Programmation distribué avec Akka.net

Akka.Net est une librairie qui perrmet de construire simplement des applications distribués en utilisant le pattern « Actor Models ». Comme son nom l’indique, c’est un portage de la librairie java Akka dans notre écho système.

<trol> c’est la preuve qu’il reste encore des choses sympa à pomper de java </troll>

# Le pattern Actor Models

La défnition wikipédia est la suivante :

« Le **modèle d'acteur** est un [modèle mathématique](https://fr.wikipedia.org/wiki/Mod%C3%A8le_math%C3%A9matique) qui considère des *acteurs* comme les seules [fonctions primitives](https://fr.wikipedia.org/wiki/Fonction_primitive) nécessaires pour la[programmation concurrente](https://fr.wikipedia.org/wiki/Programmation_concurrente). Les acteurs communiquent par [échange de messages](https://fr.wikipedia.org/wiki/Passage_de_messages). En réponse à un message, un acteur peut effectuer un traitement local, créer d'autres acteurs, ou envoyer d'autres messages »

Le modèle considère donc que tout est acteur (de la même manière qu’en programmation orienté objet, on considère que tout est objet).

Un Acteur doit être en mesure d’effectuer à la réception d’un message :

* Envoyer des messages à d’autres acteurs
* Créer de nouveaux acteurs
* Modifier son état (et donc potentiellement son comportement pour les messages suivants)

Ce que l’on doit retenir ici à mon avis sont les deux notions suivantes :

* Il y a une distinction forte entre le message et le traitement (contenu dans les acteurs)
* Les acteurs traitent les messages les uns après les autres sans se soucier de l’état des autres acteurs (ils sont complètement indépendant dans leurs tâches).

# Akka.net

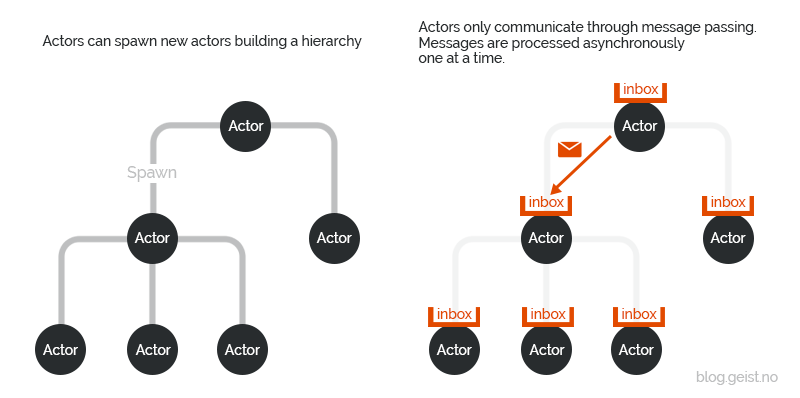
Akka.Net est donc une implémentation de ce pattern pour le framework .net et nous allons ici vous faire une petite démonstration en C#.

Akka.net distingue trois types d’objets :

* Les « Message » : chargé de transporter l’information d’un acteur à l’autre
* Les « Actor »: chargé de traiter les messages, ils sont comstitué d’un état » (state) et d’un « comportement» (behaviour) . Chaque acteur à sa propre fil d’attente qui est dépilé selon le principe FIFO.
* Les « Actor System » :c’est le host qui va gérer une hiérarchie d’acteurs (une sorte d’espace de travail)

A partir de là, nous allons modéliser nos traitements avec quelques règles à respecter :

* Nous allons éviter, par exemple, d’avoir de gros traitements contenus dans un unique acteur mais de le découper en petites fonctionnalités contenues dans un acteur (on retrouve ici un concept proche du principe de responsabilité unique de SOLID) formant ainsi une hiérarchie d’acteur.
* Nos messages devront être immutables pour rendre le modèle plus robuste (on évite de retomber dans les problèmes classique de la gestion concurrentielle des données)
* Acteur est un état et un comportement
* Un acteur doit être codé de manière indépendante des autres acteurs et de leur emplacement (il est potentiellement sur un autre process/machine et ne partage donc pas forcément de la mémoire avec ses petits amis).



Un peu de code

Je vous propose un exemple simple de transformation (un mini ETL) d’une donnée qui passe par deux traitement distinct : une phase d’extraction et une phase de transformation, à l’extraction si les données en entrée (de simple chaines de caractères) sont fausses (la ligne commence par la lettre « a ») on l’envoi au traitement de rejet.

1. **Le message**

ici, tous les messages seront du même type mais rien n’empêche d’en créer plus)

|  |
| --- |
| public class ETLMessage  {  public ETLMessage(string message, int dataId)  {  Message = message;  DataId = dataId;  }  public string Message { get; private set; }  public int DataId { get; private set; }  } |

C’est une classe standard chargé uniquement de porter des messages d’un acteur à l’autre, dans notre exemple, nous n’avons besoins de stocker le contenue de la data et sont identifiant.

