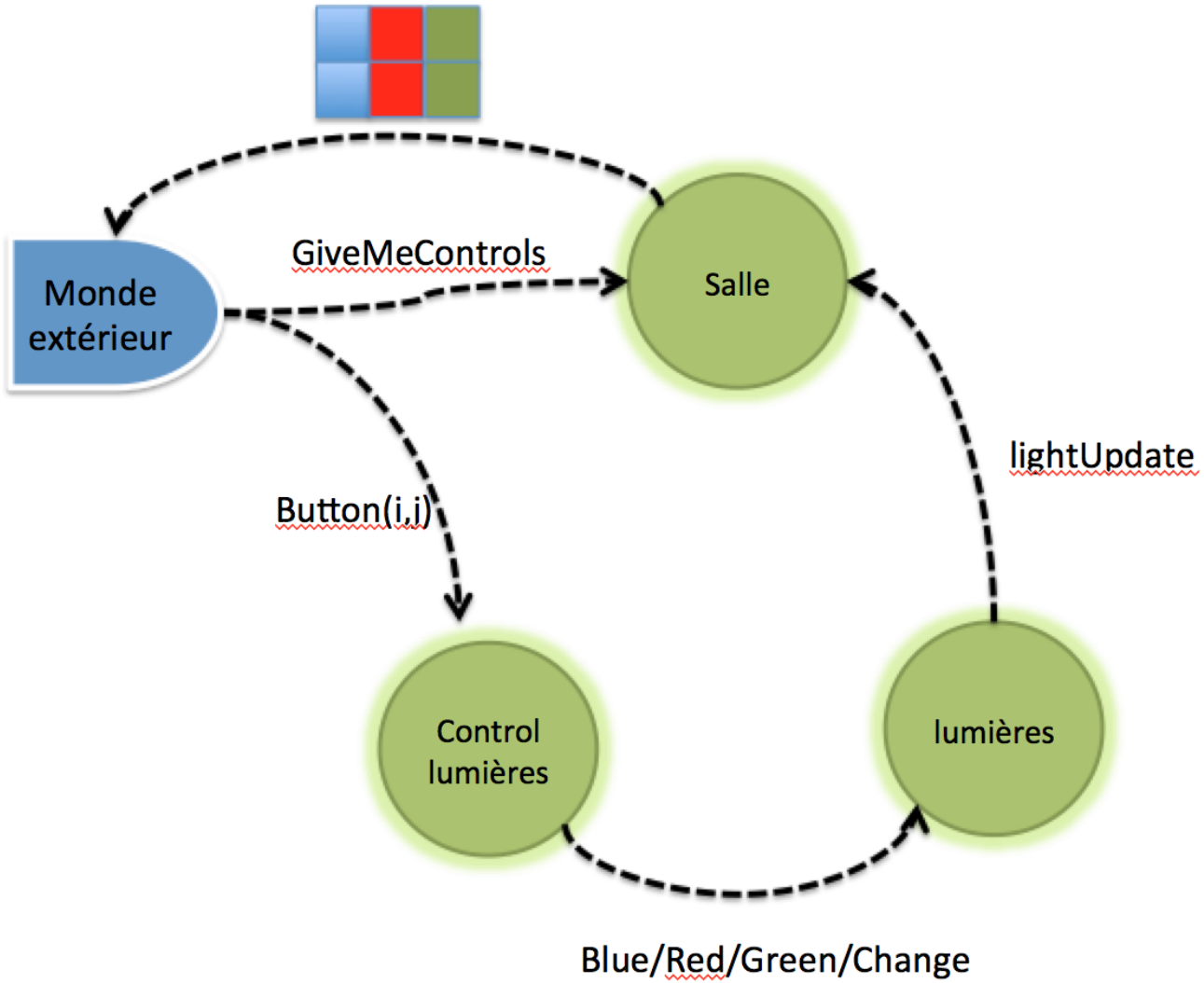
```
case class DoSomething(withThis: String)

class MyConcurrentObject extends Actor {
        def receive ={
                case DoSomething(withThis) =>
                                // ... calcule  une heure ...
        }
}

system.actorOf(Props[MyConcurrentObject]) ! DoSomething("With this")
println("Hi !")
```

On voit "Hi !" tout suite. L'envoi d'un message est asynchrone

Création d'un Actor

```
import akka.actor.{Props, Actor, ActorSystem}

class MyActor extends Actor { ...}

//la racine d'un group d'actors
val system = ActorSystem("MyActors")

//pour parametrer la structure de l'actor
//e.g. le contexte d'evaluation
val actorProps = Props[MyActor]

//retourne un actorRef
val actor = system.actorOf(actorProps)
```

