Scala et akka
`Tweeter` n'est pas Twitter
avec les acteurs

Jacques Noyé

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

31 janvier 2017

1 Objectif

Découvrir la programmation à base d'acteurs à partir d'un exemple inspiré de Twitter : `Tweeter`.

2 Les bases à partir d'un exemple : un annuaire

La figure 1 montre l'exemple d'une classe `Registry` qui implémente sous la forme d'un *acteur* un annuaire faisant correspondre un nom à une valeur. Le fichier correspondant est disponible sur Campus. Il va tout d'abord s'agir de comprendre cet exemple, qui contient certains des idiomes dont nous allons avoir besoin par la suite, et que nous allons aussi réutiliser tel quel.

Mais tout d'abord qu'est-ce qu'un acteur ? On peut voir un acteur comme un objet *actif* (muni de son propre fil d'exécution) qui va interagir avec d'autres acteurs au travers de messages *asynchrones*. Dans le cas d'un appel de méthode, on parle aussi d'envoi de message, mais ces envois de messages sont *synchrones* : suite à l'envoi d'un message (un appel de méthode) plus rien ne se passe au sein de l'objet source du message, l'exécution continue du côté du destinaire, qui n'a pas d'autre choix que de prendre ce message en compte, avec l'exécution du corps de la méthode. Le retour de la méthode transfère l'exécution de l'objet destinataire à l'objet source tout en fournissant, en général [1], une réponse au message initial.

Un acteur continue par contre son exécution une fois un message envoyé, sans attendre qu'il ait été traité. Du côté de l'acteur qui reçoit le message, un traitement immédiat du message n'est pas requis, le message peut rester dans la *boîte à lettres* de l'acteur en attente d'être traité. Tout se passe comme si les acteurs s'exécutaient simultanément.

1. Ce n'est pas le cas en Java dans le cas d'une méthode `void`.

```scala
package tweeter
import akka.actor.{Actor, ActorRef}

object Registry {
  import akka.actor.{ActorSystem, Props}
  // definition of specific messages received by the actor
  case class Bind(name: String, value: Any)
  case class Lookup(name: String)
  // definition of specific message sent by the actor
  case class LookupAnswer(value: Option[Any])
  // create dedicated actor infrastructure
  val system = ActorSystem("TweeterInfrastructure")
  // factory creates actor with a name and register it with the infrastructure
  def apply(name: String):ActorRef = system.actorOf(Props(new Registry()), name)
}
class Registry extends Actor {
  import Registry._

  private var registry:Map[String, Any] = Map()
  def receive = {
    case Bind(name, value) => registry = registry + (name -> value)
    case Lookup(name) => sender ! LookupAnswer(registry.get(name))
  }
}
object TestRegistry extends App {
  import akka.actor.Inbox
  import scala.concurrent.duration._
  import Registry._

  val r = Registry("registry")
  // create an "inbox" to interact with registry actor
  // (via an internal actor of the inbox)
  val i = Inbox.create(Registry.system)
  // test the registry
  r ! Bind("Alice", "DummyValue")
  // r ! Lookup("Bob") would not properly instantiate "sender" on the registry
  // side as "!" is not called within an actor, use inbox instead
  i.send(r, Lookup("Bob"))
  i.send(r, Lookup("Alice"))
  println(i.receive(FiniteDuration(1, MILLISECONDS)))
  println(i.receive(FiniteDuration(1, MILLISECONDS)))
}
```

FIGURE 1 – La classe `Registry` avec un scénario de test

2.1 Définition d'un acteur

La création d'un acteur s'effectue en deux temps. Il faut tout d'abord créer un objet standard, *passif*, qui hérite du trait `Actor`. Une fois cet objet créé, il est passé au *système d'acteurs* qui va créer un acteur associé à cet objet et l'activer.

Première étape Le trait `Actor` défini dans le paquetage `akka.actor.Actor` (cf lignes 2 et 16 de la figure 1). Ce paquetage n'appartient pas au noyau de Scala et requiert l'ajout d'une archive `akka-actor` et `config` disponibles sur Campus.

Le comportement d'un acteur est défini par la méthode `receive` (cf ligne 20) qui indique comment traiter les messages reçus par l'acteur, ici un des messages suivants :

— `Bind`, ajout d'une nouvelle paire (nom, valeur) à l'annuaire, ou
— `Lookup`, consultation de l'annuaire pour un nom donné.

Deuxième étape Créer des acteurs requiert l'existance d'un système d'acteurs préexistant, ici `Registry.system` (cf ligne 12 et 14). Comme on n'utilise qu'un unique système d'acteurs et un unique annuaire, on peut associer les deux.

