PROJET SYSTÈME DISTRIBUE

UNIVERSITÉ DE BOURGOGNE MASTER 1

15 JANVIER 2024

KHENTICHE Wissam
BELDILMI Guillaume

Table des matières

Introduction

Nous avons choisi de relever le défi de l'algorithme d'élection de leader de Chang et Roberts dans le cadre de notre projet en Systèmes distribués. Cet algorithme, élaboré pour résoudre le problème de l'élection d'un leader au sein d'un réseau de processus interconnectés, revêt une importance capitale dans le domaine de la coordination et de la gouvernance des systèmes répartis.

Dans ce rapport, nous documenterons en détail notre démarche qui implique l'utilisation du langage de programmation Java pour la réalisation de l'algorithme, tout en explorant la communication distribuée via les modèles d'acteurs avec le framework Akka.

Notre objectif principal est de concevoir une librairie mettant en application l'algorithme de Chang et Roberts qui puisse être intégré de manière transparente dans divers projets distribuées.

Dans la première partie de ce rapport nous décrirons en détail l'algorithme d'élection de leader de Chang et Roberts. Nous examinerons ses étapes, ses mécanismes de fonctionnement, et son importance dans la gestion de la cohérence et de l'ordonnancement au sein de système distribués en donnant des cas d'utilisation.

Ensuite, nous nous plongerons dans la mise en œuvre pratique de cet algorithme en Java. Nous expliquerons comment nous avons modélisé les processus, géré la communication entre eux, et créé une solution qui puisse être réutilisée comme une librairie pour différentes applications distribuée.

La deuxième partie sera dédiée à l'utilisation d'Akka et le principe des acteurs. Nous détaillerons les choix de conception, les avantages et les limitations de cette approche, et nous présenterons les résultats des tests effectués sur notre implémentation.

Enfin, nous conclurons ce rapport en synthétisant les résultats de nos tests, en mettant en évidence les forces de chaque approche, et en offrant des recommandations pour les cas d'utilisation appropriés.

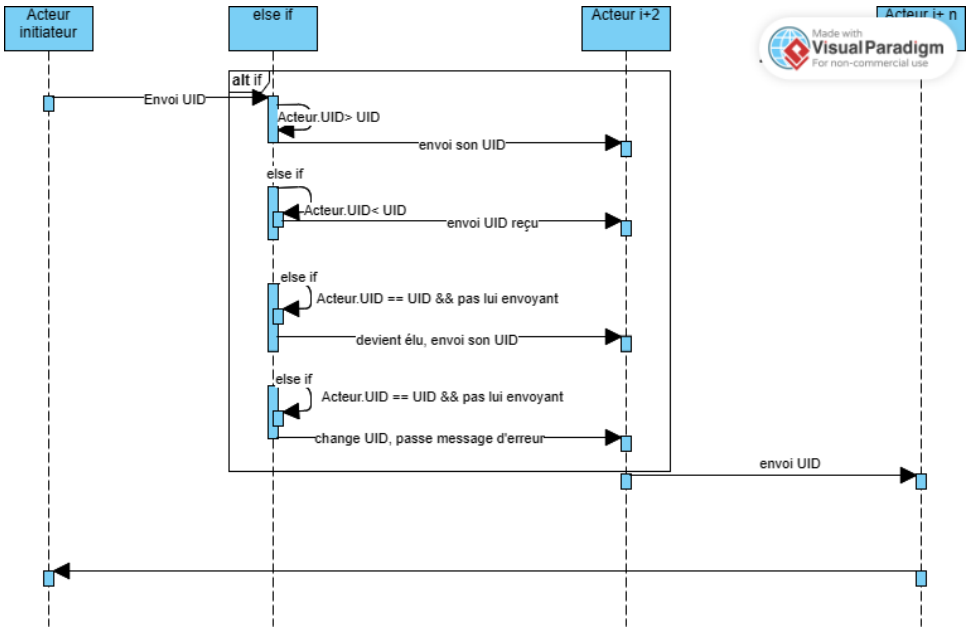
L'Algorithme d'élection de leader Chang et Roberts

En informatique distribuée, l'élection d'un leader est le processus de désignation d'un seul processus en tant que maitre ou référence dans l'organisation d'une tâche répartie entre plusieurs ordinateurs. Avant que la tâche n'ait commencé tous les nœuds du réseau ne savent pas quel nœud sera désigné leader.

