RAPPORT

Implantation d'un espace partagé d'objets tuple en version centralisée, mono-serveur et multi-serveurs

réalisé par Bakhouch Taha, de Langlard Mathieu et Icho Othman

SOMMAIRE

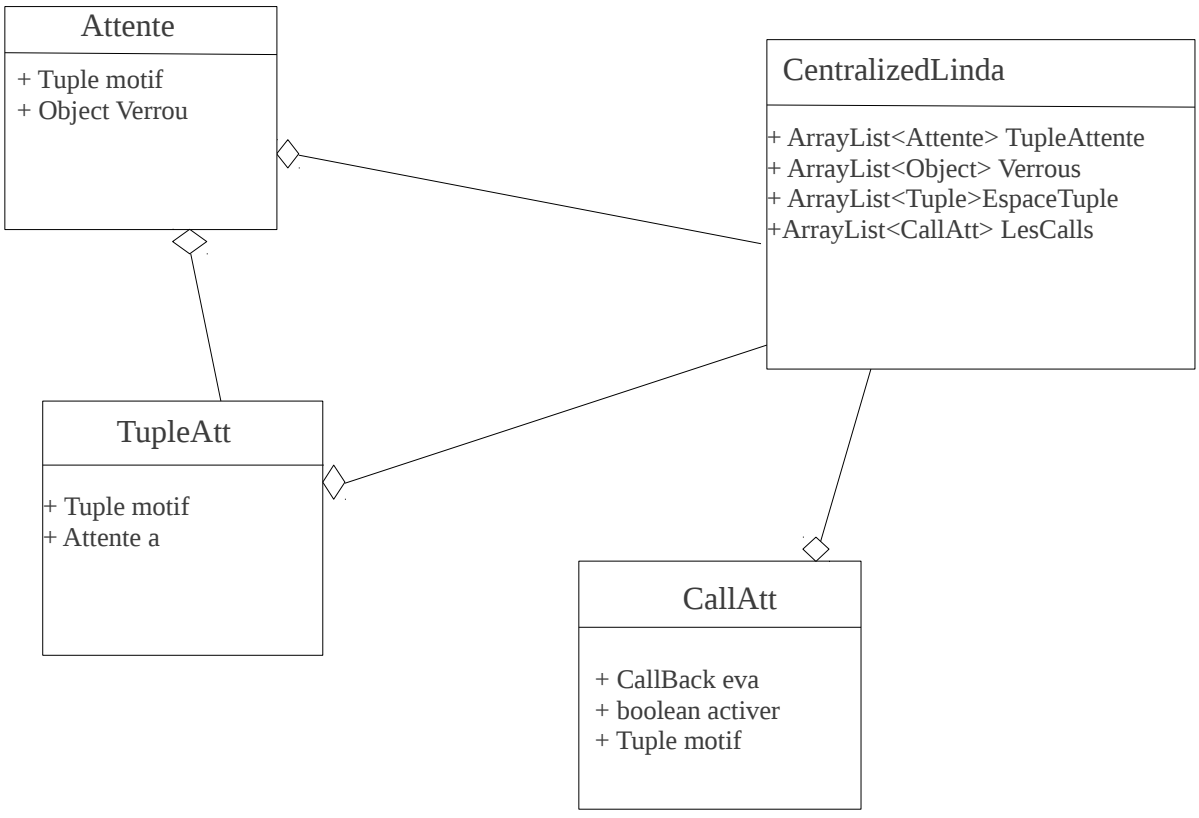
I) Première partie : version centralisée.

1) Spécifications :

Les spécifications du projet au niveau de la synchronisation restent libérales : le but principal étant de ne pas avoir des résultats incohérents (la priorité au t*ake* ou au *read* n'est pas spécifié ; on peut débloquer un *read* ou plusieurs ou encore débloque un *take* ou plusieurs). De même, en ce qui concerne les *callback* il n'y a rien de détailler au sujet du comportement que ceux ci-doivent adopter : les risques d'interblocage et de boucles infinis ne sont pas forcément à gérer.