Exemple 2 : des lumières



Comment fonctionnent les messages ?

```
def ! (message: Any)(implicit sender: ActorRef = null): Unit
```

Quelques remarques :

- ! est une fonction avec effet secondaire
- le message peut être n'importe quoi
- `implicit ActorRef`, ainsi on n'a pas à écrire le sender
- il y a un default value `null`

```
implicit val self: ActorRef
```

Pour accéder au sender on a :

```
def sender: ActorRef
```

Mais, attention sender est une méthode :

```
case SomeMessage =>
 context.system.scheduleOnce(5 seconds) {
 sender ! DelayedResponse
}
```

```
case SomeMessage =>
 val requestor = sender
 context.system.scheduleOnce(5 seconds) {
 requestor ! DelayedResponse
}
```

null sender et forwarding

null sender

- Si on répond à un null sender on envoie dans un actor particulier appellé `dead letter office`
- `dead letter office`, est un actor simple de l'`ActorSystem` qui peut être accedé en utilisant la méthode `deadLetters`.

Forwarding

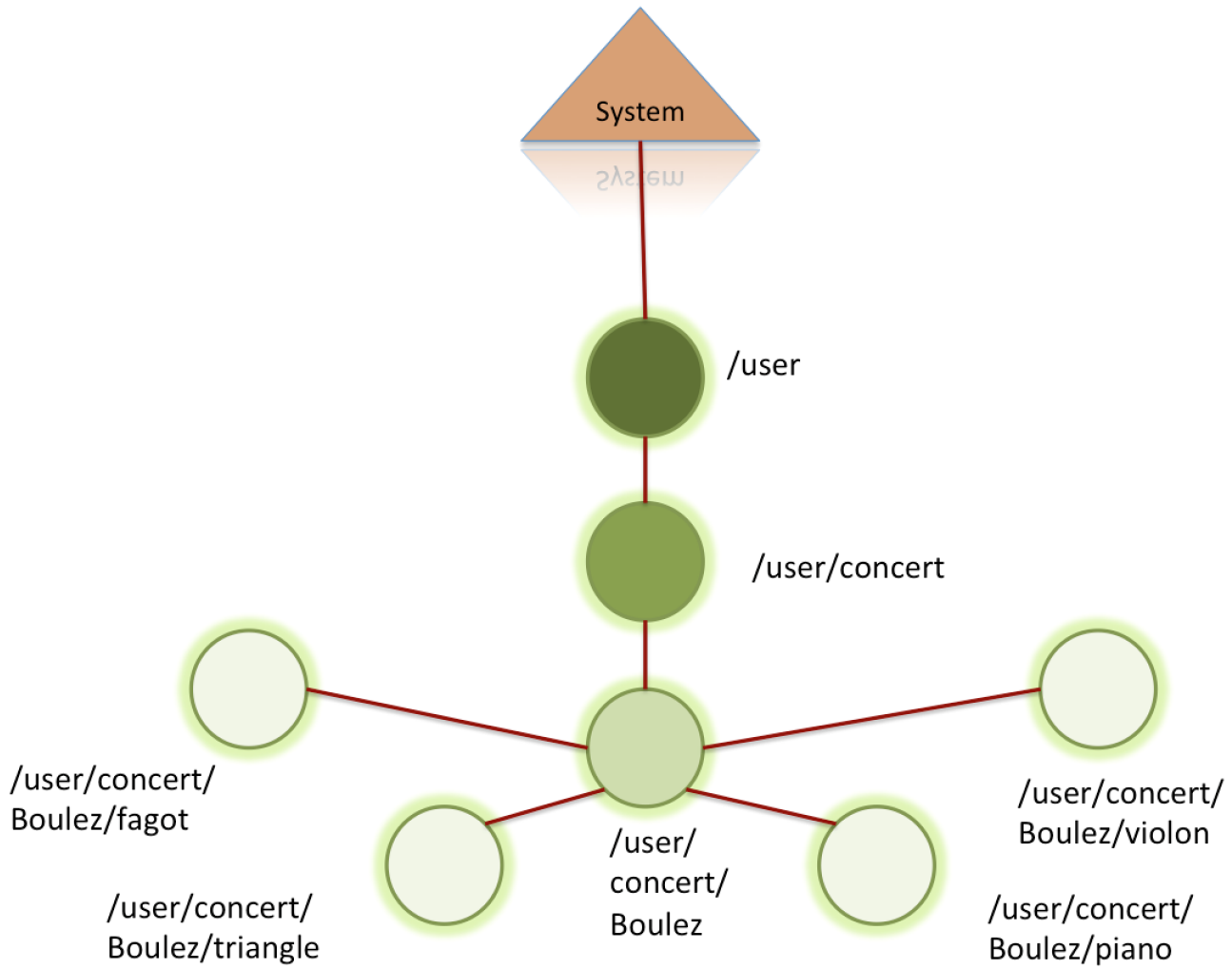
- Si A envoie à B et B forward à C, alors C voit A comme son sender et non B.
- rien à voir avec :

```
case msg @ SomeMessage =>
   someOtherActor ! msg
```

