Projet Systèmes Concurrents :

Création d'un espace mémoire de tuples

Damien Kleiber
Philippe Leleux
Arthur Manoha

13 janvier 2013

Résumé :

Ce projet consiste à implanter un espace mémoire de type **Linda** : une mémoire **gérant des tuples**. Nous devons coder quelques fonctions élémentaires de cela : **write**, **read**, **take**, **tryTake**, **tryRead**, **takeAll**, **readAll** et enfin **eventRegister** dont seules read et take sont bloquantes. Trois implémentations nous sont demandées :

- la première est une implantation de l'interface Linda en **mémoire partagée**.
- la deuxième impose que l'interface Linda accède à un **serveur distant** centralisant la mémoire des tuples.
- la dernière contiendra **plusieurs serveurs interconnectés** les uns aux autres.

Table des matières

I Présentation générale

I.1. Introduction

Définition d'un espace mémoire Linda :

Un espace mémoire Linda est une base de donnée qui s'occupe de gérer l 'enregistrement et le renvoi de tuple à des clients. Un tuple est un n-uplet composé d'éléments de type quelconque (ex ['a' 21 ['z' 51] « voiture »]) qui peuvent même être, dans le cas des templates, des types eux-même, correspondant alors à n'importe quel élément de ce type (ex [?Integer …]).

Un espace de tuples permet d'échanger des données de type quelconque, avec d'autres utilisateurs en local, sur un serveur ou à travers un réseau de plusieurs serveurs. L'un des atouts majeurs de cette implantation est l'aspect bloquant des fonction **read** et **take** mais nous devons avouer que la réelle difficulté de ce projet réside en l'implantation des **EventRegister**.

I.2 Distribution fournie

I.2.1 Package Linda

- Tupla.java : implémente les opérations nécessaires sur les tuples comme le fait que deux tuples correspondent.
- TupleFormatException : code d'une Exception signalant le fait que le format d'un tuple est incorrect. Cela signifie par exemple que l'on a pas mis d'espace avant le premier élément ou encore qu'on les a séparé par des virgules (arrivé très souvent au début du projet).
- Linda.java : Interface des différentes implantation de la base données. C'est dans cette interface que les entêtes des fonctions à coder sont fournies.
- Callback : interface implantant les callback, éléments essentiels pour les eventRegister.
-AsynchronousCallback : implantation de l'interface Callback qui crée un thread faisant appel à un autre callback et renvoie false du premier coup (c'est à dire se désabonne).

I.2.2 Package test

Ce package fournit des classes de tests élémentaires qui doivent théoriquement fonctionner.

I.3 Spécifications

I.3.1 trois implantations différentes

Nous devons rendre 3 versions de la base données Linda.

La première, la plus simple, consiste à partager un espace mémoire local, tout va alors s'exécuter directement dans la même machine virtuelle que les codes utilisateurs. Les classes Linda,

Tuple, Callback et AsynchronousCallback ne doivent pas être modifiées et la base donnée doit être implantée dans une classe CentralizedLinda avec un constructeur sans paramètre.

La deuxième version est une implantation telle que les utilisateurs accèdent à un serveur distant qui possède l'espace des tuples. La classe qui implante la base de données est une classe LindaClient qui doit être une implantation de Linda et dont le constructeur doit prendre un unique paramètre String qui est l'adresse du serveur Linda à utiliser.

La troisième version est une version multi-serveurs : un client est connecté à un unique serveur et les serveurs sont interconnectés entre eux. Cette version utilise la même interface LindaClient qu'en mono-serveur mais l'implantation change. Un client qui écrit un tuple dans la base donnée le fait dans le serveur auquel il est connecté ; la lecture se fait aussi dans ce serveur mais si elle échoue, le serveur se charge de propager la demande aux autres serveurs

I.3.2 Méthodes élémentaires

write : cette méthode ajoute un tuple dans l'espace mémoire, qui peut contenir plusieurs fois le même tuple.

read : cette méthode cherche un tuple dans l'espace et le renvoie, sans l'enlever.

take : même méthode que read sauf qu'elle enlève le tuple trouvé avant de la renvoyer.