2) Explications et choix d'implémentation :

Tout d'abord, nous avons opté pour l'utilisation de *synchronized* non pas sur les méthodes, car cela nous semblait trop restrictif, mais sur des verrous d'objets de type *Attente*. Voici, l'architecture de notre première solution pour en avoir une première idée :



La décision importante que nous avons prise pour la version *CentralizedLinda* est de pourvoir permettre à plusieurs threads d'exécuter une méthode *read* en même temps : il n'y a pas d'exclusion mutuelle des *read* entre eux. Cependant, on inclut un nombre maximum de 30 threads qui peuvent exécuter un *read* simultanénement. De plus, si jamais il existe des threads en attente sur

des méthodes *read* et *take,* on donne la priorité au read mais seulement 3 threads bloqués sur un *read* peuvent se débloquer successivement, sinon ce sera un thread bloqué sur une méthode take qui sera débloqué.

En ce qui concerne les méthodes tr*yRead*, *tryTake*, *readAll* et *takeAll*, qui sont non bloquantes, on respecte les spécifications énoncées dans le sujet du projet tout en permettant toujours que plusieurs threads puissent effectuer des read, des *tryRead* et des *readAll* simultanément (étant donné que ces méthodes ne modifient pas l'espace de tuple).

3) Description des méthodes principales :

3.1) Méthode write :

On utilise une méthode récursive *write_aux* qui permet d'obtenir tous les verrous pour être certain que le thread qui exécute le *write* est bien le seul à avoir accès à l'espace de tuple. Cette méthode récursive retourne un booléen qui est vrai si un callback a déjà consommé le tuple et false sinon. Le raffinage niveau R0 de la méthode *write* est le suivant :

i) Obtention des verrous (appel à la méthode write_aux) ;
ii) Ajout du tuple dans l'espace de tuple (si Callback ne l'a pas consommé entre temps);
iii) Recherche des threads en attente sur le motif déposé et réveil de tous les threads (read et take) ;

3.2) Méthode read :

Le raffinage niveau R0 de la méthode read est le suivant :

ii) Obtention d'un verrou pour avoir accès à l'espace de tuple (ce verrou n'exclut pas les threads qui exécute un read (ou les autres demande de lectures) sur l'espace de tuple mais les threads qui effectuent n'importe quelles autres méthodes ;
iii) Recherche d'un motif qui correspond dans l'espace de tuple ;
iv) Si un motif est trouvé alors il retourne une copie ;
v) Si un motif n'est pas trouvé, création d'un objet de type Attente et blocage sur le verrou de cet objet si aucun motif d'objet Attente correspond celui demandé ou simplement blocage sur un objet Attente déjà présent dont l'attribut motif correspond au tuple demandé ;

3.3) Méthode take :

On utilise une méthode récursive *take_aux* qui permet d'obtenir tous les verrous pour être certain que le thread qui exécute le *take* est bien le seul à avoir accès à l'espace de tuple. Cette méthode récursive retourne un objet de type *TupleAtt*. Le raffinage niveau R0 de la méthode *take* est le suivant :

i) Obtention des verrous (appel à la méthode take_aux) ;
ii) Recherche d'un motif qui correspond dans l'espace de tuple ;
iii) Si un motif est trouvé alors il l'enlève de l'espace de tuple et le retourne ;

iv) Si un motif n'est pas trouvé, création d'un objet de type Attente et blocage sur le verrou de cet objet si aucun motif d'objet Attente correspond celui demandé ou simplement blocage sur un objet Attente déjà présent dont l'attribut motif correspond au tuple demandé ;

4) Problèmes rencontrés et critique de la solution:

La difficulté de cette version centralisée résidait essentiellement dans le fait d'avoir adopter une solution optimale du point de vue du parallélisme. En effet, en laissant la possibilité à plusieurs threads de faire appel aux méthodes *read, tryRead et readAll* simultanément, on a du prendre en comptes des aspects de la synchronisation qui n'apparaissent pas s'ils l'on aurait opté pour une solution en exclusion mutuelle pour la totalité des méthodes de *CentralizedLinda* (synchronized sur les méthodes).

Cependant, il est vrai que même si cette solution permet la lecture de plusieurs threads en même temps (bonne parallélisation), le code reste difficile à implanter avec des synchronized et est également difficile à adapter et à modifier. De plus, le nombre de threads qui peuvent effectuer une lecture en parallèle reste arbitraire (fixé au nombre de 30 dans notre cas).

II) Seconde Partie : version mono-serveur.

