Données réparties : Rapport de projet

SOUCHON Théo, ROGARD Diego, NEYENS Pablo, GROS Basile
(10 / 05 / 22)

Synthèse des corrections/modifications apportées à la version originale de la plateforme Linda

Reprise de l'architecture globale :

Nous avions dans la précédente version utilisée des structures peu claires et peu optimisées pour gérer les callbacks. Nous avons opté dans la nouvelle version pour un objet de type TriggerCallback que nous avons créé.

Ensuite le fonctionnement des read et take sont basés sur les callback ce qui n'était pas le cas auparavant.

Enfin, nous avons commencé à paralléliser notre algorithme pour augmenter les performances de ce dernier.

Améliorations envisagées à la plateforme Linda existante

Parallélisation :

Nous avons commencé la parallélisation de notre algorithme et l'avons comparé à la première partie.

Pour cela, nous avons déplacé les moniteurs pour réduire les sections critiques et donc augmenter la parallélisation.

Nous avons décidé de créer plusieurs threads pour le parcours de notre liste de tuple dans le but d'augmenter les performances pour les read.

Cache :

Les caches sont des write-through : chaque écriture est effectuée à la fois sur le cache et sur le serveur. Les caches ne possèdent qu'une copie de chaque tuple, et le serveur va vérifier à chaque Take qu'il ne reste aucun tuple identique dans l'espace et si c'est le cas appeler une méthode des caches pour détruire les tuples en cache.

Le cache est un linda classique. La propriété qu'il n'y a qu'un seul tuple de chaque valeur est garantie par les opérations entre les clients et le serveur. Les clients peuvent être accédés à distance par le serveur par rmi et vice-versa. Le serveur est initialisé avant et obtenu à l'aide d'un rmi registry, tandis que le client est envoyé à l'aide d'une méthode du serveur.

Les lectures et écritures dans les caches sont gouvernées par un seul verrou par client. Les appels au serveur sont effectués à l'extérieur de ce verrou pour éviter les interblocages. De même, le verrou du serveur est déverrouillé avant l'appel aux fonctions rmi des clients.

Multi-serveur :

Pour l'implémentation du multi-serveur, nous avons décidé de faire un système cyclique. Ce système fonctionne de la manière suivante :

- Plusieurs serveurs sont démarrés directement à partir d'un script java avec comme paramètre le nombre de serveurs que l'on souhaite démarrer.
- Write : write est implémenté de tel façon qu'avant d'écrire quoi que ce soit sur le serveur, une requête sera créée et parcourra l'ensemble des serveurs pour des read en attente et refera la même chose pour les takes.
- Take/Read : chacune des deux requêtes tentera de le faire sur son serveur et si cela échoue, il passera au suivant s'il effectue un tour complet sans succès, il se bloquera sur son serveur.
- Les requêtes try fonctionnent de la même manière en non-bloquant.
- Pour ce qui est des takeAll/readAll : ce sont des requêtes classiques qui s'étendent sur tous les serveurs.

Instrumentation

Nous avons créé une instrumentation des différentes implémentations de linda. Celle-ci peut être de deux types : lecture et exécution d'un fichier de test ou interpréteur de commandes. Pour le détail de l'utilisation, se référer à la documentation dans le dossier *Instrumentation/linda/DocumentationInstru.txt*.
Cette instrumentation permet de réaliser toutes les actions basiques de linda. On peut aussi écrire des boucles for et des commandes "sleep" dans les fichiers.
Cette instrumentation est surtout intéressante pour la version parallélisée car les informations que l'on cherche à obtenir sont le plus adaptées à cette version. Cependant, l'instrumentation prend aussi en compte les autres versions. Par rapport à la version multi-serveur, cette instrumentation n'a pas été adaptée pour et les fonctionnalités mises à dispositions n'étaient de toute façon pas intéressantes.
Nous avons utilisé cette application pour effectuer la comparaison des performances. En effet, exécuter l'instrumentation sur un fichier nous donne le temps réel d'exécution.

