Sorbonne Université
Paradigmes de Programmation Concurrente 5I553

Cours 8 - Le Modèle Actor
Erlang - SCALA

Carlos Agon

17 novembre 2024

Historique

Carl Hewitt, Henry Baker "Actors and continuous functionals", MIT 1977.

Imposer des contraintes (des lois) pour tenir compte :

- des ressources partagéesw
- de la reconfiguration dynamique (mobilité)
- du parallélisme interne

Vision sociale (anthropomorphe, IA) du calcul

Actors et Agents

"An actor is a computational agent" Carl Hewitt, 1977.
Un agent (IA) est :

- autonome
- persistant
- intelligent (peut raisonner)
- mobile (traverse les machines)
- adaptable (peut apprendre)
- connaisseur (d'un domaine au moins)
- collaboratif (communique avec d'autres agents)
- possède une personnalité, de l'autorité, etc...

Actors et Agents : caractéristiques communes

- *Identité*
  Donnée par sa boîte aux lettres
- *Autonomie*
  self-contained (il a tout ce qu'il faut pour travailler),
  self-regulating (lui même contrôle comment réagir aux messages),
  self-directed (lui même contrôle son comportement)
- *Communication*
  Basée sur le message. Asynchrone. Distribution non déterministe.
  Chez les acteurs, le récepteur est connu. Chez les agents, la valeur du message peut déterminer le récepteur.
- *Coordination*
  d'un groupe d'acteurs/agents autour d'une intention particulière :
  - calcul, synchronisation et performance chez les acteurs
  - modélisation, partage de raisonnement et connaissance chez les agents.

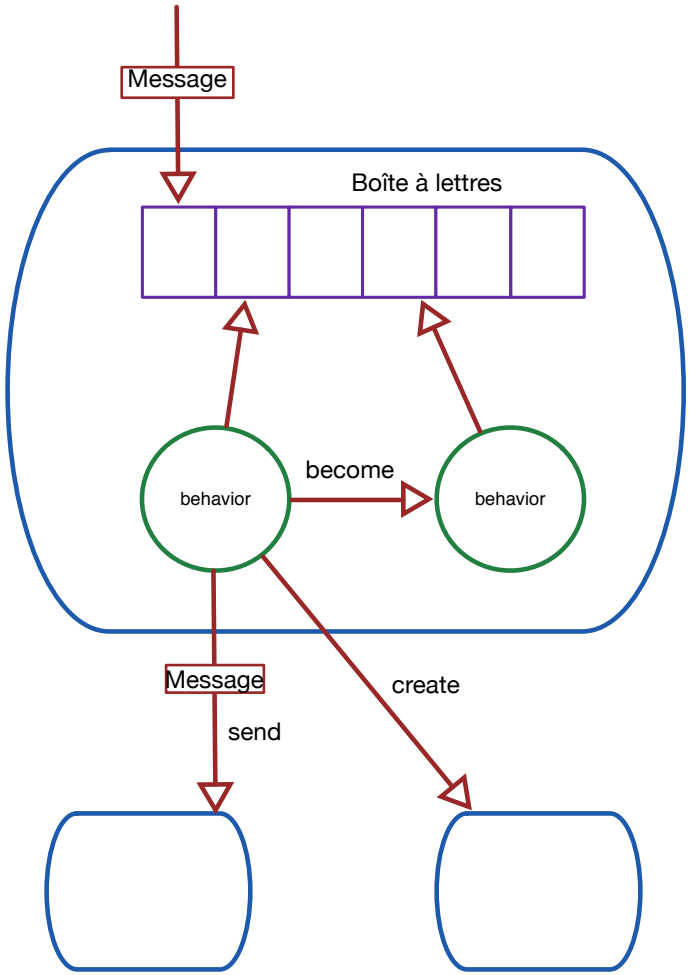
Pour quoi faire ?

- *Flexibilité*
  mettre différents groupes d'acteurs en contact (sans besoin de reprogrammation)
- *Adaptabilité*
  création dynamique d'acteurs pour répondre à une nouvelle situation
- *Ouverture*
  aux acteurs implantés dans des langages différents ou sur différents systèmes d'exploitation

Concurrence

G.Agha et C. Hewitt. "A Concurrent Programming Using Actors", MIT 1985.

