Modèle acteur et Scala

Merwan Achibet

Université du Havre

Vendredi 24 février 2012

Acteurs

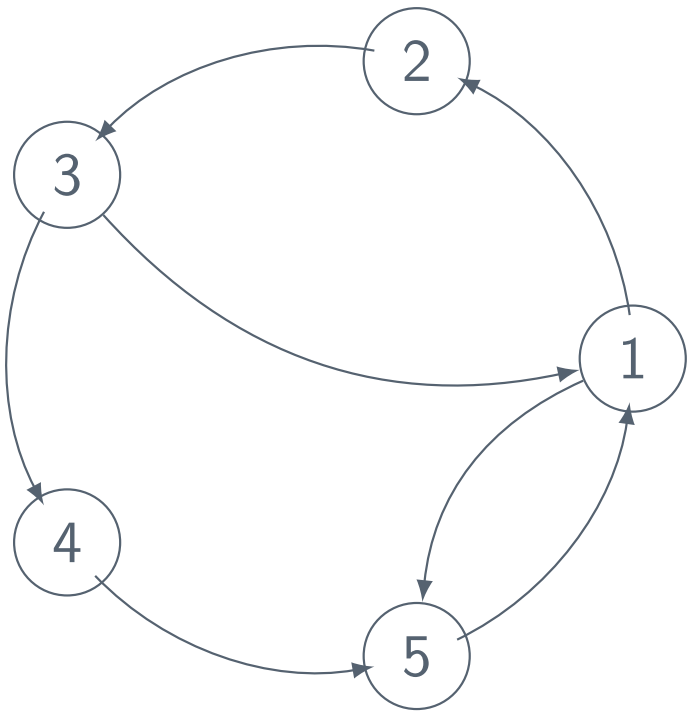
Processus concurrent communiquant avec d'autres acteurs par échange de messages. Un acteur peut répondre à un message asynchrone en créant un nouvel acteur, en envoyant des messages ou en changeant de comportement. [HO09]

Idées conductrices :

- Tout est acteur
- Asynchronisme
- Fault tolerance

Motivations

- Parallélisation croissante du matériel
- Distribution des calculs
- Un paradigme structuré autour de ces idées



Modèle acteur au sens strict [KSA09]

State encapsulation

Aucun partage de donnée hormis les messages.

Safe-messaging

Les messages contiennent des copies strictes.

Mobility

Le code et l'état d'un agent peuvent se déplacer entre processeurs, nœuds d'un réseau...

Location transparency

Quelle que soit sa position, un agent dispose de la même adresse.

JVM et concurrence

Thread-based

+ Simple
- Lourd
- Deadlock

Event-based

+ Performant
- Vite tortueux

Une couche supplémentaire

Le langage Scala, de plus haut niveau, cache ces considérations techniques et permet de traiter la concurrence via le modèle agent.

Origines

- Créé en 2003 à l'EPFL
- Académique
- Pragmatique
- Versatile

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Un langage orienté objet...

```
object App {

        class Personne(nom: String) {

                def parle(phrase: String) {
                    println(nom + " : " + phrase)
                }
        }

        var moi = new Personne("Merwan")
        moi.parle("Bonjour !")
}
```

... mais aussi fonctionnel

```
def sort(xs: Array[Int]): Array[Int] = {

    if (xs.length <= 1)
       xs
    else {

         val pivot = xs(xs.length / 2)

         Array.concat(
               sort(xs filter (pivot >)),
               xs filter (pivot ==),
               sort(xs filter (pivot <)))
    }
}
```

Le modèle acteur avec Scala

```
class MonActeur extends Actor {

  ...

  def act() {

    ...

  }
}
```

Un acteur hérite de la classe `Actor` et doit définir une méthode `act()`.

Communication entre acteurs

```
class E(r : Actor) extends Actor {
  def act() {
    receveur ! "Salut"
...
```

```
class R() extends Actor {
  def act() {
    loop {
      react {
        case message : String =>
          Console.println(message)
          sender ! "Bonjour"
    }
...
```

Exemple de la partie de dés

Règles du jeu

- Deux acteurs joueurs
- Un acteur arbitre
- le joueur qui tire le plus grand n gagne $(1 \geq n \geq 6)$

Déroulement de la partie

1. L'arbitre informe les joueurs du début de la partie
2. Les joueurs tirent un n au hasard et l'envoient à l'arbitre
3. L'arbitre détermine le gagnant

On commence par définir les types de messages possibles.

```
case object Jouez
case object Fin
case class Choix(id: Integer, x : Integer)
```

Le comportement des joueurs :

```
class Joueur(id : Int) extends Actor {

  def act() {

    Console.println(id + " : je rentre dans la partie")

    loop {
      react {

        case Jouez =>
          var random = new Random
          var c = random.nextInt(5) + 1
          Console.println(id + " : je joue " + c);
          sender ! new Choix(id, c)

        case Fin =>
          Console.println(id + " : je sors de la partie")
          exit()

      ...
```

Le comportement de l'arbitre :

```
class Arbitre(j1 : Joueur, j2 : Joueur) extends Actor {

  def act() {

    j1.start
    j2.start

    j1 ! Jouez
    j2 ! Jouez

    react {
      case Choix(id1, x1) =>

        react {
          case Choix(id2, x2) =>

            if(x1 > x2)
              Console.println("Le joueur " + id1 + " gagne")
            else if(x1 < x2)
              Console.println("Le joueur " + id2 + " gagne")
            else
              Console.println("Egalite")

            j1 ! Fin
            j2 ! Fin
        }
    }
  }
}
```

La fonction principale du programme :

```
object de extends Application {

  val j1 = new Joueur(1);
  val j2 = new Joueur(2);

  val arbitre = new Arbitre(j1, j2)

  arbitre.start
}
```


Philipp Haller and Martin Odersky.
Scala actors : Unifying thread-based and event-based programming.
Theoretical Computer Science, 410(2–3) :202 – 220, 2009.


Rajesh K. Karmani, Amin Shali, and Gul Agha.
Actor frameworks for the jvm platform : A comparative analysis.
In *PPPJ '09 : Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20, New York, NY, USA, 2009. ACM.