Comparaison de performances

Nous avons réalisé une comparaison de performance entre les différentes versions. D'abord, nous avons testé avec un fichier qui faisait un nombre de write, read et take équilibré. Nous

voyons que la version parallèle est la plus efficace suivie de la version utilisant le cache et enfin la version basique est la plus longue.

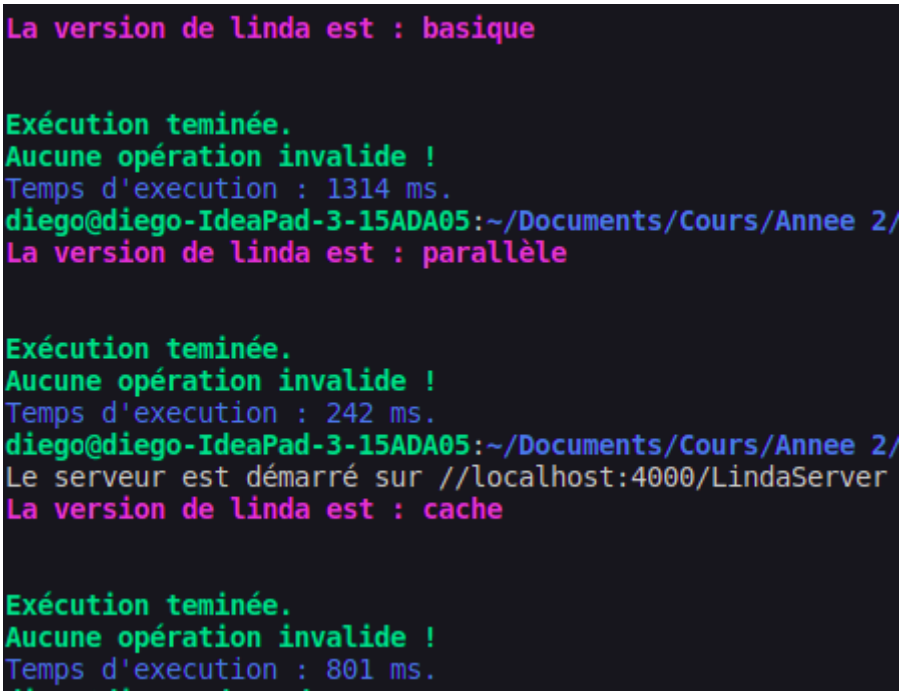

Figure 1 : Comparaison de performances : version équilibrée

Ensuite pour le deuxième test, nous avons fait beaucoup plus de read que de take. Ici, le cache n'est pas forcément plus efficace mais la parallélisation reste plus efficace.