Les deux dernières méthodes sont bloquantes, c'est-à-dire que si le tuple demandé n'est pas présent le client se met en attente jusqu'à ce qu'un nouveau tuple correspondant soit ajouté.

tryTake et **tryRead :** versions non bloquante des méthodes read et take, si le tuple n'est pas trouvé elles renvoient null.

takeAll et readAll : ces méthodes cherchent toutes les instances d'un tuple pour les lire ou les retirer, et si aucune instance du tuple n'est trouvée elles renvoient null.

eventRegister : cette méthode permet de s'abonner à l'occurrence d'un tuple. Elle appelle un Callback qui va se charger de retirer le tuple de l'espace mémoire et ce jusqu'à ce qu'elle renvoie false. L'abonnement peut être effectif immédiatement si les tuples nécessaires sont déjà présents.

I.3.3 Choix libres

Le sujet laisse libre la priorité d'une opération par rapport à une autre, c'est-à-dire que lorsqu'un write est effectué et que par exemple deux opérations ou abonnements sur un même tuple sont en attente , le choix de la priorité est libre. Nous avons donc décidé de donner la priorité aux abonnés (trouvant normal qu'un abonné reçoive son tuple avant les autres) puis au read et enfin au take, pour débloquer un maximum les clients. Le read est prioritaire sur le take car il laisse le tuple disponible et ainsi les take peuvent toujours se débloquer par la suite. Cependant, aucun read n'est prioritaire par rapport aux autres read et cela est valable pour les take et les abonnements.

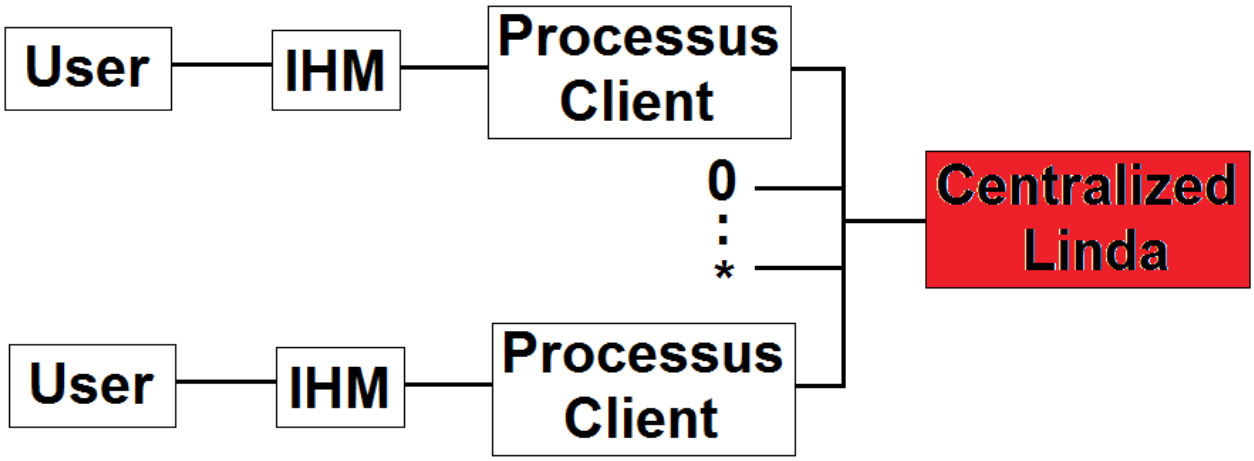
Nous avons également décidé, pour faciliter l'utilisation, de créer une interface graphique utilisateur à laquelle accède l'utilisateur et qui renvoie ainsi les commandes directement à la classe client elle-même.

Nous avons choisi d'implanter readAll et takeAll.

