Projet Données Réparties

Rapport final

Paul Anaclet

Guohao Dai

Théo Desprats

2A SN

2021

1. Introduction

Vous trouverez tout au long de ce rapport, la description des fonctionnalités, points et choix principaux induits de ce projet, et des problèmatiques qu'il soulève, sous forme de liste. Une annexe présentant quelques commandes pratiques est aussi disponible en fin de document.

2 Linda

2.1 Version mémoire partagée

- Stockage des tuples

Pour stocker les 'Tuple' écrits dans Linda, nous avons fait le choix d'utiliser une simple 'ArrayList<Tuple>' car suffisante et adaptée au problème. L'ajout d'un tuple dans l'espace partagé ne s'effectue que si aucun callback 'TAKE' n'a été appelé.

- Gestion des callbacks

Afin de pouvoir rappeler les différents 'Callback' enregistrés par 'eventRegister', nous avons décidé de mettre en place deux 'HashMap<Tuple, ArrayList<Callback>>' : une pour les 'Callback' en mode 'READ' et une pour les callbacks en mode 'TAKE', qui associent à un template une liste de 'Callback' à appeler. Chaque fois qu'un nouveau tuple est écrit dans l'espace partagé, on récupère les potentielles listes de 'Callback' à réveiller en matchant le tuple en question avec les tuples des 'keySet' des deux 'HashMap'. En cas de correspondance, tous les callback 'READ' sont réveillés puis seul le plus ancien callback 'TAKE' est appelé.

- Maintien de la cohérence

Pour maintenir la cohérence de l'espace partagé, mise en danger par les écritures et lectures concurrentes, nous avons mis en place deux verrous 'ReentrantLock' pour protéger la liste des 'Tuple' et les 'HashMap' de callbacks. Ces verrous sont utilisés pour créer des exclusions mutuelles dès qu'un accès à l'une des collections à lieu dans une méthode.

- Primitives bloquantes

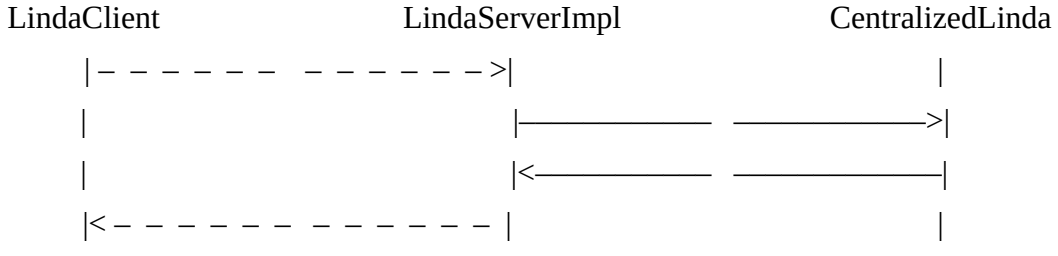
Les méthodes 'read' et 'take' se devant d'être bloquantes pour l'utilisateur en cas de non correspondance entre le tuple recherché et ceux dans Linda, nous avons mis en place une synchronisation en créeant un nouveau type de callback 'SynchronousCallback' utilisé en interne par Linda. Le fonctionnement est le suivant : par exemple, lors d'un 'take' (ou d'un 'write'), si le tuple recherché n'est pas présent dans l'espace partagé, alors un 'SynchronousCallback' est instancié et 'eventRegister' est appelé avec pour paramètres le tuple voulu et ce nouveau callback. Ensuite, le bloquage se fait alors grâce à un bloc 'synchronized(synchronousCallback)' contenant l'instruction 'synchronousCallback.wait() ;' : nous attendons donc un signal sur cet objet pour être débloqués. Le signal en question sera émis lorsque le 'SynchronousCallback' est appelé, c'est à dire, lorsque sa méthode 'call', contenant l'instruction 'this.notify()', sera appelée suite à l'arrivée d'un nouveau tuple valide. On pourra donc continuer l'exécution en récupérant le tuple valide dans le callback.