- Concurrence inter-acteurs
- Concurrence intra-acteurs (parallélisme interne)



Formalisme original

G.Agha et C. Hewitt. "A Concurrent Programming Using Actors", MIT 1985.

- Acteur : on nomme acteurs les entités primitives du modèle d'acteurs. Tous les objets sont des acteurs. L'ensemble des acteurs est noté A^*
- Evénement : Un événement E est un pair (M, T) où M et T sont deux acteurs appelés respectivement message et cible. On peut noter aussi $E[T \leftarrow M]$.
  Un événement correspond intuitivement à l'arrivée d'un message M à une cible T.
  Les événements sont ainsi les étapes structurelles de l'exécution dans un langage d'acteurs.
- Histoire de l'exécution : On définit un ensemble $\mathcal{H} = \mathcal{P}(E^*)$ que l'on appelle *l'histoire d'une exécution* (l'ensemble des événements ayant eu lieu au cours de l'exécution d'un programme).
- Continuation : une continuation correspond à l'acteur auquel la réponse au message doit être envoyée. L'événement peut ou non avoir une continuation, mais si elle existe, elle est unique.

Axiomes

1. Chaque acteur ne peut communiquer qu'avec un nombre fini d'acteurs au cours d'une exécution.
2. Principe de stricte causalité : $\forall E \in E^*, \neg E \xrightarrow{act} E$.
   $\xrightarrow{act}$ est appelée l'ordre d'activation. $E1 \xrightarrow{act} E2$ si l'envoie du $M1$ à $T1$ aura par consequence l'envoie de $M2$ à $T2$ ($E2$ aura lieu).
3. Pour toute $E1$, $E2$ il n'existe une chaine infinie $E1 \xrightarrow{act} ... \xrightarrow{act} Ei... \xrightarrow{act} ...E2$
   Cet axiome empêche les machines de calculer infiniment vite, c'est-à-dire de réaliser une infinité de tâches en un temps fini.

Langages fonctionnels à acteurs

1. Amber (Cardelli, 86)
2. Facile (Giacalone, et al. 89)
3. CML (Reppy, 91)
4. Erlang (Armstrong, 93)
5. Obliq (Cardelli, 94)
6. Pict (Pierce et Turner, 94)
7. Scala (Odersky, 03)

Erlang

```
-module(math1).
-export([factorial/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1)
```

Erlang

```
-module(math3).
-export([area/1]).

area({square, Side}) -> Side * Side;
area({rectangle, X, Y}) -> X * Y;
area({circle, Radius}) -> 3.14159 * Radius * Radius;
area({triangle, A, B, C}) ->
                  S = (A + B + C)/2,
                        math:sqrt(S*(S-A)*(S-B)*(S-C))
```

```
>Thing = {triangle, 6, 7, 8}.
{triangle,6,7,8}

> math3:area(Thing).
20.3332
```

Filtrage de motifs sur les listes

```
map(Func, [H|T]) -> [apply(F, [H])|map(Func, T)];
map(Func, []) ->   [].
```

```
> lists:map({math1,factorial}, [1,2,3,4,5,6,7,8]).
[1,2,6,24,120,720,5040,40320]
```

Erlang Concurrence

Création d'un acteur spawn/3

```
Pid = spawn(Module, Function, ArgumentList)
```

- L'acteur finit avec la fin de `Function`.
- La valeur de `Function` est perdue.

Communication entre acteurs

Send

```
Pid ! Message
```

-Send est asynchrone.
-Même si le récepteur n'existe plus, l'expéditeur ne le saura pas.
-Pour un récepteur, l'ordre d'envoi est toujours respecté.

```
foo(12) ! bar(baz)
```

Tous les acteurs ont une `mailbox` qui garde les messages dans l'ordre de réception.

```
Receive
  Message1 [when Guard1] -> Actions1 ;
  Message2 [when Guard2] -> Actions2 ;
  ...
end
```

