Rapport du Projet Linda

projet réalisé par Matthieu Legrand et Anna Stephany
SN FISA 2A

Introduction

Le projet Linda avait pour objectif d'implémenter deux versions de Linda. La première version en mémoire partagé et la deuxième en mode client et serveur.

Par la suite, on a également dû réfléchir à des solutions pour la tolérance aux fautes, comme la sauvegarde et un système de backup.

Nous avons utilisé les concepts et technologies vus en cours d'intergiciels (RMI), mais aussi les verrous et systèmes de synchronisation vus en cours de système concurrents.

Version en mémoire partagée CentralizedLinda

CentralizedLinda est le cœur du projet, cette classe contient toute la logique de Linda pour la version centralisée de l'api et pour la version distante de l'API.

Cette classe contient la liste de tuples et a comme responsabilité de modifier cette liste et d'ordonnancer ces modifications et accès.

Les structures de données et informations

Lorsque l'on parle de la conception d'une application, la première chose qu'on doit évaluer sont les données gérées par cette application et les accès et mutation de ces données.

Les tuples

Logiquement, une classe ayant pour responsabilité de gérer une collection de tuples aura une liste de tuples. Ces données n'ont pas demandé trop de réflexions. Les seuls choix techniques sont :

- Quel type de liste devrions-nous utiliser :

    Les tuples peuvent être dupliquées donc ça exclut les Sets.

    Les tuples ne sont pas ordonnables ce qui exclut les SortedList

    On n'a pas d'identifiant (hash, uid) pour les tuples, qui pourrait faciliter la recherche, donc ça exclut les maps

    Ce qui nous laisse avec les listes. Considérant que la liste est une liste souvent éditer (particulièrement pendant le parcours) LinkedList est probablement la meilleure structure de données

- Faut-il utiliser une SynchronizedCollection ? :

  Non, l'atomicité des SynchronizedCollection n'est pas suffisante pour les accès et mutations qu'on va avoir. [voir Concurrence et Synchronisation]

Concurrence et synchronisation

CentralizedLinda est accédé et modifié par plusieurs threads simultanément. Il nous faut donc assurer la cohérence des données. Idéalement, cette cohérence ne serait pas au prix de la concurrence.

Les verrous et synchronisation.

Nous avons principalement une ressource à protéger des modifications concurrentes : la liste des tuples.

Pour ce faire nous allons utiliser une RWReantrantLock qui nous permet une meilleure précision, une meilleure parallélisation et d'intercaler les sections critiques.

Nous aurons aussi besoin de bloc synchronized pour les variables de conditions.

Les lectures bloquantes

Pour faire les lectures bloquantes (read & take) il va nous falloir des variables de conditions. Dans notre situation, on a un objet très approprié pour nos variables de transition : les tuples template. En effet, si on les garde dans une liste et qu'on endort les threads dessus, on peut réveiller les threads avec une précision parfaite.

Donc ce que fond nos read et take :

- Ils appellent leurs try
- S'ils ne reçoivent pas de résultat, ils s'endorment sur leur template après l'avoir ajouté à la liste.
- Une fois réveillés, ils bouclent (récursion) pour retenter de prendre un tuple.

Les threads sont reveillé par le thread qui write un tuple. La fonction write ajoute le tuple à la liste avant de parcourir les tuples template pour réveiller les read et les take qui correspondrait au nouveau tuple.

Les Callbacks

Les callbacks fonctionnent sur le même principe :

- vérifier si un tuple qui match existe déjà (dans le cas des callback immédiate)
- sinon ajouter son template avec son callback dans une liste

Similairement, les callbacks seront appelés par write dans la fonction wakeCallbacks. On doit bien faire attention à parcourir la liste des tuples pour trouver les matchs en section

critique. Puis les appeler hors section pour assurer l'absence d'interblocage dans le cas ou le callback appellerait une méthode de CentralizedLinda.