2.2 Version client/mono-serveur

- Architecture de communication

Pour pouvoir mettre en place un échange correct entre clients et serveur Linda, nous avons les éléments suivants : une interface ‘LindaServer extends Remote’, une implémentation de cette interface ‘LindaServerImpl extends UnicastRemoteObject’ (qui utilise une version ‘CentralizedLinda’) et une implémentation de Linda nommée LindaClient permettant de propager les requêtes clients au serveur auquel il est connecté.

Les appels de procédure des clients traversent donc les « couches » suivantes (aller-retour) :



– – –>: distant
——> : local

Contrairement aux ‘Tuple’, les ‘Callback’ ne sont pas sérialisables et posent un problème pour la communication RMI Client/Serveur. Nous avons donc décidé de créer un nouveau type de callback distant appelé ‘RemoteCallback’ (qui naturellement étend ‘Remote’, mais pas ‘Callback’) pour qu’ils puissent être passés en paramètres des méthodes de ‘LindaClient’ (de ‘eventRegister’ notamment) et donc être exploitables aussi par le ‘LindaServer’. A la manière des ‘AsynchronousCallback’, les ‘RemoteCallback’ « encapsulent » un callback qu’ils réveilleront en même temps qu’ils ont été réveillés. Cependant, côté serveur, comme les ‘AsynchronousCallback’ ne prennent que des types ‘Callback’ en paramètre de leur constructeur, nous avons crée une classe équivalent acceptant les ‘RemoteCallback’ et nommée ‘ServerAsynchronousCallback’.

Au final, on aura l’encapsulation suivante (permettant le réveil en cascade des callbacks) au niveau du ‘CentralizedLinda’ du serveur, pour un ‘eventRegister’:

ServerAsynchronousCallback(RemoteCallback(ClientCallback()))

- Serveur

Notre implémentation de l’interface ‘LindaServer’ utilise simplement les primitives d’un ‘CentralizedLinda’ local, stocké en tant qu’attribut privé. Les seuls réels ajouts consistent en la possibilité de le démarrer en tant que serveur RMI grâce au ‘main’ et l’encapsulation du ‘remoteCallback’ dans ‘eventRegister’ comme décrit précédemment.

- Client

‘LindaClient’ se connecte à l’URL qu’on lui donne puis propage les méthodes sur un objet ‘LindaServer’. L’enregistrement via ‘eventRegister’ crée un nouveau thread pour éviter que le client ne soit bloqué.

3 Applications

3.1 Crible d'Eratosthène

- Version séquentielle

Nous avons implémenté la méthode du crible d'Eratosthène en utilisant Linda de la manière suivante : nous écrivons tous les entiers inférieurs à 'n' sous forme de 'Tuple(valeur)' dans Linda, puis, pour 'i' de 2 jusqu'à 'n', si 'Tuple(i)' est présent dans Linda, alors, nous supprimons ses multiples de Linda. Ne resterons dans l'espace partagé que les tuples non-supprimés, c'est à dire, les nombres premiers recherchés.

- Versions parallèles

Trois versions parallélisées de l'algorithme sont disponibles dans l'archive.

La version 'write' propose une parallélisation de l'écriture initiale des 'Tuple(valeur)' dans Linda. Un nombre fixe de threads 'Writer' (défini au lancement) se partageront, dans la mesure du possible, des partitions égales d'écriture des naturels de 2 à 'n'. Le thread principal attendra leur terminaison grâce à plusieurs '.join()' sur un tableau stockant ses threads fils et passera à la suite du traitement.

La version 'take' crée de manière similaire des threads 'Taker' qui se partageront équitablement la suppression des multiples d'une valeur dans Linda.

La version 'takewrite' est une simple combinaison des deux versions précédentes : le remplissage de Linda et la suppression des multiples sont parallélisées.