II Linda en mémoire partagée

II.1 Architecture

L'architecture de cette version de Linda est simple et est représentée par le schéma ci-dessous :



L'élément User correspond à l'utilisateur lui-même, l'humain devant sa machine ; il a pour vision du système l'interface graphique IHM. Cette interface correspond à la partie vue de l'application et permet d'entrer un tuple et de demander à faire appel aux différentes méthodes de CentralizedLinda. L'interface graphique transmet la demande de l'utilisateur au client lui-même en changeant son état (ProcessusClient) qui lui fait la requête au CentralizedLinda.

Ces éléments correspondent à ce que possède un utilisateur. Plusieurs client peuvent être connectés à CentralizedLinda en même temps et ils partagent donc cet espace mémoire local, qui est situé dans la même mémoire virtuelle que leur propre code.

II.2 EventRegister

Pour implanter ces eventRegister, nous nous sommes d'abord heurtés à un problème de taille: comment fonctionnent-ils et comment les intégrer à notre architecture et à notre code ?

Au moment où on fait appel à eventRegister, celui-ci doit créer un callBack. C'est dans cette méthode que le tuple de l'abonnement sera recherché et si besoin renvoyé au client via la méthode call() du callBack. Lorsque celle-ci est appelée on va commencer par regarder si l'abonnement peut être satisfait, c'est-à-dire qu'il y a assez de tuples correspondant pour que le callback renvoie faux. Si ce n'est pas le cas, il faut enregistrer le callback en l'attente de nouveaux tuples correspondant.

Cela impose deux choses, la première étant d'implanter un nouveau callBack : MyCallback, qui a dans sa fonction call tout le fonctionnement nécessaire pour renvoyer le tuple au client (ce code sera détaillé en II.3). La deuxième chose est que l'on doit enregistrer ces callBack et lors de l'ajout d'un tuple les parcourir, pour cela nous avons créé une class bookList qui contient une liste

de Callback et une liste de tuples ajoutés en parallèle pour savoir à quoi correspond chaque callback et une méthode notify qui se charge à chaque écriture de vérifier les abonnements et faire appel à la méthode call() du bon abonnement si le tuple est disponible.

II.3 Codage

Dans tous les pseudo codes suivant :

- moniteur est un Lock dont newTupleTake et newTupleRead sont des conditions correspondant respectivement au déblocage d'un take et d'un read
- readCount et takeCount sont respectivement le nombre de read en attente et de take en attente
- liste correspond à la liste de tuples de l'espace mémoire
- template est le tuple recherché
- bookList est la liste de Callback dans BookList et liste_tuple la liste de tuples dans BookList

II.3.1 Methodes read, take

Le pseudo-code pour les méthodes read et take sont identiques à quelques éléments près. Pour la méthode read les éléments spécifiques seront en rouge et en bleu pour le take.

```
Tuple résultat = null
Booléen solution trouvée = faux
moniteur.lock()
Tant que(!solution trouvée)
    tant que(!solution trouvée et i<liste.size())
        si ((liste.get(i)).matches(template))
            resultat = liste.remove(i)
            takeCount = takeCount -1
            resultat = liste.get(i).clone()
            readcount = readcount – 1
            solution trouvée = vrai
        fin si
        i = i + 1
    fin tant que
    si(! Solution trouvée)
        newTupleTake.wait()
        newTupleRead.wait()
    fin si
fin tant que
moniteur.unlock()
return résultat
```

II.3.2 Méthodes tryRead, tryTake

Ces methodes sont les mêmes que read et take à deux differences près :

- Si la solution n'est pas trouvé, on ne se bloque pas sur une condition.
- Il n'y a pas de boucle qui englobe la méthode en attendant de trouver une solution.

II.3.3 Méthodes readAll, takeAll

Ces méthodes ne diffèrent pas beaucoup de tryRead et tryTake, la seule différence réside dans le résultat : ce n'est plus un tuple mais une liste de tuples, ainsi lorsque l'on parcourt la liste à la recherche de tuples correspondants, on ne s'arrête plus à la première solution trouvée : le booléen solution trouvé disparait.