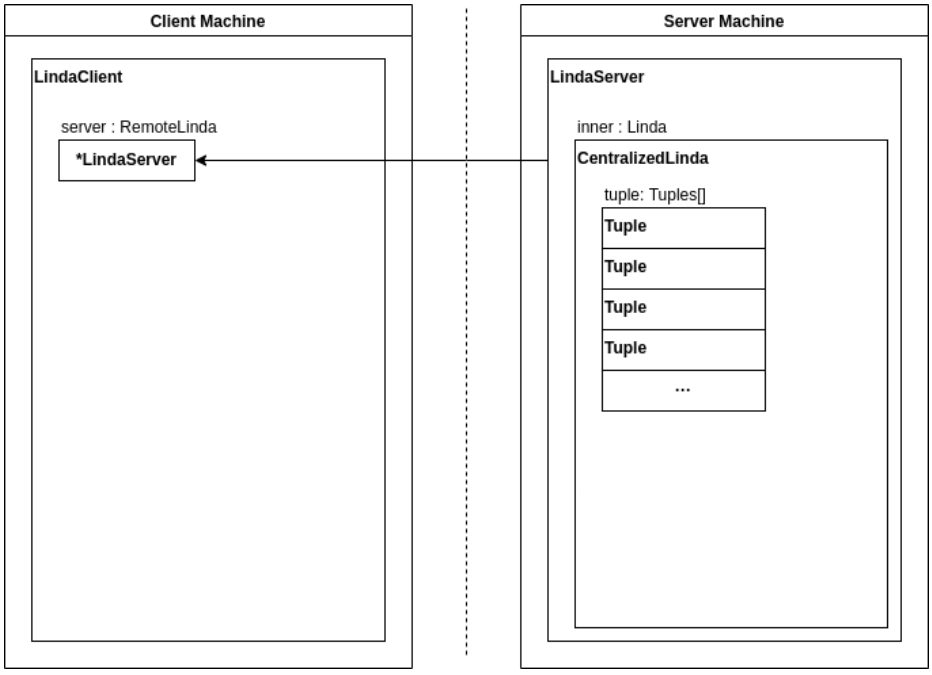
Version client / MonoServeur

LindaServer & LindaClient

Pour concevoir LindaServer et LindaClient, nous avons beaucoup utilisé le patron proxy.

LindaServer est un proxy de CentralizedLinda, ou plus précisément de Linda, qui transforme l'interface Linda en RemoteLinda qui pourra être utilisé par LindaClient pour appeler les méthodes du Linda sous-jacent (CentralizedLinda).

LindaClient est un proxy de LindaServer, ou plus exactement de RemoteLinda, qui transforme l'interface RemoteLinda en Linda qui pourra être utilisée par le code client.