- Performances

Temps d'exécution pour la recherche de nombres premiers < 400 et 4 threads

	Non parallèle	Writers parallèles	Takers parallèles	Takewrite parallèle
Temps d'exécution	38-49 ms	45-68 ms	82-208 ms	133-187 ms

Temps d'exécution pour la recherche de nombres premiers < 4000 et 4 threads

	Non parallèle	Writers parallèles	Takers parallèles	Takewrite parallèle
Temps d'exécution	373-696 ms	299-889 ms	730-1100 ms	583-1063 ms

3.2 Recherche de mot dans un fichier (version parallèle)

- Main

Pour l'approche parallèle de cette application nous avons modifié la classe 'Main' pour qu'elle serve de lanceur. Dans un premier temps, le 'Main' démarre un nombre fixe de 'Searcher' (décrits plus bas) donné en entrée par l'utilisateur. Ensuite, le script lit un fichier contenant une requête par ligne ('<mot> <fichier>') et instancie 1 thread 'Manager' (décrits plus bas) pour chacune de ces requêtes. Tous les threads instanciés sont alors connectés au même serveur Linda, les 'Searcher' s'abonnant en 'TAKE' aux tuples 'Request', et les 'Manager' postant leur requête respective.

- Manager

Les ‘Manager’ sont des threads qui déposent chacun une requête dans Linda et attendent le résultat. L’utilisateur peut instancier un thread ‘Manager’ en exécutant le ‘main’ de sa classe et en fournissant le mot, le fichier de recherche et l’URL du serveur Linda. Dans cette version, les ‘Manager’ chargent les données sous la forme de ‘Tuple(Code.Value, <mot>, **<reqUUID>**)’ pour que les mots soient associés à la requête déposée et pour que les ‘Searcher’ ne consomment que les mots concernés par celle-ci. Nous avons aussi ajouté la taille en octets du fichier de recherche aux tuples formant les requêtes pour que les ‘Searcher’ puissent créer leurs fils de manière proportionnelle, comme expliqué ci-dessous.

- Searcher

Les ‘Searcher’ sont des threads instanciables dynamiquement par l’utilisateur et s’occupent d’effectuer la recherche dans Linda. Un ‘Searcher’ peut instancier lui même des ‘Searcher’ fils seulement si son attribut ‘isStarter’ est positionné à ‘true’. Les ‘Searcher’ starter sont donc des threads qui s’abonnent en ‘TAKE’ (1 starter traite 1 requête) aux requêtes et génèrent 1 ‘Searcher’ fils (‘isStarter = false’) par tranche de 1 000 000 octets du poids du fichier de recherche (récupéré dans le tuple requête). Les pères participent à la recherche puis attendent la terminaison de leurs fils. Les fils générés n’effectuent donc que la recherche dans Linda (consommation des mots et envois de résultats) puis meurent. Une fois l’attente terminée, le ‘Searcher’ starter rend le tuple requête dans Linda puis dépose le tuple signalant la fin de la recherche, débloquant ainsi le ‘Manager’ en attente de finition.

Pour la version avec les interruptions, les pères vont s’abonner en ’TAKE’ à un Tuple (Code.Interupt). Quand un père reçoit un tel signal, il va envoyer un tuple (Code.Interupt, UUID) pour que tous ses fils s’arrête puis il va s’interrompre aussi. Cependant, cette version a besoin d’améliorations. En effet, un des problèmes et qu’avec un signal d’interruption, le searcher interrompt bien sa recherche et n’en fera pas d’autres, mais le processus ne meurt pas pour autant.

- Performances

Recherche du mot ‘agneau’ dans le dictionnaire français

	Version basic	Version parallèle (4 searcher)
Temps d’exécution	619-918 ms	2368-4087 ms

On remarque une baisse des performances sûrement due à un surplus non-nécessaire de créations de threads pour un problème trop petit. Cette baisse peut aussi s’expliquer par l’utilisation d’un serveur Linda dans la version parallèle plutôt que d’un Linda local.