Au début l'algorithme suppose que chaque processus dans le système possède une identification unique UID, la communication entre un processus et son voisin est basée sur une structure d'anneau unidirectionnel. L'algorithme présente en deux parties, s'articule comme suit :

- Chaque processus de l'anneau débute en tant que non-participant.
- Un processus initiateur (le dernier initialisé dans notre cas) déclenche une élection en créant un message d'élection contenant son UID et en l'envoyant à son voisin sur l'anneau.
- Chaque processus marqué comme participant lors de l'envoi du premier message d'élection. Dans la phase d'élection, lorsqu'un processus reçoit un message d'élection, il compare son propre UID avec l'UID du message, puis il peut y'avoir différents cas possibles :
  - Si l'UID du message est supérieur au sien, il transfère le message à son voisin, et il devient passif.
  - Si l'UID du message est inférieur au sien, le processus transmet son propre UID à son voisin.
  - Si l'UID du message est égal au sien et que c'est le lui envoyant, il devient élu.
  - Si l'UID du message est égal au sien, mais ce n'est pas l'envoyant, ici c'est un cas d'erreur, dans notre algorithme dans ce cas le nœud change d'UID, puis passe le message d'erreur à son voisin, ça continue jusqu'à ce que tous les nœuds sont informés de l'erreur et aient changés leur UID, puis l'élection se relance comme au premier message.

Une fois que le dernier message avec l'UID le plus grand a fait le tour complet de l'anneau, le nœud l'ayant émis est déclaré leader et l'élection est terminée.



Cas d'utilisation

L'utilisation de l'algorithme de Chang et Roberts pour l'élection de leader est pertinent dans les systèmes distribués lorsqu'il s'agit de gérer la cohérence de données et l'exécution de code sur divers nœuds. Des cas d'utilisation concrets peuvent être :

Les systèmes de gestion de base de données distribuées

Par exemple dans une base de données distribuée, un leader élu peut superviser les opérations de lecture et d'écriture, et ainsi faciliter la cohérence des données et la gestion efficace et ordonnée des transactions, en évitant par conséquent les conflits qui pourraient survenir lors d'accès concurrentiel aux données.

Les environnements cloud pour le calcul distribué

L'élection du leader simplifie la coordination en centralisant l'allocation des ressources, la répartition équilibrée des tâches entre les nœuds, et la gestion efficace des résultats. Par exemple lors de l'exécution d'analyse de données distribuées, un leader optimise l'utilisation des ressources, facilite la répartition des charges de travail, et garantit une collecte cohérente des résultats, assurant ainsi une exécution efficace des calculs dans le cloud.

Les systèmes de files d'attente réparties

Dans ce système, l'élection d'un leader offre la supervision centralisée de la gestion des messages. Par exemple, dans une plateforme de messagerie distribuée, un leader peut garantir un traitement ordonné et efficace des demandes, il coordonne la réception, l'ordonnancement et la distribution des messages entre les différents nœuds, ainsi une gestion cohérente des files d'attente et une réactivité optimale face aux requêtes.

Les applications de commerce électronique distribuées

Par exemple lors de transactions électroniques distribuées, l'élection d'un leader permet e faciliter la gestion des paiements, la supervision des niveaux de stock, ainsi que la synchronisation des données entre les différents serveurs impliqués dans le processus.

Les réseaux de télécommunications distribués

Dans un réseau de télécommunications distribué, un leader peut être élu pour gérer la distribution des ressources réseau, optimiser le routage des appels et garantir la qualité de service.

Analyse du sujet

Nous avons choisi d'utiliser le paradigme des acteurs en particulier le framework Akka pour la mise en œuvre de cet algorithme. Ce choix est dû à la nature distribuée du problème. Les acteurs fournissent un modèle léger et concurrent, permettant une gestion efficace de la communication et de la coordination entre les nœuds du système distribué. Ce qui implique la modélisation des acteurs, la définition des messages et la mise en œuvre des étapes spécifiques de l'algorithme.