II.3.4 Méthode call de la classe MyCallback

Cette méthode ne fait que décrémenter un compteur (le nombre d'abonnements à renvoyer) et fait afficher le tuple reçu par l'interface graphique de l'utilisateur.

II.3.5 Méthode register de la classe bookList

La méthode eventRegister de CentralizedLinda ne fait que faire appel à cette méthode qui renvoie les tuples de l'abonnement immédiatement si possible et, si après avoir épuisé les tuples correspondants le callback renvoie vrai, on ajoute ce Callback à la bookList.

```
Booléen registerCallback = vrai          //booléen vrai si on doit enregistrer le callback
Booléen template présent = vrai          //booléen vrai si le template
Tant que (registerCallback et template présent)
        Template présent = faux
        Pour i de liste.size()-1 à 0 faire
                Si (liste.get(i).matches(template)
                        Template présent = vrai
                        registerCallback = callback.call(liste.remove(i))
                fin si
        fin pour
fin tant que
si (registerCallback)
        bookList.add(callback)
        liste_tuple.add(template)
fin si
```

II.3.6 Méthode notify de la classe bookList

La méthode notify cherche si un abonnement correspond au tuple ajouté et si le Callback renvoie faux, on l'enlève de la liste ainsi que son tuple. De par la structure du code on implante une priorité FIFO pour les abonnements.

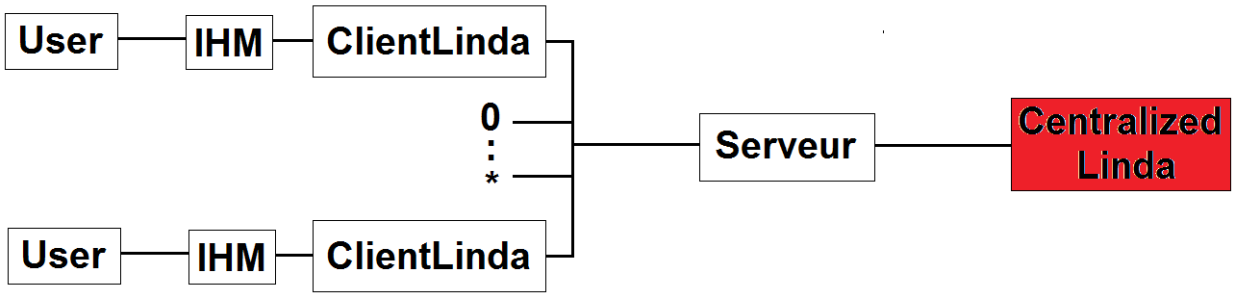
```
booléen résultat = faux
booléen abonnement trouvé
i = list_tuple.size()-1
tant que (i>=0 et !abonnement trouvé)
        Si (t.matches(list_tuple.get(i)))
                Résultat = vrai
```

```
                Si (!bookList.get(i).call(t))
                        booklist.remove(i)
                        list_tuple.remove(i)
                fin si
        fin si
fin tant que
return résultat
```

III Linda avec accès à un serveur distant

III.1 Architecture

L'architecture de cette version de Linda est représentée par le schéma ci-dessous :



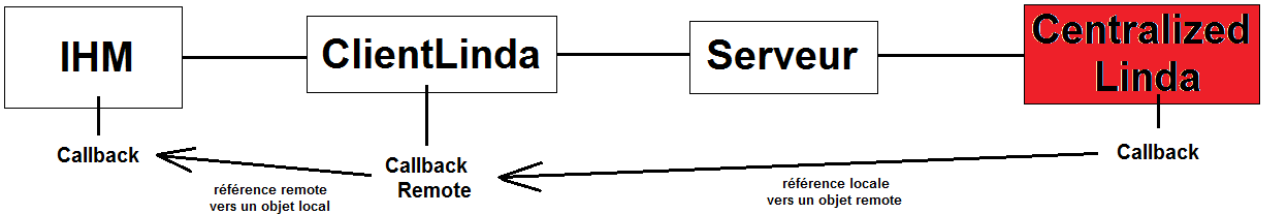
Côté utilisateur, on a toujours la même chose avec une interface graphique mais cette fois-ci, le client est un ClientLinda. Ce client est connecté à un serveur qui centralise les requêtes. C'est ce serveur qui va ensuite appeler les méthodes de CentralizedLinda pour agir sur l'espace mémoire de tuples.