2.1) Spécifications :

Dans cette version, l'Espace de tuple sera centralisé sur un serveur distant. Pour avoir accès à cet espace, le client doit faire appel au serveur selon le modèle client-serveur en utilisant l'API RMI de java. L'unique contrainte à respecter réside dans le fait que la classe *LindaClient* fournie doit être une implémentation de l'interface Linda (interface à ne pas modifier).

2.2) Architecture :

Une première architecture consistait tout simplement à faire appel, au niveau de la classe *LindaClient*, aux méthodes *read*, *take*, *write* (ainsi que toutes les autres) du serveur distant *LindaServer.* En effet, la classe *LindaServer*, qui hérite de la classe *UnicastRemoteObject* et implémente l'interface *LindaRMI* qui elle hérite de *Remote*, possèdent alors des méthodes qui peuvent être appelées à distance (Remote Procedure Call). De plus, les méthodes de *LindaServer* font alors appels aux méthodes de *CentralizedLinda* de la partie précédente : en effet notre espace de Tuple se trouve au niveau du serveur. Voici un schéma résumant l'architecture proposée :

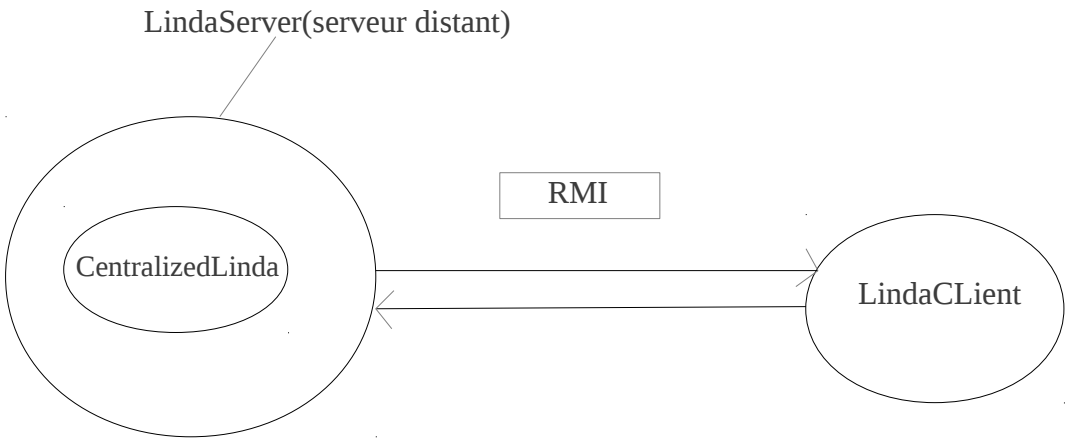


fig2-Architecture n°1

2.2) Difficultés rencontrées :

La seule difficulté qu'il a fallut résoudre est le celle des Callback. En effet, il est spécifié de ne pas toucher à cette classe. Cependant, lors de l'appel à une méthode eventRegister on passe en argument un Callback qui n'est pas un objet Remote ou Sérializable. Il faut donc passer par une sérialisation au moment de l'envoi et une dé-sérialisation au moment de la réception par le client.

III) Troisième Partie: version multi-serveurs.

1) Spécifications :

Les spécifications restent toujours libérales et nous laisse la possibilité de choisir l'architecture du réseau (en anneau ou autre). Cependant, il nous ait demandé de laisser la possibilité de créer des serveurs autant que possible (n>1) et de respecter le fait que chaque client reste connecté à un unique serveur(pas de possibilité de se connecter à un autre serveur en cours de fonctionnement). Ils nous ait simplement demander

2) Choix d'implémentation :

L'objectif principal étant de synchroniser l'ensemble des clients et des serveurs entre eux afin d'avoir des résultats cohérents, nous avons pris le parti de programmer une solution simple quitte à perdre en terme d'efficacité (contrairement à la partie une où nous avons essayer d'implanter une solution rapide et efficace). Nous utilisons toujours des synchronized mais cette fois si sur les objets serveur de type *LindaServer*,

L'architecture du réseau est la suivante : tous les serveurs se connaissent les uns les autres. Il y a une seule classe *LindasServer* qui va crée le registre de noms et à chaque fois que l'on veut lancer un serveur on exécute la méthode main de cette classe (on donne en paramètre du main l'URI du serveur que l'on lance). Lorsqu'un serveur ce connecte il signale à tous le monde qu'il s'est connecté et il récupére la liste de tous les serveurs connectés avant lui. Ainsi, lorsqu'un client demandera un tuple à son serveur, et si aucun tuple ne correspond au motif demandé par le client, son serveur jouera le rôle de client et enverra une demande à tous les autres serveurs en même temps étant l'interconnexion entre serveurs. Le comportement des différents threads seront décrit plus précisément dans la prochaine partie.