Structure de données

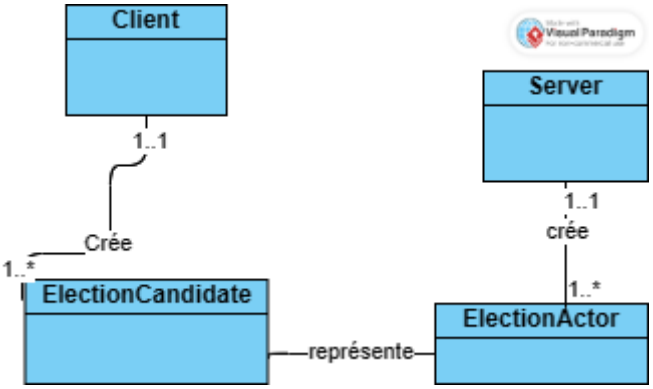
Acteurs Akka : ils représentent des entités autonomes qui peuvent traiter des messages et interagir avec d'autres acteurs. Chaque acteur a son propre état interne (actif ou passif). Un acteur est un candidat

Messages : les messages échangés entre les acteurs peuvent être des objets contenant les informations nécessaires à l'élection, comme l'UID du candidat.

Architecture

Partie librairie

La structure de notre librairie se présente ainsi :



Avoir choisi ce lien entre ElectionCandidate et ElectionActor nous permet de séparer les comportements internes à notre module de librairie tout en permettant son extension par le programme utilisateur via une implémentation de l'interface ElectionCandidate. En effet, cette dernière définit uniquement deux méthodes qui devront être appelées, une fois l'élection de leader terminée, par les ElectionActors auxquels elle sera rattachée (à la façon d'un évènement).

Idéalement, l'exécution des ElectionActors se ferait uniquement sur le nœud "central" puis, une fois les ElectionCandidate élus ou non, leur exécution reprendrait sur leur nœud client d'origine, répartissant au mieux le temps d'exécution entre les différentes machines.

Server

La classe Server sert à initialiser et rendre disponible l'anneau d'acteurs ElectionActor aux clients. Pour cela, il lui est indiqué le nombre de clients ainsi que le nombre d'acteurs à attribuer à chaque client (par exemple, si chaque nœud est amené à gérer plus d'un acteur). Utilisant un ActorSystem, cette classe dispose donc d'une méthode permettant d'achever l'exécution de ce dernier.

Client

La classe Client permet, quant à elle, de se connecter à une instance de Server, de récupérer la liste des acteurs alloués au client afin d'y passer les références aux objets implémentant ElectionCandidate. Cette classe permet aussi de lancer l'élection en envoyant un message ElectionMessage (avec l'UID -1 pour être sûr que l'acteur ElectionActor réceptionnant le message transmette son propre UID à son voisin) au premier acteur lui étant alloué. Utilisant, elle aussi, un ActorSystem, cette classe dispose donc d'une méthode permettant d'achever l'exécution de ce dernier.

ElectionCandidate

L'interface ElectionCandidate décrit juste un objet (idéalement Remote) ayant au moins les méthodes elected() et unelected() respectivement appelées par leur ElectionActor après la fin d'une élection dans le cas où le candidat est élu leader ou non.

ElectionActor

La classe ElectionActor est la plus complexe du module, il s'agit d'une machine à état pouvant passer d'un état à l'autre par la réception de certains messages.

L'objet ElectionActor peut se trouver dans l'un des quatre états suivants :

- Waiting : dans cet état, l'acteur n'est pas prêt pour l'élection. L'une de ses références nextActorRef ou candidate n'est pas renseignée et attend par conséquent un message ActorRefMessage et/ou un message CandidateMessage afin de renseigner l'une de ses références.
- Candidate : cet état reprend les spécifications définies par Chang et Roberts en ajoutant la détection d'erreur et l'émission d'un ErrorMessage dans ce cas-là.
- NotCandidate : dans cet état, l'acteur est passif dans l'élection et transfert donc tous les messages ElectionMessage à son voisin mais reste attentif aux messages ElectedMessage et ErrorMessage permettant respesctivement de passer à l'état ElectionEnded et réinitialiser et reprendre l'élection depuis le début.
- ElectionEnded : dans cet état, l'élection de leader est terminée, la méthode de l'objet implémentant ElectionCandidate a été appelée en fonction du résultat de l'élection pour le noeud courant. L'acteur ne réagira plus à aucun message lui étant adressé.