- c'est plus un fwd de téléphone que fwd d'email

```
def forward(message: Any)(implicit context: ActorContext) =
   tell(message, context.sender)
```

Exemple 3 : l'orchestre

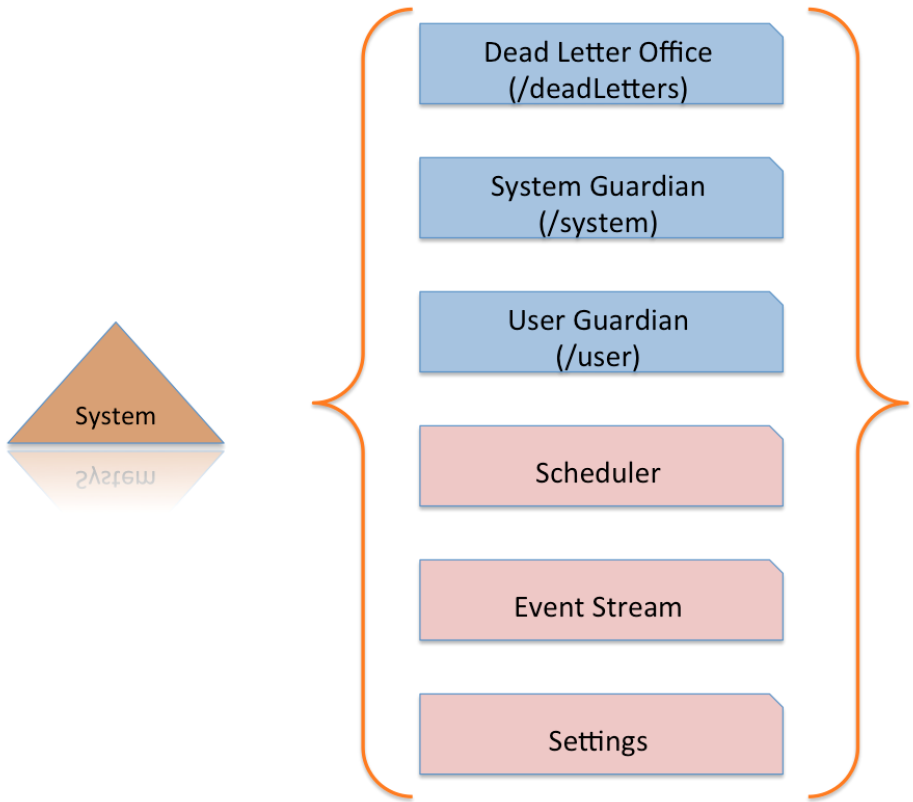


ActorSystem

Tout `Actor` a un ActorSystem à la racine de sa hierarchie. L'ActorSystem :

- n'est pas un `Actor`,
- garde une configuration,
- propose un scheduler,
- garde la dead letter office
- permet de localiser et manipuler d'autres actors