4 Analyse

4.1 Difficultés

- Problème de synchronisation Take/Read si Tuple déjà présent dans ‘CentralizedLinda’

Rendre les primitives ‘take’ et ‘read’ bloquantes en ayant un résultat cohérent nous a posé quelques soucis : nous avons mis du temps avant de comprendre que notre erreur venait du fait qu’on ne vérifiait pas si le tuple était déjà présent avant d’enregistrer un callback interne. S’il était déjà présent dans Linda, ‘eventRegister’ réveillait le callback interne avant même qu’il se mette en attente : on avait donc un bloquage de la primitive.

- Possibilité d’utiliser des ExecutorService

Nous nous sommes rendus compte trop tard que l’utilisation des ‘Executor’ Java auraient pu nous simplifier certains aspects de programmation. La mise en place du timeout de 5 secondes sur les ‘Manager’ aurait pu être implemantée par le biais de tâches ‘Callable’ et de la méthode ‘Future.get(long, TimeUnit)’.

- Quantité de tests insuffisante

Malheuresement nous n’avons pas réussi à produire suffisamment de jeux de tests pour pouvoir couvrir toutes les classes d’équivalences, nous nous sommes plutôt basés sur les résultats obtenus lors du développement des différentes versions des applications.

- Take du Manager sur ‘done’ effectué avant ‘call’ des résultats

Dans l’application parallélilsée ‘Search’ persiste un problème concernant la synchronisation ‘Searcher’/’Manager’. Notre implémentation semble trouver des résultats finaux cohérents mais quand le ‘Manager’ récupère le tuple ‘done’, il n’a pas fini de recevoir tous les appels des tuples ‘Result’ sur son callback. Au final, le ‘done’ est correctement écrit (après terminaison des ‘Searcher’ fils) et reçu, mais les appels sur le callback du ‘Manager’ arrivent après, alors qu’il ne les considère plus comme nouveaux résultats.

4.2 Conclusion

Au final, malgré quelques fonctionnalités manquantes et optimisations possibles nous sommes assez satisfaits de notre production. Nous aurions aimé avoir eu le temps d’implémenter la totalité des attendus, de rédiger plus de tests et d’obtenir des meilleures performances car ce projet était plaisant et nous aura permis d’approfondir nos connaissances techniques, notamment en Java.

ANNEXE 1
Compilation et exécution

Makefile

Le ‘Makefile’ à la racine du projet permet de simplifier plusieurs actions :

- ’make build’ : compilation de tout le projet à l’aide de JavaC
- ‘make clean’ : supprime les ‘.class’ générés à la compilation
- ‘make server’ : démarrage de Linda en mode serveur RMI
- ‘make tests’ : exécution de l’ensemble des tests contenus dans ‘linda/test’
- ‘make help’ : affiche l’ensemble des commandes disponibles

Application ‘Crible d’Eratosthène’

Exemple de démarrage pour recherche de nombres premiers inférieurs à 100 partagée entre 4 threads :

```
$java linda/primenumber/<NomVersion>/Main.java 100 4
```

Application ‘Recherche d’un mot dans un fichier’

Exemple de démarrage pour 5 ‘Searcher’ de base (version ‘parallel’):

```
$java linda/search/parallel/Main.java 5 linda/parallel/tests/requests1.txt
```

Exemple de création d’un ‘Searcher’ (version ‘parallel’) :

```
$java linda/search/parallel/Searcher.java //localhost:4000/LindaServer
```

Exemple de création d’un ‘Manager’ (version ‘parallel’) :

```
$java linda/search/parallel/Manager.java ‘agneau’ /usr/share/dict/french //localhost:4000/LindaServer
```

Application ‘Recherche d’un mot dans un fichier’ avec interruption

Même instructions que pour la version sans interruption

Exemple de création d’une interruption :

```
$java linda/search/parallelinteruption/Interupt.java //localhost:4000/LindaServer
```