Partie exemple

Cette partie reprend les modules client et server qui servent à proposer des exemples d'implémentations afin de mettre en application notre librairie.

Package client

Foo

Cette classe sert juste à implémenter l'interface ElectionCandidate avec deux méthodes elected() et unelected() bateaux affichant l'état de fin du candidat accompagné de son ID.

App

Cette classe statique sert à accueillir la méthode main() initialisant la liste des ElectionCandidate via la classe Foo puis initialisant un client de notre librairie afin de lancer une nouvelle élection.

Cette classe est accompagnée des méthodes askId() et scan() afin de demander à l'utilisateur l'identifiant que le client doit utiliser pour se connecter au serveur.

Package server

App

Cette classe statique héberge juste la méthode main() exécutée par le nœud serveur. Cette dernière crée juste une nouvelle instance de Server de notre librairie avec un nombre de client et de candidats par client prédéfini dans deux constantes.

Objectifs souhaités et résultats obtenus

Les objectifs souhaités dans ce projet étaient :

- D'implémenter l'algorithme de Chang Roberts en Java avec les acteurs Akka ;
- De proposer une détection d'erreur au cas où deux ElectionActor aient obtenu le même UID par hasard ;
- De faire l'exécution de l'élection sur le coordinateur uniquement puis de reprendre l'exécution d'un système défini par le développeur utilisant la librairie sur les nœuds clients uniquement.

Concernant, l'implémentation de l'algorithme de Chang Roberts, elle a été réalisée sans difficulté particulière, d'abord en local sur un seul nœud puis en programmation distribuée déporté vers un nœud coordinateur externe.

La détection d'erreur a été relativement simple à ajouter à l'algorithme de base. Le choix a été fait de réinitialiser entièrement le système d'élection après avoir détecté une erreur afin de ne pas provoquer d'erreur en conservant certains états internes d'acteurs.

Le renvoi de l'exécution sur les différents clients n'est malheureusement pas encore fonctionnel avec l'utilisation d'objets distribués envoyés au serveur.

Difficultés rencontrées et pistes d'améliorations

Tout d'abord, le projet a été créé en utilisant un seul nœud avec un seul ActorSystem afin de tester l'implémentation avant de le transformer en un programme distribué. Jusque-là aucune difficulté majeure n'avait été rencontrée.

Lors du passage du programme en distribué, notre première idée avait été de ne pas utiliser de noeud "central" pour exécuter les élections mais de construire un anneau en peer-to-peer. Cependant, ce type d'implémentation nous a posé plusieurs problèmes notamment dû à la classe ConfigFactory permettant d'initialiser l'instance d'ActorSystem offrant peut de configuration à l'exécution, et la création de liens en anneau à travers les différents ActorSystems où l'on doit s'assurer que chaque nœud est correctement initialisé et disponible à l'exécution de l'instruction de connexion d'un nœud voisin.

Cette première tentative aura pris une grande quantité de temps et de réflexion avant d'avoir été finalement avortée.

Par la suite, nous avons finalement choisi de passer par un nœud central pour déléguer l'exécution de l'algorithme d'élection via les acteurs, et ainsi utiliser une architecture client-serveur plus classique et simple à implémenter.

Une fois cette nouvelle implémentation réalisée, nous nous sommes retrouvés face à la difficulté de renvoyer l'exécution du code sur les divers clients (afin de poursuivre l'exécution d'un éventuel programme distribué une fois l'élection terminée). Pour régler ce problème, nous avons essayé de passer un objet Remote issu de Java RMI à notre acteur afin qu'il puisse lancer l'exécution à distance du code correspondant à l'état de sorti de l'acteur ElectionActor après l'élection sur les divers clients.

Cependant, cette implémentation produit toujours des exceptions à l'exécution, nous avons donc choisis d'empêcher temporairement l'exécution de cette partie en commentant les appels correspondant dans la classe ElectionActor.

Après plus d'approfondissement, cette fonctionnalité aurait peut-être été réalisable en remplaçant les objets ElectionCandidate par des références d'acteur ActorRef pointant sur un acteur hébergé sur le nœud client.

De cette façon, on aurait pu déclencher l'exécution de la suite du code du client via l'envoi d'un message à ce dernier, correspondant à l'état de sortie de l'élection.

