Master 1 - Systèmes Distribués
Akka

1 Le modèle d'acteurs

Le **modèle d'acteur** est un paradigme permettant d'organiser des programmes pour introduire le **parallélisme** des traitements et **éviter la gestion de la concurrence** : il n'y a pas d'état partagé mais des acteurs autonomes échangeant des messages. Ce modèle permet de construire des programmes distribués à très grande échelle.

Les **acteurs** sont les entités centrales du modèle. Un acteur peut :
— envoyer des messages ;
— créer des acteurs ;
— décider du comportement à avoir lors de la réception d'un message.

Les acteurs encapsulent leur état interne, et le rendent **inaccessible** aux autres éléments de l'application. Le seul moyen d'apporter une **modification** à cet état est d'**envoyer un message** à l'acteur, pour qu'il se charge d'apporter la modification.

De cette manière, **les accès concurrents à une ressource sont évités** lorsque cette ressources fait partie de l'état interne d'un acteur. Les messages sont ordonnés dans une file et consommés un par un par l'acteur.

2 Utilisation d'Akka

Un programme exemple est disponible sur Github [1], la suite du document sert d'explication complémentaire au code disponible. Commencer par le fichier `App.java` qui contient la méthode `main`.

Bien que disponible dans le langage Java, Akka est développé et plutôt orienté pour une utilisation avec la langage Scala. Le style de programmation est proche du fonctionnel, et s'appuie sur des concepts comme les lambda expressions (`https://www.jmdoudoux.fr/java/dej/chap-lambdas.htm`).

2.1 Importation de la librairie

Le fichier `pom.xml` est utilisé par Maven pour définir et construire le projet :

```
<properties>
    <akka.version>2.6.10</akka.version>
    <scala.binary.version>2.13</scala.binary.version>
</properties>
<dependencies>
    <dependency>
        <groupId>com.typesafe.akka</groupId>
        <artifactId>akka-actor-typed_${scala.binary.version}</artifactId>
        <version>${akka.version}</version>
    </dependency>
</dependencies>
```

2.2 Une classe d'acteur

Une classe d'acteur a besoin de plusieurs éléments :
— elle doit étendre la classe `AbstractActor` ;
— elle doit implémenter la méthode `createReceive()`, qui sert à déterminer le comportement de l'acteur en fonction du message qu'il reçoit ;
— elle peut avoir une méthode `static` pour faciliter la création des acteurs. Cette méthode retourne un objet de type `Props`, correspondant à la technique de création des acteurs.

1. `https://github.com/EricLeclercq/Distributed-Systems-Master/tree/master/Akka`

```java
import akka.actor.AbstractActor;
import akka.actor.Props;

public class HelloWorldActor extends AbstractActor {

    private HelloWorldActor() {}

    @Override
    public Receive createReceive() {
        return receiveBuilder()
                .match(SayHello.class, message -> sayHello(message))
                .match(SayBye.class, message -> sayBye(message))
                .build();
    }

    private void sayHello(final SayHello message) {
        System.out.println("Hello World to " + message.getName());
    }

    private void sayBye(final SayBye message) {
        System.out.println("Bye World");
    }

    public static Props props() { // Pour creer un acteur (ne pas utiliser new)
        return Props.create(HelloWorldActor.class);
    }
    [...]
}
```

La méthode `createReceive()` permet de mettre en correspondance un type de message reçu et l'exécution d'une fonction. Par exemple, la ligne :

```java
.match(SayHello.class, message -> sayHello(message))
```

indique que lorsqu'on recevra un message de type `SayHello`, on aura une variable disponible (`message`, du type du message reçu), et on exécutera la fonction `sayHello(message)`. Plusieurs appels de ce type peuvent être enchaînés, afin de déterminer le comportement de l'acteur pour différents types de message.