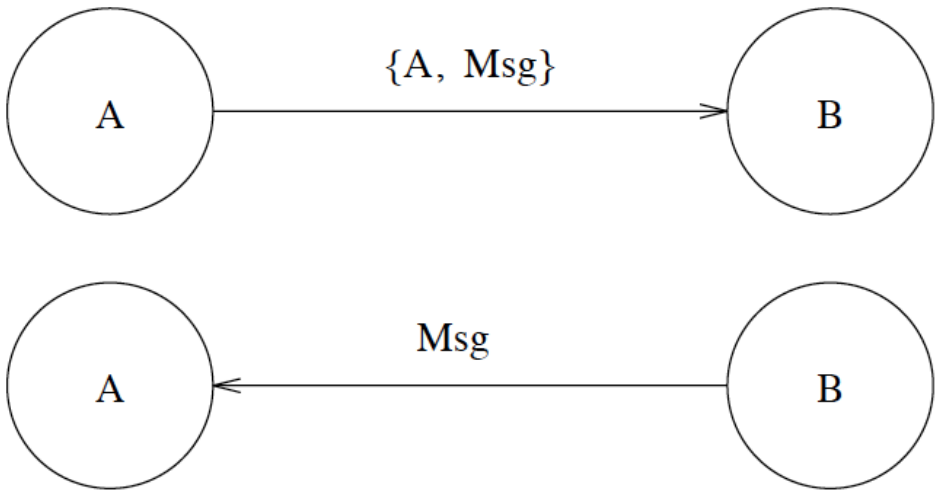
Si un message n'est pas lu, il reste dans la mailbox jusqu'au prochain `receive`.

Echo Actor

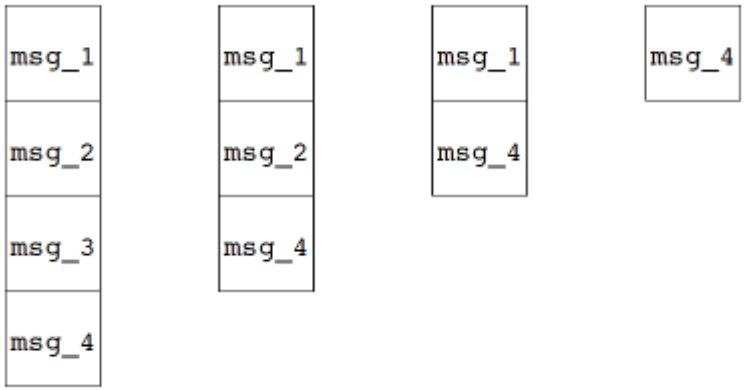
```
-module(echo).
-export([start/0, loop/0]).

start() ->
  spawn(echo, loop, []).

loop() ->
  receive
    {From, Message} ->
        From ! Message,
        loop()
  end.
```