Compilation et exécution du code

Pour compiler notre code source il suffit d'exécuter le script "build.sh" qui est présent à la racine du projet, ensuite, les scripts "run-server.sh" et "run-client.sh", exécutés via des terminaux différents, permettent respectivement de lancer l'exécution du programme serveur et d'une instance de client.

Il est recommandé de lancer l'exécution du serveur avant le client.

Jeu de test

Afin de voir les résultats de l'élection, nous avons fait des tests afin de visualiser à quoi ressemble le résultat :

On commence par la compilation, du code comme ici :

```
wk060453@MI104-15:~/projet-sd$ ./build.sh
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/com/typesafe/akka
/akka-bom_2.13/2.6.20/akka-bom_2.13-2.6.20.pom
Downloaded from central: https://repo.maven.apache.org/maven2/com/typesafe/akka/
akka-bom_2.13/2.6.20/akka-bom_2.13-2.6.20.pom (7.7 kB at 15 kB/s)
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] lead                                                               [pom]
[INFO] lib                                                                [jar]
[INFO] client                                                             [jar]
[INFO] server                                                             [jar]
[INFO]
[INFO] -------------------------< com.gbeldilmi:lead >-------------------------
[INFO] Building lead 0.0.1                                              [1/4]
[INFO] --------------------------------[ pom ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ lead ---
[INFO]
```

Ensuite, on lance le serveur :

```
wk060453@MI104-15:~/projet-sd$ ./run-server.sh
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.gbeldilmi:server >------------------------
[INFO] Building server 0.0.1
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ server ---
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plex
us/plexus-utils/3.4.2/plexus-utils-3.4.2.pom
```

Et à la fin on lance le client dans un autre terminal, il va nous demander de saisir l'ID du client, comme nous n'avons qu'un client on tape '0' :

```
wk060453@MI104-15:~/projet-sd$ ./run-client.sh
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< com.gbeldilmi:client >------------------------
[INFO] Building client 0.0.1
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ client ---
Enter this client ID:0
```

Sachant que nous avons choisis un nombre de candidat par client égal à 10, le résultat de l'élection est affiché ainsi :

```
[INFO] [01/15/2024 13:40:35.133] [com.gbeldilmi.lead_example.server.App.main()] [Art
eryTransport(akka://server)] Remoting started with transport [Artery tcp]; listening
 on address [akka://server@127.0.0.1:8000] with UID [-7337205733341649575]
Elected ---- UID : 980687
Unelected -- UID : 856770
Unelected -- UID : 845181
Unelected -- UID : 642538
Unelected -- UID : 65142
Unelected -- UID : 908656
Unelected -- UID : 315169
Unelected -- UID : 202342
Unelected -- UID : 687513
Unelected -- UID : 376055
[INFO] [akkaDeadLetter][01/15/2024 13:41:03.278] [server-akka.actor.default-dispatch
```

Conclusion

En conclusion, ce projet représente une réussite globale, démontrant avec succès l'implémentation fonctionnelle de l'algorithme de l'élection de Chang et Robert fonctionne et en distribué. Cela a constitué une mise en pratique concrète des concepts abordés dans le module de Systèmes Distribués, enrichissant notre compréhension pratique des mécanismes liés à la coordination dans des environnements distribués complexes.

Cette expérience nous a non seulement validé l'efficacité de l'algorithme d'élection, mais elle a également offert une occasion exceptionnelle d'explorer les subtilités de la programmation distribuée en utilisant le paradigme d'acteurs avec Akka. La création d'une librairie modulaire a apporté une dimension supplémentaire, transformant notre réalisation en un outil polyvalent qui peut être facilement intégré dans divers projets distribués.

Malgré des défis rencontrés, notamment lors de la distribution du programme et du renvoi de l'exécution dur les clients, nous avons pu les surmonter en identifiant des pistes d'améliorations.

En résumé, ce projet représente une avancée significative dans la mise en pratique des concepts de SD, offrant une solution fonctionnelle et ouvrant la voie à des améliorations futures pour une utilisation encore plus étendue. C'est une étape importante dans notre parcourt académique et professionnel, nous préparant à relever de nouveaux défis dans le vaste domaine des SD.

GitHub du projet : https://github.com/gbeldilmi/projet-sd