Il faut également créer les classes de message que l'acteur peut recevoir. Par convention, ils sont contenus dans la classe de l'acteur qui les utilise, et repose sur le mécanisme des *inner class*.

```java
public class HelloWorldActor extends AbstractActor {
        [...]
        // Definition des messages en inner classes
        public interface Message {}

        public static class SayBye implements Message {
                public SayBye() {}
        }

        public static class SayHello implements Message {
                public final String name;

                public SayHello(String name) {
                    this.name = name;
                }
        }
}
```

2.3 Création d'acteurs

Avant de pouvoir créer un acteur, il faut d'abord initialiser le système d'acteurs, grâce à la méthode `ActorSystem.create()`. On peut alors créer des acteurs dans ce système, grâce à des objets de type `Props`, c'est pourquoi une méthode permettant de créer un objet de ce type est souvent trouvée dans les classes d'acteurs.

```java
import akka.actor.ActorRef;
import akka.actor.ActorSystem;

public class App {
  public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create();
        ActorRef helloActorRef = actorSystem.actorOf(HelloWorldActor.props());

        helloActorRef.tell(new HelloWorldActor.SayHello("Akka"), ActorRef.noSender());
        helloActorRef.tell(new HelloWorldActor.SayBye(), ActorRef.noSender());

        actorSystem.terminate();
  }
}
```

Le système d'acteurs se ferme avec la méthode `terminate()`.

2.4 Les modes de communication

2.4.1 `tell`

Après avoir créé un acteur, il est possible de lui envoyer des messages avec la méthode `tell`, qui prend en paramètre un objet représentant le message, ainsi qu'un acteur désigné comme l'expéditeur. `tell` permet d'envoyer un message sans attendre de réponse ni de confirmation de réception (*fire and forget*). L'opération n'est pas bloquante, et l'expéditeur peut passer à la suite de ses traitements directement.

```java
helloActorRef.tell(new HelloWorldActor.SayHello("Akka"), ActorRef.noSender());
helloActorRef.tell(new HelloWorldActor.SayBye(), ActorRef.noSender());
```

2.4.2 `ask`

`ask` permet d'envoyer un message, et d'attendre une réponse en retour. L'envoi du message n'est pas bloquant en lui-même, mais peut le devenir en faisant appel à la méthode `toCompletableFuture().get()`.

```java
import akka.pattern.Patterns;
import java.time.Duration;
import java.util.concurrent.CompletionStage;
[...]
// Le 2e parametre optionnel permet d'attribuer explicitement un nom a un acteur
joueur1 = actorSystem.actorOf(PingPongActor.props(), "joueur1");
[...]
// new PingPingActor.GetScore() cree un message pour l'acteur de type PingPongActor pour
    demander le score du joueur
CompletionStage<Object> result1 = Patterns.ask(joueur1, new PingPongActor.GetScore(),
    Duration.ofSeconds(10));
int scoreJoueur1 = 0;
try {
        scoreJoueur1 = (int) result1.toCompletableFuture().get();
} catch (Exception e) {
        e.printStackTrace();
}
```

2.4.3 `forward`

`forward` est assez proche du comportement de `tell`, sauf que l'expéditeur n'a pas à être précisé : le même expéditeur que celui du message qui vient d'être reçu est utilisé pour le nouvel envoi (voir figure 1). Ce qui veut dire que l'acteur final peut répondre directement à l'expéditeur initial du message.

```java
joueur1.forward(new PingPongActor.StartGame(joueur2), getContext());
```

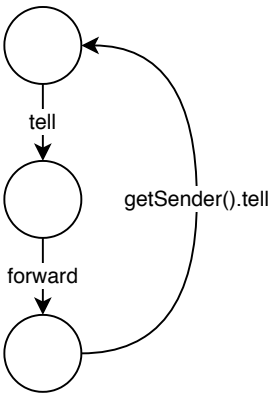

FIGURE 1 – Comportement de la methode `forward`

2.5 Changement de comportement

Au cours de la vie d'un acteur, l'acteur peut modifier dynamiquement le comportement qu'il applique à la réception d'un message. Le comportement utilisé à la création d'un acteur est celui implémenté dans la méthode `createReceive()`. Pour en changer, il suffit d'appeler la méthode avec en paramètre un objet de type `Receive`. Par exemple :