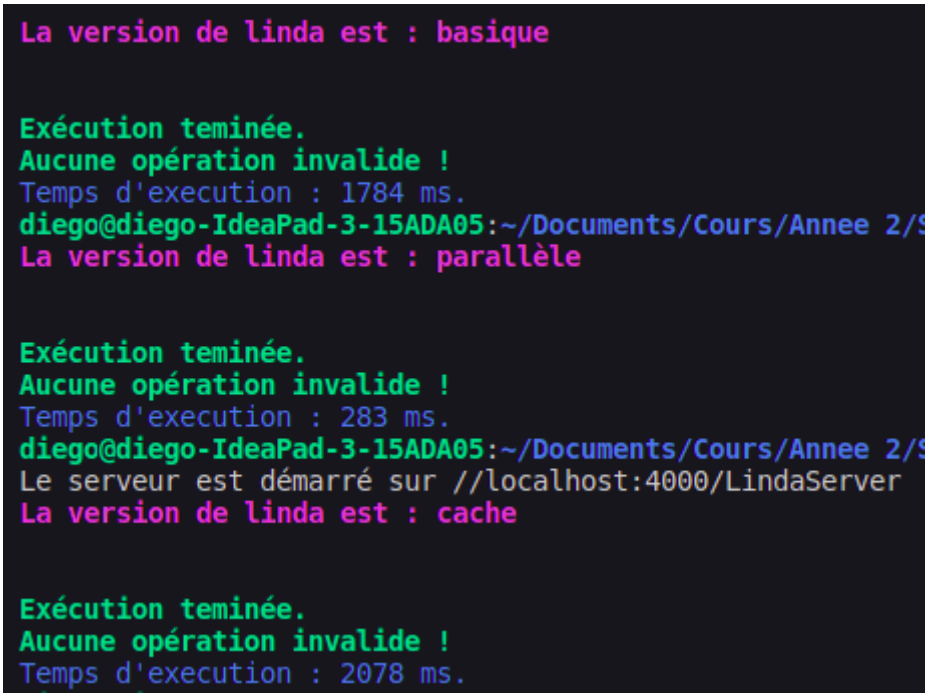

Figure 2 : Comparaison de performances : version avec plus de read

Puis, nous avons fait un test avec plus de take que de read. Encore une fois, la version parallèle a été plus efficace, suivie du cache.

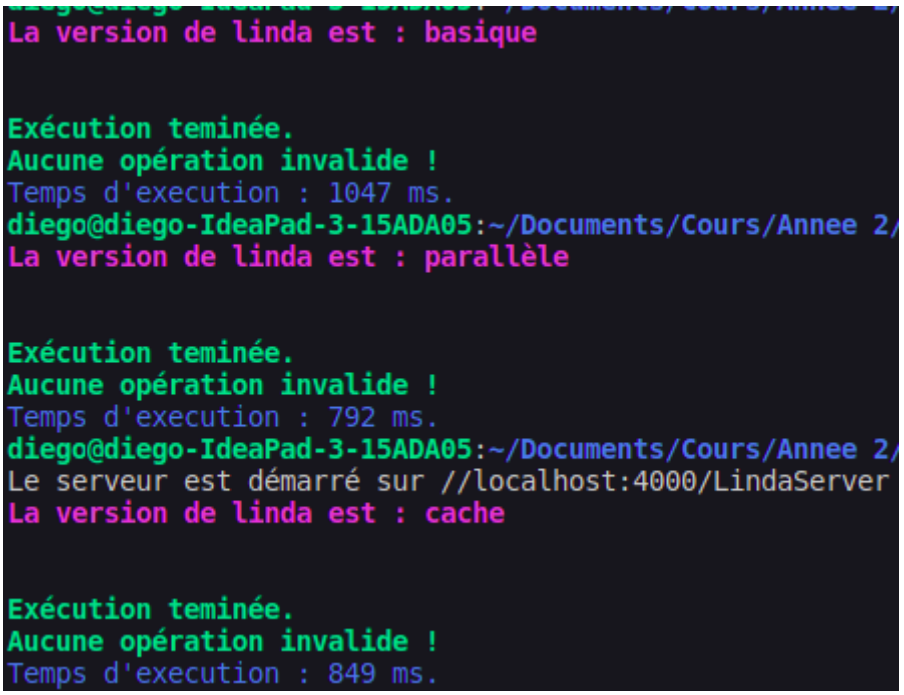

Figure 1 : Comparaison de performances : version avec plus de take

Enfin, nous avons créé de nombreux threads dans un même test. On voit alors que la gestion de tous ces threads entraîne un surcoût qui surpasse largement le gain de la parallélisation ou du cache.

Tests

Les tests de base n'ont pas été modifiés et ils ont été réutilisés pour la version parallélisée. De même, ces tests ont été réutilisés pour le cache.
Pour la partie multi-serveur, nous avons écrit des tests pour vérifier si les serveurs interagissent bien entre eux. Pour être plus précis, nous avons testé sur généralement 3 ou 4 serveurs avec plusieurs clients connectés à ses serveurs des write/read/take. Par la suite, nous avons testé les EventRegister. Pour tester, il faut auparavant démarrer les serveurs avec le fichier MultiServeur.java