ActorSystem



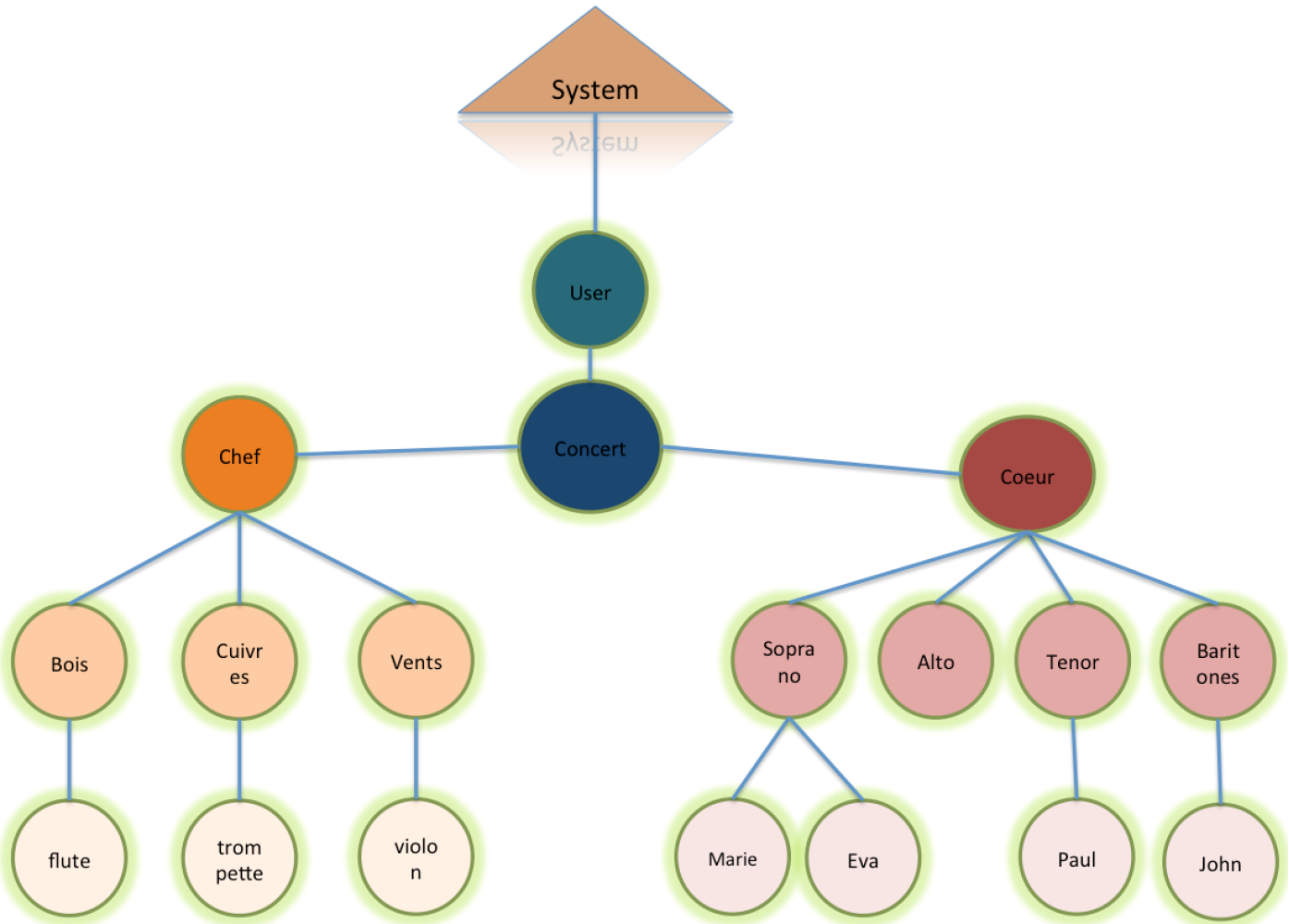
- `User Guardian Actor` est le parent de tout acteur.
- `Dead Letter Office` est un actor comme un autre
- `System Guardian Actor` le parent de tous les acteurs internes au système
- `Event Stream` e.g. Log,
- `Settings` e.g. Human-Optimized Config Object Notation (HOCON)

L'idée des Actors est d'avoir une petite foret avec de gross arbres

Paths

```
val system = ActorSystem("TheSystem")
val a = system.actorOf(Props(
                  new Actor
                    def receive = { case _ => } }),
                  "Actor")

println(a.path)
> "akka://TheSystem/user/Actor"

println(a.path.elements.mkString("/", "/", ""))
> "/user/Actor"

println(a.path.name)
> "Actor"
```

Exemple 4 : toute l'orchestre



Contexte

Le `ActorContext` permet :

- Actor Creation : tout en structurant la hiérarchie des Actors.
- System Access : à partir de n'importe quel point dans l'arbre.
- Relationship Access : entre parent, fils, frères, etc.
- State : access au sender par exemple.
- autres fonctionnalités

Chercher les Actors

```
actorSelection(path: Iterable[String]): ActorRef
```

A partir d'une collection e.g. List("/user", "/Concert", "/Conductor").

```
actorSelection(path: String): ActorRef
```

A partir d'un String e.g. "/../violon".

```
context.actorSelection("../*") ! msg
```

Exemple 4 : pi en distribué

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots = \frac{\pi}{4}$$