Ordre de réception des messages

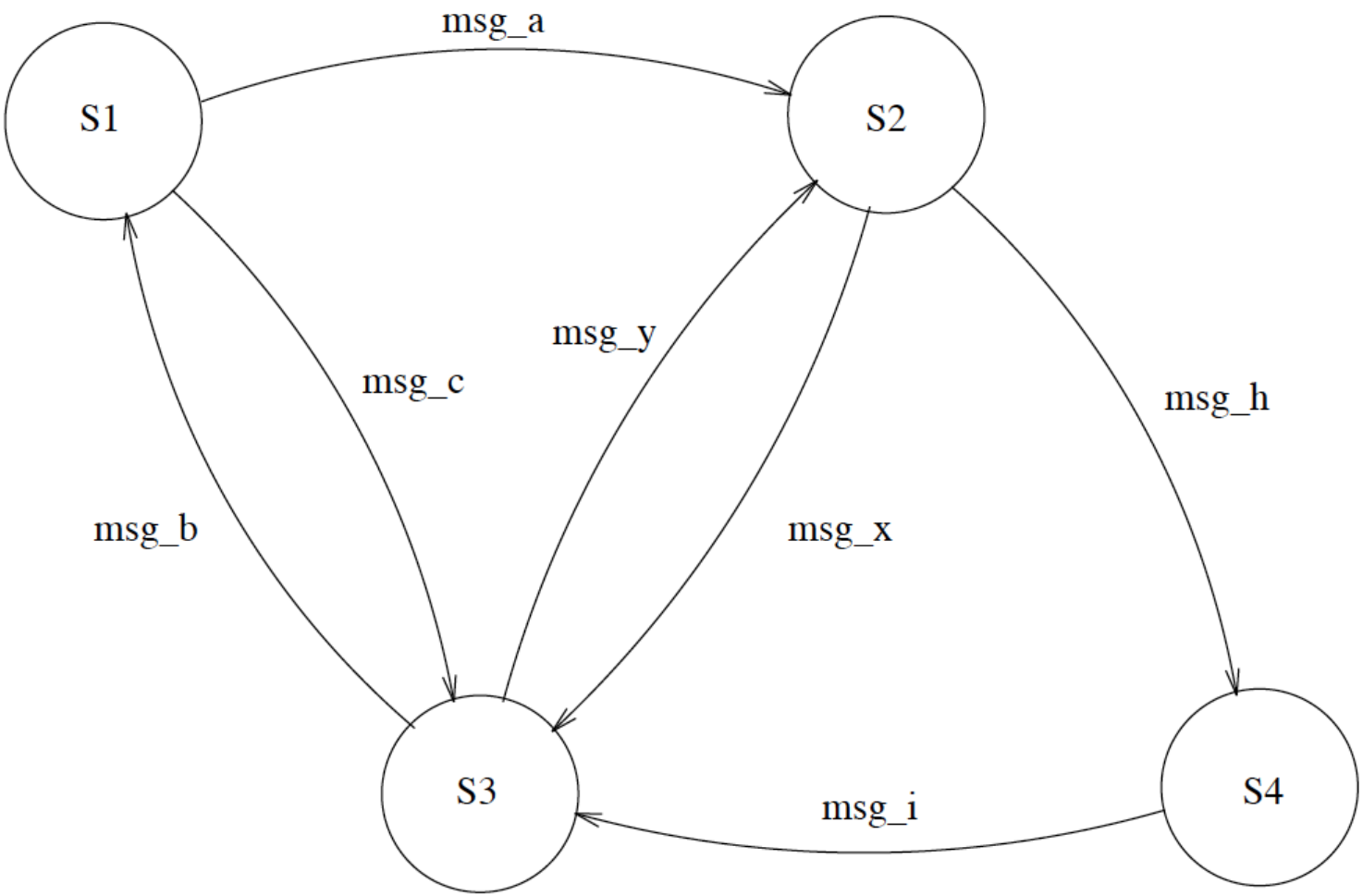



```
receive
 msg_3 ->
 ...
end

receive
 msg_4 ->
  ...
 msg_2 ->
   ...
end

receive
 AnyMessage ->
 ...
end
```

Une FSM



State Machines

```
s1() ->
  receive
    msg_a ->
      s2();
    msg_c ->
      s3()
    end.
s2() ->
  receive
    msg_x ->
      s3();
    msg_h ->
      s4()
  end.
s3() ->
  receive
    msg_b ->
      s1();
    msg_y ->
      s2()
  end.
s4() ->
  receive
    msg_i ->
      s3()
    end.
```

Programmation répartie

```
spawn(Node, Mod, Func, Args)
```

Pour créer un actor dans un node remote.

```
spawn_link(Node, Mod, Func, Args)
```

Pour créer un actor sur un node et faire un lien avec le node *current*

```
monitor_node(Node, Flag)
```

Si Flag = true, le node current devient le monitor de node. Si node crash le mode current est informé.
Si Flag = false, on enlève le lien de monitor.

```
node()
```

retourne le nom du node current

```
nodes()
```

retourne la liste de nodes connus par node()

```
node(Item)
```

retourne le node Item

Catch and Throw

```
foo(1) ->
   hello;
foo(2) ->
   throw({myerror, abc});
foo(3) ->
   tuple_to_list(a);
foo(4) ->
    exit({myExit, 222})
```

- Eval `foo(1)` retourne hello.
- Eval `foo(2)`finit avec une erreur
- Eval `foo(3)` finit avec une erreur, *a* n'est pas une tuple.
- Eval `foo(4)` retourne `{myExit, 222}`

Catch and Throw

```
demo(X) ->
    case catch foo(X) of
        {myerror, Args} ->
            {user_error, Args};
        {'EXIT', What} ->
            {caught_error, What};
        Other ->
            Other
end.
```

- Eval `demo(1)` retourne hello.
- Eval `demo(2)` retourne `{user_error, abc}`
- Eval `demo(3)` retourne `{caught_error, badarg}`
- Eval `demo(4)` retourne `{caught_error, {my exit, 222}}`