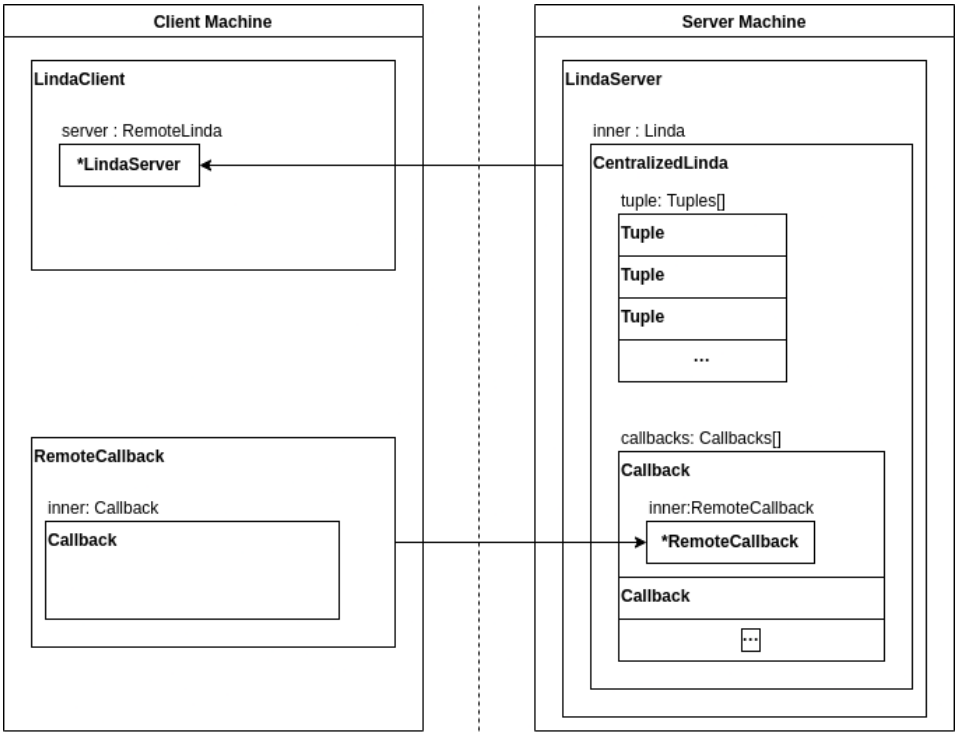


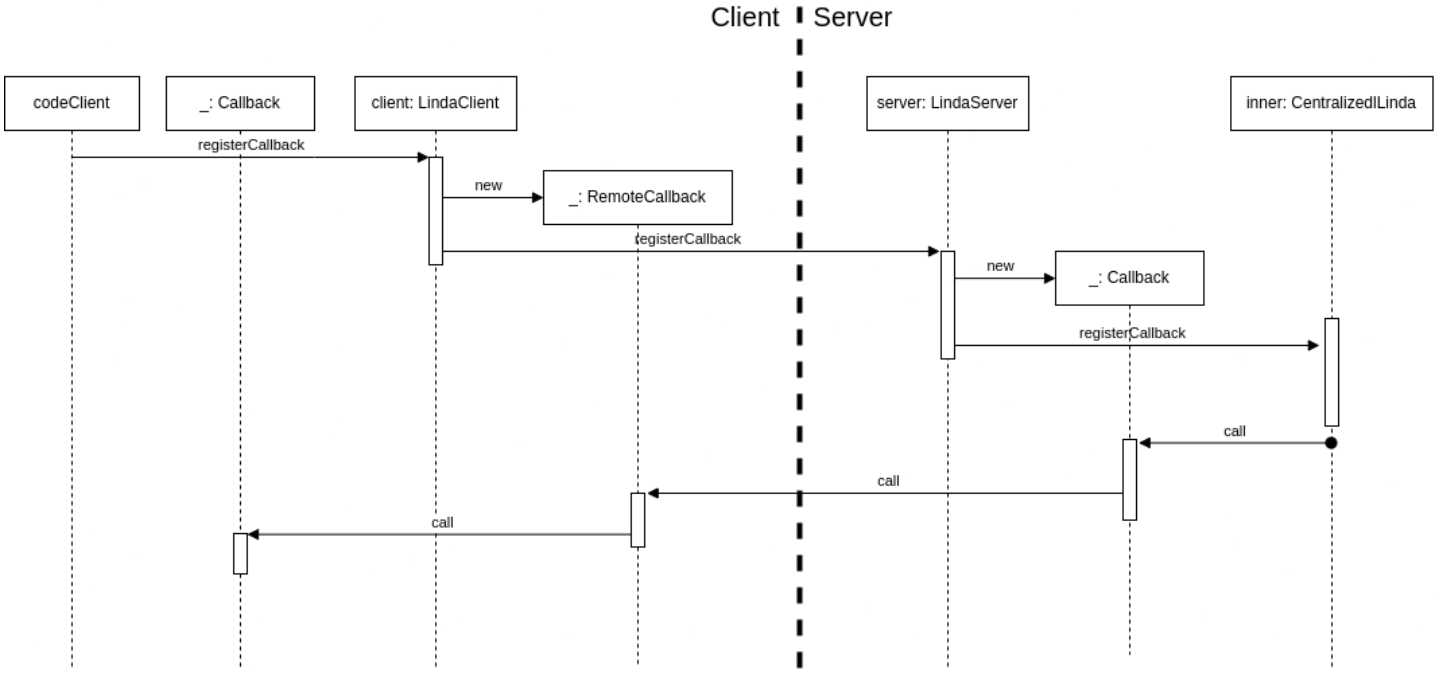
Callback

Similairement les callback vont beaucoup utiliser de proxy.

Le code client nous donne un objet callback qui n'a du sens que dans le contexte de la jvm du client. Il va donc falloir faire un RemoteCallback qui pourra être envoyé comme une référence sur le serveur.

Cependant, ce RemoteCallback ne correspond pas à l'interface attendue par le CentralizedLinda donc il va falloir recréer un callback qui contiendra et appellera le remote callback





Tolérance aux fautes

L’objectif de cette partie était de trouver une solution pour la sauvegarde et récupération de l’espace de tuples et mettre en place un système de backup.

Sauvegarde de l’espace des tuples

On a utilisé la sérialisation afin de sauvegarder l'espace des tuples dans un fichier, afin de pouvoir récupérer l'espace des tuples lors du redémarrage du serveur.