Le serveur ne fait littéralement que transmettre les requêtes en appelant les méthodes de CentralizedLinda pour effectuer les actions. Nous n'avons pas du tout touché à CentralizedLinda.

III.2 EventRegister

Les eventRegister sont le vrai défi de cette implantation. Il y a besoin de plusieurs Callback pour coder les abonnements, un premier est détenu par la base de donnée, il est sérialisable. Le deuxième, remote, est possédé par le client et va permettre de faire le lien entre le client et le troisième callback détenue par l'IHM, c'est-à-dire entre client et utilisateur.

On a ainsi un schéma de ce type :



III.3 Codage

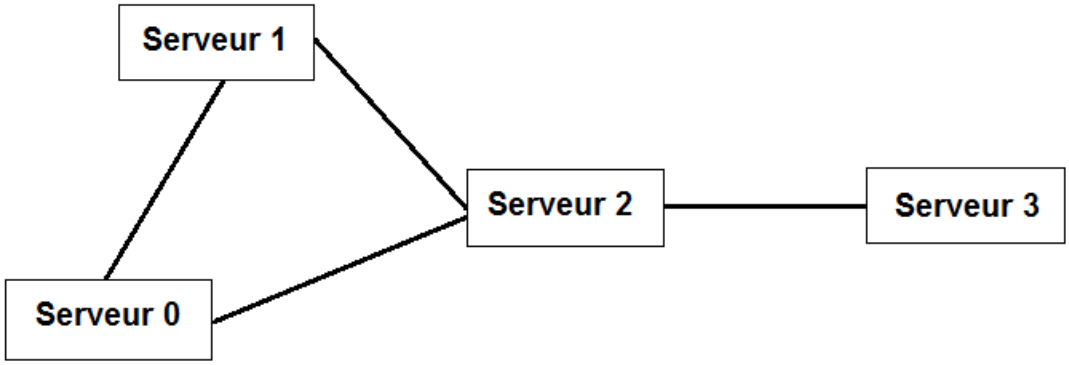
Dans cette implantation de Linda, il n'y a pas réellement eu de code spécial à écrire étant donné que nous avons réutilisé entièrement la classe CentralizedLinda. Ainsi dans le serveur on ne fait que appeler les méthodes de CentralizedLinda et dans ClientLinda nous appelons celles du serveur.

IV Linda avec accès à un Linda en multi-serveurs

IV.1 Architecture

Les différents serveurs ne sont pas forcements tous relier entre eux. Chaque serveur mémorise la liste des ID des serveurs à qui il est connecté, et transmettra les messages à ceux-ci. Le coté client n'est pas changé par rapport à la version mono-serveur.

Dans les exemples que nous avons utilisés, nous avons 3 serveurs reliés entre eux et un relié uniquement à l'un d'entre eux.



Lors d'un appel à une fonction dans le serveur, c'est le processus du client qui effectue tous les échanges entre les serveur. Le code aurait été plus optimisé si on avait un nouveau Thread qui tournait pour appeler les fonctions, mais cette version plus simple nous permet d'être sûr que l'échange entre les serveurs est fini lorsque le client finit l'appel à la fonction. Les procédures appelées dans les autres serveurs n'étant pas bloquantes, on est sûr que le client ne sera pas bloqué.

L'utilisation d'une ConcurrentLinkedQueue<Tuple> pour ranger les tuples du serveur nous assure le côté concurrent des requêtes, et permet de ne pas bloquer le moniteur lors des appels à des

fonctions non bloquantes (comme try take, read all …).

