École nationale supérieure d'électrotechnique, d'électronique, d'informatique,
d'hydraulique et des télécommunications
Filière Informatique et Mathématiques appliquées

Rapport du projet de systèmes concurrents / intergiciels

Olivier Lienhard
Tom Lucas
Thibault Hilaire

Toulouse, le 13 Janvier 2015

Sommaire

Partie 1

Introduction

Ce projet s'attarde sur l'implémentation d'un modèle de coordination et communication nommé Linda. Il est constitué par un espace de stockage d'objets appelés Tuple. Un Tuple est un n-uplet ordonné de valeurs comme ["voiture", 1]. Son template peut etre [String Integer] ou encore [Object Integer]. Une fois un tuple écrit il peut être pris (take) ou lu (read) par des processus. Notre implémentation de Linda doit pouvoir gérer les demandes de plusieurs threads à la fois selon une spécification décrite précédemment à l'implémentation. Les take consomment les tuples de l'espace, contrairement aux read. L'avantage d'un tel modèle est que les différents processus n'ont pas conscience des autres, ils communiquent vers l'espace partagé qui s'occupe de gérer les problèmes de concurrences.

Partie 2

Choix de la spécification libérale

Explications des différents choix de la spécification libérale :

- quand plusieurs tuples correspondent, take retourne le premier à avoir été écrit dans la mémoire (FIFO)
- quand plusieurs take sont en attente et qu'un dépôt peut en débloquer plusieurs, on débloque le premier take à avoir demandé (FIFO)
- quand des read et un take sont en attente, et qu'un dépôt peut les débloquer, on les débloque dans l'ordre de demande (FIFO)
- quand il y a un take et un callback enregistré pour le même motif, le take est prioritaire

Partie 3

Version en mémoire partagée

L'implantation mémoire partagée se situe dans le package linda.shm, et les tests associés dans le package linda.test.

3.1 Choix d'implémentation

Voici les différentes structures choisies pour respecter nos choix de spécification:

- Collection<Tuple> tuples : pour sauvegarder nos tuples qui ont été écrits dans la mémoire, et les enlever lors d'un take
- Map<Tuple, LinkedList<Integer» MatchEnAttente : cette map va permettre de stocker tous les take/read bloquants
- int id : un entier qui nous permettra de simuler notre FIFO, de savoir dans quels ordres les take/read ont été écrits dans notre map
- ArrayList<Condition> classe : un ensemble de conditions liées à un Lock, dont on associera chaque condition à un take/read bloquants
- Condition writeCondition : une condition qui va nous permettre de faire une priorité au signalé lors du réveil d'un take/read
- Boolean takeEffectue : un booléen permettant de vérifier, lors d'un write, si un take a été effectué

A partir de ceci, on peut expliquer l'algorithme de la fonction write :

Tout d'abord on ajoute le tuple dans la collection de tuples. Ensuite on récupère les templates dont le tuple ajouté correspond (i.e. tuple.matches(template)==true), ce qu'on appelle les templatesCorrespondants. Puis tant que cette collection de templatesCorrespondants n'est pas vide ET qu'un take (sur ce tuple ajouté)

n'a pas été effectué (vérifiable sur notre booléen takeEffectue), on réveille le premier template correspondant (celui dont l'indice id est le plus bas). Si un take est effectué, le booléen takeEffectue devient vrai, on sort de la boucle et on ne réveille pas les callback et la fonction write est finie. Sinon, si aucun take n'est effectué et si on a parcouru toute la liste des templatesCorrespondants, on notifie les callback qui sont des observateurs. Un tel algorithme garantit alors le respect de la spécification ci-dessus, qui est ensuite validée par les tests décrits ci-dessous.

3.2 Tests

Pour les tests, outre les tests donnés, des tests unitaires sont effectués pour chaque fonction pour chaque type de paramètre différent (notamment pour les callback). Ainsi une liste non exhaustive des tests unitaires principaux est :

- BasicTestAsyncCallbackReadFuture.java
- BasicTestAsyncCallbackReadImmediate.java
- BasicTestAsyncCallbackTakeFuture.java
- BasicTestAsyncCallbackTakeImmediate.java
- BasicTestCallbackReadFuture.java
- BasicTestCallbackReadImmediate.java
- BasicTestCallbackTakeFuture.java
- BasicTestCallbackTakeImmediate.java
- BasicTestRead.java

D'autres tests unitaires pour les fonctions tryRead, tryTake, TakeAll, ReadAll sont aussi définis. Des tests vérifiant quel les callback peuvent se réenregistrer sont aussi effectués.

Puis, pour vérifier que la spécification est bien respectée, nous avons effectué des tests pour vérifier chacun des quatre points, nommés respectivement dans l'ordre :

- BasicTestTakeSpec1.java
- BasicTestTakeSpec2.java
- BasicTestTakeReadSpec3.java
- BasicTestTakeCallbackSpec4.java

Ces tests-là pour les principaux, ajoutés à d'autres tests, donnant tous des résultats cohérents, nous permettent de valider notre implémentation.

Partie 4

Version client / mono-serveur

4.1 Implémentation

L'implantation mono-serveur se situe dans le package linda.mono.server, et les tests associés dans le package linda.testMonoServ.