Chaque serveur sera constitué d'une liste d'objet attente (sur la même idée que la partie 1), d'une liste d'objet *LindaServer* pour permettre la propagation des requête si besoin et d'un attribut URI qui correspond à sa propre URI .

3) Description des méthodes principales :

3.1) Méthode « write » :

Lorsqu'un thread souhaite écrire un tuple il faut avant tout vérifier s'il n'y a pas de thread en attente du tuple passé en paramétre de la méthode *write* (on parcours la liste d'objet Attente en attribut du server) ou encore s'il n'y a pas de *CallBack* activé qui le consommerait (la partie *CallBack* n'a pas été géré pour la version multi-serveurs). Pour permettre cela, l'objet de Type Attente est composé d'un attribut motif de type tuple correspondant au motif attendu par un client et d'un attribut serveur de type *LindaRMI* pour connaître le serveur en attente du tuple. De plus, la liste d'objet Attente est locale à chaque serveur et c'est pour cela qu'il est impératif d'actualiser cette

liste sur tous les serveurs à chaque nouvel ajout par n'importe quel serveur d'un objet de type Attente. Voici, ci-dessous, le raffinage niveau R0 de cette méthode :

i) On vérifie s'il y a des threads en attentes sur motif correspondant au tuple ;
ii) Si oui alors, on écrit le tuple sur le serveur qui a besoin du tuple et on réveille tous les client sur le serveur où on écrit le tuple ;
iii) Si non on écrit le tuple sur le serveur avec lequel le client qui fait la méthode write est connecté ;

3.2) Méthode take :

Lorsqu'un client effectue un *take*, l'on va tout d'abord effectuer un *tryTake* sur son serveur. Deux cas se présentent alors : soit le tuple est présente sur son serveur et la méthode *take* renvoi le tuple soit il ne l'est pas et alors la requête est propagée aux autres serveurs. Dès que un serveur répond favorablement à la requête alors on n'effectue plus de t*ryTake* et on retourne le tuple trouvée au serveur qui l'a demandé. Vois le raffinage de cette méthode :

3.3) Méthode read :

La méthode read est similaire à la méthode take. Son raffinage est identique à la méthode take, la différence repose ,comme dans autres parties, qu'on ne renvoi qu'une copie du tuple.

4) Difficultés rencontrées :

La première difficulté rencontrée est provenue de l'interconnexion des serveurs entre eux. Nous avons passé beaucoup de temps avant de réussir cette interconnexion.

D'un point de vue algorithme et architecturale notre solution proposée semble simple et assez facile à implanter. Cependant, il réside des erreurs de programmation en ce qui concerne les méthodes bloquantes *take* et *read.* Cela ne provient pas de la structure du code mais plutôt d'une erreur indésirable qui s'est glissée dans le code.

IV) Tests :

4.1) Version centralisée :

BasicTest1 : OK ;

BasicTest2 : OK ;

BasicTest3 : OK (test du tryTake) ;

BasicTest4 : OK (test du tryRead) ;

BasicTestCallBack : OK (test du takeAll);

BasicTestCallBAck2 : OK ;

BasicTestAsynCallBack : OK (test du readAll);

Test du jeudi 18/01/2013 réalisé par Philppe Mauran : OK ;

4.2) Version mono-serveur :

BasicTest1 : OK ;

BasicTest2 : OK ;

BasicTest3 : OK (test du tryTake) ;

BasicTest4 : OK (test du tryRead) ;

BasicTestCallBack : OK (test du takeAll);

BasicTestCallBAck2 : OK ;

BasicTestAsynCallBack : OK (test du readAll);

Test du jeudi 18/01/2013 réalisé par Philppe Mauran : OK ;

4.3) Version multi-serveurs :

TestMultiServeur1 : OK ;

TestMultiServeur2 : OK ;

TestMultiServeur3 : OK ;

TestMultiServeur4 : KO ;