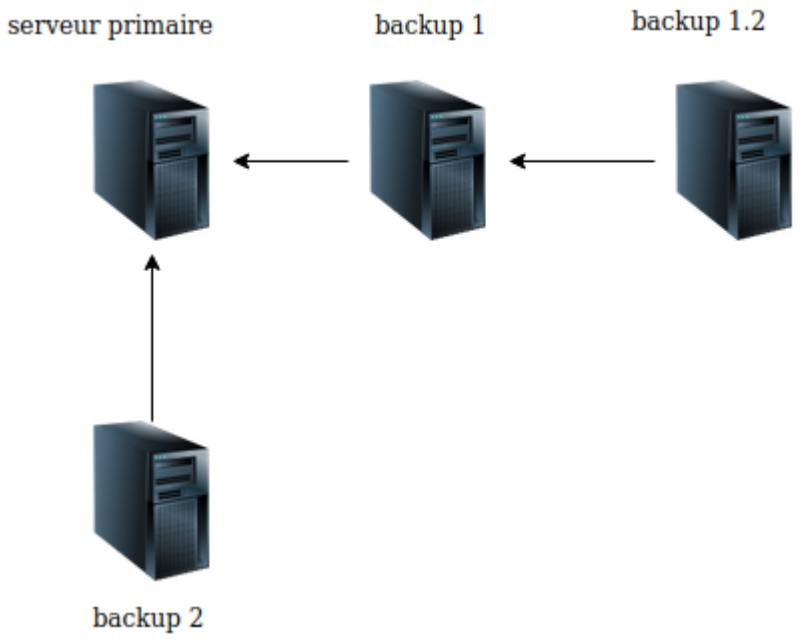
Pour cela, on a ajouté une méthode save et loadsaved au centralizedLinda qui permettent de sauvegarder et de recharger l'espace de tuples à partir d'un fichier.

Pour la partie, sérialisation en sauvegarde uniquement la liste des tuples et sauvegardé et les autres attributs sont "transient" et doivent être réinitialisé lors du rechargement.

Pour ce faire, la classe CentralizedLinda implémente l'interface "Serializable"

On a décidé de créer un timer au niveau du serveur qui déclenche une sauvegarde tous les 10 s et un shutdownlistener qui sauvegarder l'espace de tuples avant d'arrêter le serveur;

Système de Backup



On a choisi d'utiliser un système de serveurs chaînés où le serveur de backup récupère l'état (uniquement la liste des tuples) de son serveur primaire. À l'aide d'un callback, le serveur de backup récupère tous les n secondes et tous les n changements la liste des tuples.

Une fois que le serveur primaire ne répond plus, le serveur de backup prend le relais et coupe le lien avec son serveur primaire.

Le LindaClient n’a pas uniquement un serveur, mais une liste de serveur connus lors du démarrage. Une fois qu’une RemoteException est levée, il bascule sur le serveur suivant dans la liste. Vu que le serveur de backup récupère uniquement la liste des tuples, c’est au client de récupérer et recharger le eventregistry et callbacks auprès du nouveau serveur.

Pistes d’amélioration

Au lieu d’avoir une liste fixe de serveur, on pourrait se baser sur le service de nommage pour la re-connexion vers un nouveau serveur.

Pour le moment, si le serveur primaire tombe en panne, mais redémarre après un certain temps, il ne sera pas utilisé par le client .

Pour l’instant le basculement vers un serveur n’est pas transparent vis à vis d’un client.

Le Plan de test

On a trois fichiers de test, qui permettent de tester les méthodes du centralizedLinda : write, read, take , tryread, trytake, takeall, readall et eventregister

Le premier fichier “TestsLinda1 teste les read, write , take, takeAll et readAll séparément.

Il faut commenter et décommenter les lignes qui permettent de faire tourner les tests (test.run)

Le deuxième fichier “TestsLinda2” teste les eventregister et finalement le troisième fichier teste l’ensemble des opérations.

Après chaque opération, l’espace des tuples est affiché dans le terminal et c’est à l’utilisateur de vérifier si l’espace des tuples est cohérent par rapport aux opérations effectuées.

On n’a pas fait de tests de performance ou autres tests.

Pour la partie Serveur Client, on a testé en utilisant l’application Whiteboard.

Conclusion

Le projet nous a permis d’approfondir nos connaissances dans la conception de systèmes communicants dans un cadre bien posé, ce qui nous a évité de partir sur un projet non-réalisable. En plus de la partie intergiciel (RMI) , on a pu également revoir les notions de synchronisation et de verrous (système concurrent).

La troisième partie du sujet nous a comme même laissé une ouverture pour un choix de conception individuel.