Nous n'avons pas codé de priorités sur le multi-serveur faute de temps, si ce n'est qu'un Callback enregistré dans un serveur a la priorité sur tout le reste.

IV.2 EventRegister

Nous n'avons pas fini de bien traiter les Callback dans la version multi-serveur. Ainsi le client peut créer un Callback qui peut lui renvoyer immédiatement des tuples s'il y en a présent sur les servers. Pour ce faire nous faisons appel au tryTake que nous avons implémenté.

```
eventRegister(Tuple t, Callback cb)  {
Tuple rep =  tryTake(t);
tant que (rep != null && register vrai){
        register= (cb.call(rep))
        rep = tryTake(t)
}
        si register  on enregistre le callback
}
```

Ensuite nous Callback effectue un take seulement si le tuple est poster sur le server ou est enregistrer le callback, un read sinon. Ce problème vient d'une difficulté à faire supprimé un tuple dans un serveur distant, sans faire un nouveau try-take pour tous les Callback de tous les serveur.

Afin que le callback fonctionne on utilise comme dans la version mono Serveur trois Callback, avec un Remote coté client et un Callback prenant celui-ci en argument et appelant juste son call.

VI.3 Codage

L'échange entre serveurs est gérer à l'aide de MessageServer, qui est un objet qui transite d'un serveur à un autre, contenant :

-Un Objet correspondant soit au tuple de la demande, soit le tuple ou la liste de tuple réponse.

-La liste des id des serveurs déjà visité, pour qu'un message ne tourne pas en rond entre plusieurs serveurs.

Cette liste contient aussi en première position l'id du serveur ayant fait la requête, afin de pouvoir lui renvoyer la réponse.

-L'id de la demande, pour que le serveur recevant une réponse puisse savoir à quel client elle correspond.

Cet id est déterminé par le serveur au moment de la requête.

-La nature du message, savoir s'il s'agit d'un take, un read, un tryRead...

Ensuite les échanges sont traités dans les serveurs à l'aide de 4 procédure :

send(MessageServer): envoi aux serveurs adjacents le message, s'il ne l'on pas deja traités, ou envoie le message au serveur ayant créé la requête si le message est de type réponse (Dans ce cas son ID est à la première place de la liste des ID du message).

```
Void send(MessageServer message) {
```

si (nature message = reponse ou reponse_liste) {

Server Linda sl = (ServerLinda) Naming.lookup(adress + message.getList().get(0));
//message.getList().get(0) contient l'id du server ayant créé le message
//Le string adress contient l'adresse commune à tous les serveurs
sl.receive(message);
} sinon {

pour tous les voisin {
On regarde s'il a déjà traité le message, sinon on rajoute son id dans la liste du message et on retient qu'il faudra lui envoyer le message.
}
On envoie ensuite le message a tous les serveur que l'on a noté.
//Cette façon de faire permet d'éviter d'envoyer un message par exemple aux serveurs 2 et 3, et qu'ils se l'envoient mutuellement ensuite.
}

Receive(MessageServer) : Traite une demande d'un serveur. Selon la nature du message, on fera appel à un try-take particulier ou à un take-all (explicité plus loin). Un message peut aussi être de type réponse, auquel cas le serveur sait que cette réponse est pour l'un de ses clients et les réveilles.

...
Si (natureMessage== reponse) {
...
SolutionClient.put(message.Iddemande,message.Tuple);
reveil.signalAll();
}...

Le client ira ensuite chercher le tuple qui l'intéresse dans SolutionClient.

Enfin un message peut être de type signal, et signale alors aux serveurs la présence d'un nouveau tuple.

try_Take_Read_Linda_Serv(MessageServer ms, boolean take) : Traite de façon similaire un try-take et un try-read selon la nature du booléen. Cette fonction est appelée dans le cas où le message serveur est de type read, take, tryRead ou tryTake. En effet cette procédure est non bloquante et permet de factoriser les opérations, seule la nature du message diffère.