La création d'un acteur s'effectue via la méthode `actorOf` de la classe `ActorSystem` à laquelle on passe l'objet sous-jacent (cf ligne 14). Cette méthode crée l'acteur, l'active (le met en capacité de recevoir des messages et de les traiter), et retourne la référence, de type `ActorRef`, qui va permettre d'interagir avec l'acteur en lui envoyant des messages.

Notez qu'il faut bien distinguer l'objet (passif) de type `Registry`, et l'acteur de type `ActorRef`. On interagirait avec le premier via des appels de méthode alors qu'on interagit avec le second via des envois de message. En fait, le pattern utilisé ici rend volontairement l'objet passif inaccessible. Comparez avec :

```
1  def registry = new Registry
2  def apply(name: String):ActorRef = system.actorOf(Props(registry, name)
```

2.2 Définition des messages

Les messages échangés par les acteurs peuvent être de tout type, ils sont toutefois habituellement définis à l'aide de *case classes* (cf ligne 7 et suivantes, l'utilisation du mot-clef `case`). La classe définit le type du message et les paramètres ses données. L'utilisation de *case classes* permet d'utiliser le pattern matching pour gérer les messages reçus (utilisation de son type pour voir dans quel cas on se trouve et récupération des données, cf ligne 21 et suivantes. De plus, l'instanciation d'un message est très simple et ne nécessite pas l'utilisation du mot-clef `new` (cf par exemple ligne 35).

2.3 Envoi des messages

La syntaxe de base et très similaire à l'appel de méthode, sauf que l'opérateur `.` est remplacé par `!` et le nom de la méthode par un nom de message. Comparer :

```
r.bind("Alice", "DummyValue")
```

et

```
r!Bind("Alice", "DummyValue")
```

Dans le premier cas, `r` pourrait être un objet de la classe `Registry`, qu'on aurait munie d'une méthode `bind`.

Dans le deuxième cas, `r` est de type `ActorRef` et `Bind` est un constructeur de message.

Lorsque, suite au traitement d'un message, il y a besoin de renvoyer une réponse, `sender` référence l'acteur responsable de l'envoi initial (cf ligne 22). Au sein d'une instance de la classe

`Actor`, alors que `this` fait référence à l'instance, `self` fait référence à l'instance de `ActorRef` sous-jacente.

Le scénario de test interagit avec l'acteur au travers d'une API spécifique et d'un objet de type `Inbox` (ligne 33 et lignes 38 à 38) qui délègue les envois et réceptions de messages à un acteur sous-jacent si nécessaire (les envois qui n'appellent pas de réponse peuvent utiliser l'opérateur `!` comme le montre la ligne 35). Le paramètre de `receive` indique un temps d'attente pour la réception d'un message, temps au bout duquel une exception est levée. Nous n'aurons pas besoin de faire appel à ce dispositif par la suite.

2.4 Modèle de mémoire

Dans un modèle « pur » les acteurs ne communiquent que par envoi de message et ne partagent pas de mémoire. L'exemple répond à ces contraintes en rendant inaccessible l'instance de la classe `Registry` et en n'échangeant pas de données mutables.

Si on avait accès à l'annuaire à la fois en tant qu'objet et qu'acteur, en supposant de plus accessible la variable d'instance `registry`. Il pourrait être tentant, par exemple pour faire du débogage, d'écrire :

```
rAsActorRef ! Bind("Alice", "F")
println(rAsRegistryRef.registry)
```

Mais le résultat peut être inattendu. En effet, la modification de l'annuaire et sa lecture s'effectue simultanément, pas l'une après l'autre, sans que l'on sache laquelle va se terminer d'abord. La pire situation, est que la lecture s'effectue alors que la modification n'est pas terminée (on parle de modification qui n'est pas *atomique*), ce qui peut conduire à l'impression que l'état de la mémoire est incohérent. Si on remplace la lecture par une écriture, soit deux écritures simultanées, il est même possible de rendre l'état de la mémoire incohérent.

C'est la raison pour laquelle il est conseillé de :

— rendre inaccessible les objets passifs ;

— n'échanger par message que des objets non mutables.

C'est une caractéristique importante des données non mutables, il est beaucoup plus facile de les manipuler en parallèle que de manipuler des données mutables.

3 Première version de `Tweeter`

On va maintenant rajouter de nouveaux acteurs, des *tweeters*, instances de la classe `Tweeter` qui vont communiquer entre eux à la manière de twitter. En particulier, à chaque tweeter est associé une liste de *followers* qui sont d'autres tweeters intéressés par les messages du tweeter.