Terminaison des processus

```
-module(test).
-export([process/0, start/0]).
start() ->
   register(my_name, spawn(test, process, [])).
process() ->
   receive
      {stop, Method} ->
            case Method of
                return ->
                    true;
                Other ->
                    exit(normal)
            end;
        Other ->
            process()
    end.
```

- Eval `my_name ! stop, return` évalue true et finit normalement.
- Eval `my_name ! stop, hello` finit normalement
- Eval `my_name ! any_other_message` le processus ne finit jamais

Liens entre processus

- `links` entre processus
- `EXIT` signals

Quand un processus finit (normal ou anormalement), il envoie un signal à tous les processus auxquels il est lié.
Ce signal a la forme :

```
'EXIT', Exiting_Process_Id, Reason
```

Par défaut, si on reçoit un signal tel que `Reason` est différent de `normal`, le processus qui reçoit termine et envoie un signal à tous les autres processus auxquels il est connecté.

- `link(pid)` fait un lien bidirectionnel entre `current` et pid
- Quant un processus finit, il coupe tous les liens.
- `unlink(pid)` coupe le lien dynamiquement
- si `link()` et `unlink()` sont des appels incohérents, ce n'est pas grave.

```
spawn_link(Module, Function, ArgumentList) ->
    link(Id = spawn(Module, Function, ArgumentList)),
    Id.
```

Scala et Java

Scala et Java

```
import java.util.{Date, Locale} //plusieures classes d'un package
import java.text.DateFormat._   //import tout le package

object FrenchDate {
  def main(args: Array[String]) {
   val now = new Date
   val df = getDateInstance(LONG, Locale.FRANCE)
   println(df format now) // df.format(now)
  }
```

On peut aussi définir un objet qui implémente le trait App.

```
object HelloWorld extends App {
  println("Hello, world!")
}
```

Scala Fonctions

```
object Timer {
   def oncePerSecond(callback: () => Unit) {
      while (true) { callback();  Thread sleep 1000 }
     }

  def timeFlies() {
      println("message 1")
    }

  def main(args: Array[String]) {
     oncePerSecond(timeFlies)
     oncePerSecond(() => println ("message 2"))
}
```

Scala Classes

```
class Complex(real: Double, imaginary: Double) {
//Methods sans arguments
  def re = real
  def im = imaginary
  override def toString() =
            "" + re + (if (im < 0) "" else "+") + im + "i"
}

object ComplexNumbers {
  def main(args: Array[String]) {
    val c = Complex(1.2, 3.4) //new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

Scala Case Classes

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree

type Environment = String => Int
val env: Environment = { case "x" => 5 case "y" => 7 }

def eval(t: Tree, env: Environment): Int =
t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n) => env(n)
  case Const(v) => v
}
```

Scala Case Classes

```
def main(args: Array[String]) {
 val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
 val env: Environment = { case "x" => 5 case "y" => 7 }
 println("Expression: " + exp)
 println("Evaluation with x=5, y=7: " + eval(exp, env))
}
```

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
```

Scala Traid

```
trait Ord {
def < (that: Any): Boolean
def <=(that: Any): Boolean = (this < that) || (this == that)
def > (that: Any): Boolean = !(this <= that)
def >=(that: Any): Boolean = !(this < that)
}


class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d

  def <(that: Any): Boolean = {
  if (!that.isInstanceOf[Date])
    error("cannot compare " + that + " and a Date")

  val o = that.asInstanceOf[Date]
  (year < o.year) ||
  (year == o.year && (month < o.month ||
                      (month == o.month && day < o.day)))
}
```

Scala Génériques

```
class Reference[T] {
private var contents: T = _
def set(value: T) { contents = value }
def get: T = contents
}


object IntegerReference {
def main(args: Array[String]) {
 val cell = new Reference[Int]
 cell.set(13)
 println("Reference contains the half of " + (cell.get * 2))
}
}
```

Références

- "A Scala Tutorial for Java Programmers" By Michel Schinz and Philipp Haller (http ://docs.scala-lang.org)
- "Concurrent Programming in ERLANG", J. Armstrong, R. Virding, C. Wikstrom, M. Williams