1. **La phase d’extraction**

|  |
| --- |
| public class ExtractActor : TypedActor, IHandle<ETLMessage>  {  IActorRef childActorTransform;  IActorRef childActorError;  public ExtractActor()  {  childActorTransform = Context.ActorOf(Props.Create<TransformActor>());  childActorError = Context.ActorOf(Props.Create<ErrorActor>());  }  public void Handle(ETLMessage message)  {  if (message.Message.StartsWith("a"))  {  childActorError.Tell(message);  return;  }  Thread.Sleep(3000);  var newMessage = string.Format("> EXTRACT :#{0} -- {1}", message.DataId, message.Message);  Console.WriteLine(newMessage);  childActorTransform.Tell(message);  }  } |

Nous avons ici deux acteurs enfants :

* childActorTransform  chargé de la phase de transformation de la donnée à effectué si tout se passe bien
* childActorError chargé de gerer les données en erreurs (qui commencent par « a »)

Ces deux acteurs sont « chargés » dans le constructeur en utilisant la methode « Context.ActorOf » et non pas un « new » pour la simple et bonne raisons c’est qu’ici vous allez manipuler une sorte de proxy (ou adresse) sur l’acteur fils. C’est très important car cela permet de déployer les acteurs dans des process/machines différents sans impacter le code (nous vous présenterons comment faire dans un autre article).

Une méthode « Handle » (imposé par l’interface IHandle) chargé de traiter le message. Son contenu est très simple : si le message est en erreur, on le renvoi a l’acteur chargé du traitement des erreurs, sinon on simule un traitement de 3 secondes et on passe le message à la transformation.

1. **Gestion des erreurs et de la transformation**

|  |
| --- |
| public class TransformActor : TypedActor, IHandle<ETLMessage>  {  public void Handle(ETLMessage message)  {  Thread.Sleep(6000);  var newMessage = string.Format(">>>>>> TRANSFORM :#{0} -- {1}", message.DataId, message.Message);  Console.WriteLine(newMessage);  }  }  public class ErrorActor : TypedActor, IHandle<ETLMessage>  {  public void Handle(ETLMessage message)  {  var newMessage = string.Format(">>>>>> IN ERROR :#{0} -- {1}", message.DataId, message.Message);  Console.WriteLine(newMessage);  }  } |

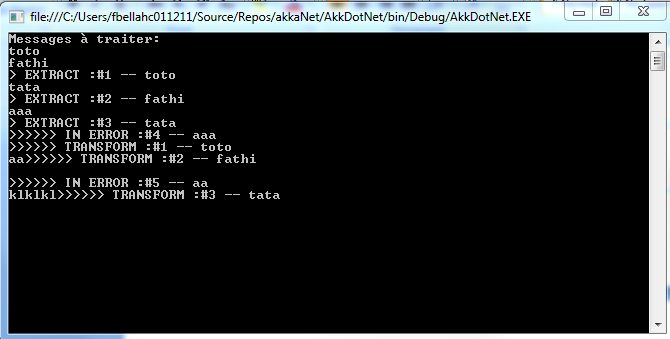
Les acteurs sont très simples, et ne nécessitent pas d’explication. Ce qu’il faut néanmoins pointé c’est qu’a aucun moment nous n’avons produit du code pour la gestions de la fil d’attente des messages (mécanisme inclus dans la classe TypActor) ou encore liée à l’asynchronisme/Parralèlisme des échanges (comme le mot clés async/await ou encore l’utilisation des Task).

1. **Exécution du code**

|  |
| --- |
| static void Main(string[] args)  {  using (var system = ActorSystem.Create("ETL"))  {  IActorRef extractActor = system.ActorOf(Props.Create<ExtractActor>());  var count = 0;  Console.WriteLine("Messages à traiter:");  while (true)  {  count++;  string line = Console.ReadLine();  var message = new ETLMessage(line, count);  extractActor.Tell(message);  }  }  } |

La première chose à faire est de déclarer un espace d’exécution (ActorSystem) pour hoster nos acteurs.

Ensuite on instancie un acteur de type extract, et on lui passe des messages via la console :



Ici J’ai saisi les données de manière aléatoire et rapproché ce que l’on peut constater :

* c’est que l’ordre de traitement des messages est respecté
* la saisie « aaa » ne prend pas de temps mais elle à du attendre que le message #3 soit traité.
* nous avons directement la sortie du traitement de l’erreur sans être bloqué par les autres traitements
* la transformation du message #2 à eu lieu pendant que je saisie un nouveau message : la console n’est jamais bloqué (freeze).

# Conclusion

Nous pouvons donc voir que les files d’attentes et l’asynchronisme sont respecté et que contrairement à un traitement classique (je finis une étape et j’envoi à la suivante) on a un comportement orienté « flux » lorsqu’un message est traité, il passe à l’étape suivante sans attendre la fin de tous les traitements.

En cela, le fonctionnement est très proche de TPL dataflow mais je suis un grand fan de la simplicité et l’élégance de la solution proposé par akka. De plus, la possibilité de déployer les acteurs de manière indépendantes les uns des autres offre des perspectives intéressante en terme de scalabilité et pour l’implémentation d’architecture orienté micro-service.