Dans cette première version, les tweeters répondent indéfiniment aux messages suivants (notez que ce n'est pas le tweeter qui décide des actions à effectuer mais bien l'utilisateur, connu par son nom `user`, du tweeter) :

— `Tweet(content: String)` : demande de relai aux *followers* du tweeter du tweet de contenu `content` ;

— `HandleTweet(user: String, content: String)` : réception par le tweeter d'un tweet dont il est le destinataire et dont la source est l'utilisateur `user` ;

— `Follow(user: String)` : demande de suivre l'utilisateur `user` (et donc de recevoir ses tweets) ;

— `AddFollower(follower: ActorRef)` : demande l'ajout d'un nouveau `follower`.

Les messages `Tweet` et `Follow` sont des messages correspondants aux demandes des utilisateurs alors que les messages `HandleTweet` et `AddFollower` sont des messages échangés entre tweeters.

L'association entre un utilisateur connu par son nom (une chaîne de caractères) et un tweeter (de type `Tweeter`) s'effectue au travers d'un annuaire, ce qui veut dire d'une part que le constructeur d'un tweeter prend deux paramètres, son utilisateur et un annuaire, et que, d'autre part, tout

nouveau tweeter doit préalablement s'enregistrer auprès d'un annuaire préexistant (on suppose ici que cet annuaire est unique).

Implémentez la classe `Tweeter` et testez sur le scénario suivant :

1. Il y deux tweeters de nom Bob et Alice.
2. Alice tweete *I am Alice*.
3. Bob suit les envois d'Alice.
4. Alice tweete à nouveau *I am Alice*.

Afin de suivre la progression de l'exécution, associez une impression à chaque réception de message, par exemple, dans le cas de la réception d'un message `HandleTweet(content)` :

```
println(s"$name receives $content from $sender")
```

où `sender` est une variable qui enregistre l'acteur qui a envoyé le message en cours de traitement.

Notez que le parallélisme entre les acteurs peut jouer des tours. Comme les messages sont envoyés de manière asynchrones et les acteurs progressent en parallèle, il se peut, par exemple, que le deuxième tweet d'Alice se fasse avant que le fait que Bob suive Alice ait été pris en compte. Une manière de donner une meilleure chance à un message d'être pris en compte avant d'en envoyer un nouveau est de faire suivre son envoi d'une instruction `Thread.sleep(`t`)`, qui va endormir le fil d'exécution du programme principal pour environ t millisecondes.

Et que se passe-t-il si Bob indique par erreur deux fois qu'il suit les envois d'Alice ?

4 Deuxième version

4.1 Ajout d'une interface graphique

On veut maintenant disposer d'une petite interface graphique permettant l'affichage des tweets reçus. À chaque utilisateur va donc être associé un premier acteur, issu de la classe `Tweeter`, un deuxième acteur qui va piloter l'interface graphique, issu d'une classe `TweeterView` et une interface graphique, issue d'une classe `TweeterViewGUI`.

Mise en place de l'acteur Une première étape consiste à simplement mettre en place ce nouvel acteur. Du côté, de l'acteur `Tweeter` au lieu d'imprimer un message à la réception d'un tweet, l'idée va être de simplement envoyer à l'acteur `TweeterView` associé un message `TweetView(source: String, content: String)`, où `source` est le nom de la personne ayant envoyé le tweet et `content` son contenu.

La question est comment associer un acteur `Tweeter` et un acteur `TweeterView`. Dans ce cadre limité où la communication s'effectue dans un seul sens, de l'acteur `Tweeter` vers l'acteur `TweeterView`, il suffit de commencer par créer un acteur `TweeterView` et de passer une référence à cet acteur lors de la création de l'acteur `Tweeter`.

Comme l'acteur `TweeterView` ne communique qu'avec `Tweeter`, il sera créé directement au sein du constructeur (`apply`) de `Tweeter`, de sorte que `Tweeter("Alice", r)` où `r` est l'annuaire [2] crée à la fois un acteur `Tweeter` et un acteur `TweeterView` pour Alice.

À ce stade, le test de la première version devrait à nouveau fonctionner sans modification.

Mise en place de l'interface Dans un deuxième temps, l'idée est de passer de l'affichage des tweets reçus dans la console à un affichage dans une interface graphique, implémentée par la classe `TweeterViewGUI`, associée à l'acteur `TweeterView`. Il y a donc une interface graphique par tweeter. Cette interface prend la forme d'une fenêtre (`TweeterViewGUI` hérite donc de `JFrame`), étiquetée par le nom du tweeter (par exemple `Alice`), qui affiche dans un composant de type `JTextArea` les tweets reçus [3].