Listing 1 – Extrait du fichier `JourNuitActor.java`

```
1  @Override
2  public Receive createReceive() {
3          return receiveBuilder()
4                          .match(Increment.class, message -> incrementeJour())
5                          .build();
6  }
7
8  public Receive createReceiveNuit() {
9          return receiveBuilder()
10                         .match(Increment.class, message -> incrementeNuit())
11                         .build();
12 }
13 [...]
14 getContext().become(createReceiveNuit());
```

2.6 Le routage

Akka peut aussi être utilisé pour créer un *pool* d'acteurs de la même classe, et de répartir les messages reçus entre eux, afin de paralléliser les traitements. Différentes stratégies de répartition des messages peuvent être utilisées, comme le round robin ou bien l'envoi à l'acteur ayant le moins de messages en attente de traitement.

```
1  import akka.routing.RoundRobinPool;
2  [...]
3  ActorRef router = actorSystem.actorOf(new
       RoundRobinPool(5).props(NombrePremierActor.props()));
4  for (int i = 3; i < 10; i++) {
5          router.tell(new NombrePremierActor.Nombre(i), ActorRef.noSender());
6  }
```

3 Exercices

1. Créer une chaîne d'acteurs de taille paramétrable (chaque acteur crée un fils jusqu'à ce que le nombre voulu soit atteint). Lorsque les acteurs reçoivent un message contenant un String, ils en changent une lettre de manière aléatoire, puis passe le nouveau message à l'acteur fils. L'acteur final doit ensuite renvoyer le message résultat à celui qui a initié la chaîne de messages.

2. Implémenter un acteur responsable d'un stock. Celui-ci doit pouvoir réceptionner des commandes pour rajouter des produits en stock, mais également prendre en charge les demandes des clients. Il ne doit pas fournir la même réponse aux clients si la commande ne peut pas être honorée à cause d'une rupture de stock. Mettre en application le changement de comportement des acteurs.
3. En se basant sur l'exemple de code implémentant une utilisation *remote* d'Akka (voir section Acteurs *remote*), adapter le `PingPongActor` pour utiliser deux machines, chacune hébergeant un acteur. Pour simplifier les interactions, il est possible de ne pas créer d'`ArbitreActor` et d'interagir uniquement avec des `PingPongActor`.
4. Réaliser un bag of tasks en utilisant le modèle d'acteurs. Il doit être possible d'envoyer une tâche au bag of tasks, qui délègue cette tâche à un pool d'acteurs. Il doit être possible de renvoyer le résultat de la tâche à celui qui l'a envoyée.

Maven

Pour créer un projet Maven :

```
mvn archetype:generate
```

Pour compiler un projet (à la racine, au même endroit que le `pom.xml`) :

```
mvn compile
```

Pour exécuter une application Java avec Maven :

```
mvn exec:java -Dexec.mainClass="sd.akka.App"
```

Acteurs *remote*

Dans le fichier `resources/application.conf`, il est possible d'indiquer les paramètres qui seront pris en compte par défaut (y compris le port) lors du démarrage de l'`ActorSystem`. Il est possible d'utiliser ces informations pour contacter les acteurs depuis une autre JVM. Un exemple de l'utilisation de plusieurs `ActorSystem`s répartis sur deux JVM est disponible sur Github [2].

Pour envoyer un message à un acteur distant, il suffit d'instancier un `ActorSelection` en précisant le nom de l'`ActorSystem`, son adresse IP, son port ainsi que le nom de l'acteur à contacter. Il est ensuite possible d'envoyer un message directement à partir de l'objet de type `ActorSelection` récupéré :

```
1 import akka.actor.ActorSelection;
2 [...]
3 ActorSelection selection =
      actorSystem.actorSelection("akka://myActorSystem@127.0.0.1:8000/user/helloActor");
4 selection.tell(new HelloWorldMessage.SayHello("Akka Remote"), ActorRef.noSender());
```

2. https://github.com/EricLeclercq/Distributed-Systems-Master/tree/master/Akka-Remote