4.1.1 Serveur

L'interface LindaServer hérite de java.rmi.remote , elle reprend les même fonctions que LindaCentralized, seule la signature de la méthode eventRegister est différente (voir section 3.1.3).

La classe LindaServerImpl implémente cette interface et hérite de UnicastRemoteObject .Elle possède un attribut de type CentralizedLinda qui est réutilisé sans avoir été modifié. Une fonction main() permet de lancer le serveur et de l'enregistrer ..... pour pouvoir être trouvé par les clients.

```
private CentralizedLinda linda;

public static void main(String[] args) {
        System.out.println("Demarrage du serveur");
        String URL = "//localhost:4000/LindaServer";
        int port = 4000;
        try{
                LindaServer serveur = new LindaServerImpl();
                LocateRegistry.createRegistry(port);
                Naming.rebind(URL, serveur);
        }
        catch(Exception e){
                System.out.println("Une erreur s'est produite");
                e.printStackTrace();
        }
}
```

A par pour les eventRegister (voir section 3.1.3), les autres fonctions du serveur consistent simplement en un appel aux fonctions équivalente du la version centralisé Linda du serveur:

```
@Override
public Tuple tryTake(Tuple template) throws RemoteException {
        return linda.tryTake(template);
}
```

4.1.2 Client

La classe LindaClient implémente l'interface Linda. Son constructeur prend en paramètre l'URL du serveur.

```
serveur = (LindaServer) Naming.lookup(URI);
```

Les fonctions de cette classe (à part EventRegister) consiste ainsi en l'appel des fonctions correspondantes du serveur.

4.1.3 Le cas des EventRegister

La principale difficulté de la version mono-serveur de Linda est l'implémentation de la méthode eventRegister car les callbacks pris en paramètre par la fonction eventRegister de la classe CentralizedLinda ne fonctionnent pas "à distance". La solution à été de créer un callback spécial pour pouvoir faire cela, l'interface RemoteCallback hérite de java.rmi.remote et est implémentée par la classe RemoteCallbackImpl : son constructeur prend en paramètre un callback classique et sa méthode Call appelle la méthode callback de ce dernier.

Le méthode EventRegister du serveur prend en paramètre un remoteCallback et crée un callback classique dont la fonction Call appelle celle du remoteCalback qu'elle passe en paramètre à la méthode EventRegister de CentralizedLinda :

```
@Override
public void eventRegister(eventMode mode, eventTiming timing,
        Tuple template, final RemoteCallback callback)
        throws RemoteException {

        final class newCallback implements Callback {
                public void call(Tuple t) {
                        try {
                                callback.call(t);
                        } catch (RemoteException e) {
                                e.printStackTrace();
                        }
                }
        }
        linda.eventRegister(mode, timing, template, new newCallback() );
}
```

La méthode EventRegister du client (qui implémente l'interface Linda) prend en paramètre un callback classique et crée un remoteCallback à partir de celui ci pour pouvoir appelé la méthode EventRegister du serveur:

```
@Override
public void eventRegister(eventMode mode, eventTiming timing,
                                        Tuple template, Callback callback) {
        try {
                RemoteCallback cb = new RemoteCallbackImpl(callback);
                serveur.eventRegister(mode, timing, template, cb);
        } catch (RemoteException e) {
                System.out.println("Erreur lors du eventRegister");
                e.printStackTrace();
        }
}
```

4.2 Tests

Tout les test effectués sur la version centralisé ont été effectués également sur la version mono-serveur, en donnant les même résultats.
De plus cette version à été testé avec des "whiteboard" et ils fonctionnent parfaitement.

Partie 5

Version client / multi-serveur

Cette version n'a pas pu être implantée correctement, cela est dû à quelques difficultés et un léger manque de temps de notre part. Cependant, quelques fichiers d'implantation sont présents dans le package linda.server, bien qu'ils n'aient pas aboutis.

Partie 6

Conclusion

Ce projet a abordé des concepts relativement nouveaux. En particulier les eventRegister où il a fallu s'approprié les sources fournies pour implementer ce système 'd'abonnement'. Une fois la spécification libérale définie, l'implémentation des primitives reposait sur des connaissances acquises. Il a fallu néanmoins prendre en compte certaines subtilités vu en cours comme la priorité au signaleur/signalé. Pour la réalisation de la version en mémoire partagée ou mono-serveur , les eventRegister ont demandé plus de temps puisqu'il fallait les prendres en compte dans le code précédemment ecrit. De plus, de nombreux tests ont du être fait pour s'assurer de leur bon fonctionnement ainsi que de la cohérence sur les priorités définies. Nous avons passé du temps sur la version multi-serveurs mais sans réussite. Nous étions partis sur un système d'eventRegister où un serveur qui reçoit une requète s'abonne sur les différents serveurs. La difficulté majeure de cette version est de gérer l'arrêt de ces abonnements lorsqu'un est trouvé. C'est sur ce point que nous avons pas trouvé de solutions. Notre groupe a cependant compris la globalité du sujet. Nous avons avancé ensemble et tenté des améliorations (implantation multi-activité, nous pourrons aussi en discuter lors de l'oral bien que nous en ayons enlevé toute trace dans le code final). Nous avons le sentiment de maîtriser les parties du sujet traîtés et sommes d'accord sur les choix qui ont du être faits.