2. Ce paramètre peut aussi être caché.
3. Un ajout de texte sur une instance de `JTextArea` s'effectue en appelant la méthode `append` avec le texte en paramètre.

Cette interface doit être accédée par l'acteur `TweeterView`. Il est donc nécessaire de passer une instance de `TweeterViewGUI` au constructeur primaire de `TweeterView`. On retrouve le schéma précédent. Cette interface n'étant utilisée que par l'acteur, il est logique de l'instancier lors de la création de l'acteur `TweeterView` (au sein de la méthode `apply`) et ainsi de garder cette modification très localisée.

L'interface graphique doit aussi fournir une API adaptée à l'acteur `TweeterView` et indépendante de ses détails d'implémentation, par exemple une méthode `display(message: String):Unit`.

Notez que tout action relative à l'interface graphique en général doit s'effectuer dans un fil d'exécution spécifique. C'est un point dont on n'a pas forcément à se soucier dans un cadre séquentiel, hors démarrage, qui utilise le pattern suivant :

```
javax.swing.SwingUtilities.invokeLater(new Runnable() {
  def run() {
    createAndShowGUI(parameters)
  }
})
```

Mais ici, l'acteur s'exécute dans son propre fil d'exécution et toute interaction avec l'interface graphique, ici l'appel à la méthode `display`, doit donc être emballé de la même manière.

À ce stade, de même que précédemment, le test initial devrait à nouveau fonctionner sans modification.

4.2 Prise en compte des *retweets*

On veut maintenant donner à un utilisateur la possibilité de ré-émettre vers ses *followers* le dernier tweet reçu. Mettez en place cette fonctionnalité en :

— mémorisant dans l'acteur `TweeterView`, le dernier message `TweetView` reçu ;

— ajoutant à l'interface de chaque tweeter `TweeterViewGUI` un bouton (`JButton`) et en lui associant une action (sous la forme d'un gestionnaire d'évènement [4]) déclenchant un retweet.

Notez que pour effectuer le retweet à partir de l'interface (instance de `TweeterViewGUI`), il faut que l'interface interagisse avec l'acteur sous-jacent (instance de `TweeterView`), qui va lui-même faire appel au tweeter (instance de `Tweeter`) [5]. Or, vu le processus de mise en place du système, l'interface `TweeterViewGUI` ne connait pas son acteur sous-jacent `TweeterView`, qui ne connait pas son acteur `Tweeter`.

Dans un cadre objet habituel ça correspond à la situation où l'on dispose de deux classes `class A(var b: B)` et `class B(var a: A)` qui requièrent qu'une instance `anA` de `A` ait un attribut de type `B` dont l'attribut de type `A` est justement `anA`, ce qui crée une dépendance mutuelle qui ne peut être mise en place directement. Il est en fait nécessaire de créer une instance de `A` incomplète, puis une instance de `B` complète. On peut alors compléter l'instance de `A` en modifiant son attribut (ce qui demande à ce que l'attribut de `A` soit mutable).

On retrouve la même situation entre `TweeterView` et `Tweeter` ainsi qu'entre `TweeterView` et `TweeterViewGUI`.

Pour ce qui est des acteurs, on ne peut pas comme avec des objets directement manipuler des attributs, par contre l'acteur `Tweeter` connait son acteur `TweeterView`, il peut donc lui envoyer un message spécifique dès sa création, par exemple `RegisterTweeter(tweeter: ActorRef)` pour lui communiquer son "adresse".

Finalement, l'acteur `TweeterView` peut, lui, demander à modifier une variable d'instance de `TweeterViewGUI`. Attention, il faut à nouveau passer par le fil d'exécution spécifique aux interfaces graphiques. Et il est intéressant de bien comprendre à quoi va servir cette donnée, à savoir à définir le gestionnaire d'un évènement "clic" sur le bouton `Retweet`, ce qui permet de simplifier la classe `TweeterViewGUI`.

4. Voir l'exemple du compteur.

5. On pourrait aussi envisager dans un deuxième temps de court-circuiter l'étape intermédiaire après en avoir analysé les bénéfices et limitations en terme de performance et d'évolution.

Le scénario initial devrait toujour fonctionner. Mais testez cette dernière version au moins sur un scénario à 3 tweeters, Alice, Bob et Carol avec Carol qui suit les messages de Bob et Bob ceux d'Alice, Alice tweete *I am Alice*, message retweeté par Bob.