Take_Read_All_Linda_Serv(MessageServer ms, boolean take) : De la même façon cette procédure est appelé dans le cas d'un readAll ou d'un TakeAll. Le server effectue tout d'abord une recherche des tuples correspondant au template contenu dans le message et les transmet au serveur ayant créé la requête. Ensuite il transmet la requête aux serveurs adjacents ne l'ayant pas encore traités.

Ces deux dernières procédures sont similaires a des tryTake ou TakeAll, mais traites des message serveurs, et effectue un send(message) plutôt que de renvoyer le tuple réponse. Elles sont utilisé par les serveur pour faire des recherches internes, commandés par un autre serveur.

V Tests Complémentaires

L'idée générale est de vérifier que la publication et la réception de tuples se fait correctement entre les clients. Pour simplifier, ces tests sont faits avec le même format de tuples, mais chaque client peut sélectionner n'importe quel motif de tuple.

Intéressons-nous aux deux classes de tests : TestReadWrite.java et TestMultiServer.java.

V.1 TestReadWrite

On commence par créer un serveur de type linda.shm.CentralizedLinda, puis on lance deux Threads.

Un premier Thread est lancé; il tente immédiatement de lire ou retirer cinquante tuples d'affilée. Puisqu'aucun tuple n'a encore été créé, ce thread reste en attente.

Un second Thread est également lancé, mais il est mis en pause pendant 5 secondes. Ensuite, il publie 25 tuples. La réception par le premier Thread commence au plus tôt juste après sa publication, mais peut aussi avoir lieu plus tard (par exemple après la publication du dixième tuple).

Après une pause de 5 secondes, le second Thread émet 50 tuples qui sont reçus de la même manière par le premier Thread.

V.II TestMultiServer

On commence par créer un multi-serveur constitué de 4 serveurs.

On lance un Thread récepteur qui demande 50 tuples au serveur 0 et qui se place immédiatement en attente.

Le second Thread crée un client sur le serveur 0, attend 5 secondes, puis émet 5 tuples qui sont reçus par le premier Thread.

Ensuite, l'émetteur envoie des tuples sur un autre serveur, par exemple le 2; ces tuples sont redirigés jusqu'au serveur 0 qui les transmet alors au premier Thread.

Conclusion

Dans ce projet, nous avons pu appliquer dans un exemple concret les nouveaux concepts que nous avons pu manipuler en TP et étudié en systèmes concurrents ainsi qu'en intergiciel. Nous avons ainsi pu de manière autonome développer une application concurrente tout en maniant des objets Remote ou Serializable.

La véritable difficulté du projet a été de coder les eventRegister, ceux-ci nous ont amené à modifier de grande parties de notre code pour pouvoir les implanter, cependant ils nous ont également permis de nous rendre compte de certaines erreurs qui s'étaient glissées dans notre code.

Dans ce genre de développement, nous avons trouvé que le partage du travail était très difficile, surtour au départ, en effet il était difficile de coder une partie du projet sans savoir exactement comment les autres faisaient le reste. Ceci nous a forcé à réellement poser une architecture au départ et à réfléchir à tous les concepts avant de pouvoir se lancer dans le vif du sujet.

Malheureusement, nous avons été rattrapés par le temps et n'avons pas pu finir l'implantation des eventRegister en multi-serveur. Si nous devions apporter une modification profonde à notre code, cela aurait été en créant un objet Requête contenant, un tuple et une condition. Ainsi lorsque le client fait une action bloquante, on crée un objet requête et pour un abonnement on lance en plus un thread qui se bloque sur la condition de la requête. Ainsi lorsqu'un tuple est rajouté on peut entièrement contrôler quelle action sera libérée. Cela aurait pu nous simplifier le travail et surtout nous permettre de coder plus facilement les eventRegister dans toutes les versions. Malheureusement, lorsque nous avons vu cette solution il était trop tard pour modifier autant notre application.