Projet Données Réparties

Rapport

Enzo PETIT Nam VU

16 janvier 2022
ENSEEIHT – 2SN-A

Table des matières

1 Introduction

Le projet Linda a pour but de réaliser un espace partagé de données typées.

Une version à mémoire partagée avec gestion de callback a été réalisée ainsi qu'une version client-serveur en RMI se reposant sur cette dernière.

Pour la première version, la classe `linda.shm.CentralizedLinda` a ainsi été complétée et des tests unitaires sous JUnit 5 ont été rédigés dans `linda.test.CentralizedLindaTest`.

La version client-serveur se trouve quand à elle dans le package `linda.server` et utilise la version *shm* inchangée pour la partie serveur.

Enfin quelques applications utilisant Linda ont été réalisées telle que la recherche de nombre premiers selon l'algorithme du crible d'Erathosthène, en séquentiel et parallèle, ainsi que la parallèlisation de l'application de recherche.

2 Version en mémoire partagée

Cette section reprend le rapport provisoire de décembre dernier.

2.1 Réalisation

A l'instanciation, `CentralizedLinda` initialise trois tableaux pour le stockage des tuples (`tupleSpace`) et les events *take* (`takeEvents`) et *read* (`readEvents`).

Ces tableaux sont de type `CopyOnWriteArrayList` qui est une variante *thread-safe* de l'`ArrayList` classique adaptée à un contexte concurrent où le nombre de lectures est bien supérieure au nombre d'écritures.

Suivent après les détails d'implémentation des différentes opérations, plus ou moins dans l'ordre de réalisation :

2.1.1 `tryTake`, `tryRead`

Ces deux méthodes sont non bloquantes, on itère simplement sur la liste (en partant de la tête) et on renvoie le premier tuple (le plus vieux) qui match le template. `null` est renvoyé si aucun tuple actuellement stocké ne correspond.

2.1.2 `takeAll`, `readAll`

Même chose que précedemment mais on stocke tous les tuples correspondants dans une `ArrayList` que l'on renvoie à la fin (qui est vide si aucun résultat).

2.1.3 `eventRegister`

En commençant à vouloir implémenter les `take` et `read` bloquant on s'est demandé comment pouvait-on *proprement* et avec le moins d'effort possible bloquer et débloquer les appels : le principe des event nous a paru bien adapté pour réaliser cette tâche (détails plus loin).

En mode `IMMEDIATE` un tuple est retourné immédiatement dans le callback en cas de match sur l'espace actuel (via `tryTake`/`tryRead`), sinon on range l'event en attente dans le tableau correspondant (`takeEvents` ou `readEvents`).

Le callback est transformé en amont en `AsynchronousCallback` afin d'éviter les problèmes liés au blocage du thread principal ou d'enregistrements récursifs de callbacks.

2.1.4 `write`

La méthode `write` étant la *porte d'entrée* de tous les tuples vers l'espace de stockage de Linda, c'est là qu'on en profite pour *résoudre* les event en attente le cas échéant.

Ainsi on itère d'abord sur les *read* en attente (`readEvents`), vérifie si le tuple à écrire *match* le template de l'event et le cas échéant on appelle le callback correspondant.

Ensuite on fait de même avec les *take* en attente (`takeEvents`) mais au premier match (du plus vieux), on résout le callback et on retourne, immédiatement. Le tuple n'est pas enregistré et les *take* en attente dessus mais plus récents attendront le prochain tuple correspondant.

Finalement si aucun *take* n'attendait le tuple, on le sauvegarde dans `tupleSpace`.

Un tuple en entrée peut ainsi résoudre tous les *read* en attente mais qu'un seul *take* en attente, le plus vieux.

2.1.5 `take`, `read`

Un *take* ou *read* bloquant revient à enregistrer un event *immédiat* dont le callback renvoie le tuple passé en entrée, rester bloqué jusqu'à résolution de celui-ci et finalement renvoyer son résultat.

On utilise pour faire ça une `LinkedBlockingQueue`, queue bloquante : le callback de l'event correspond à la méthode `offer` de la queue (dépôt non bloquant) qui sera éventuellement appelée lors d'un *write*.

Le `take`/`read` reste lui bloqué sur le `take` de la queue et renverra son résultat quand il sera débloqué par un dépôt dans la queue.

2.2 Tests

Tous les tests `Basic` fournis passent en l'état.

Une classe de tests unitaires JUnit 5 `linda.test.CentralizedLindaTest` a aussi été écrite.

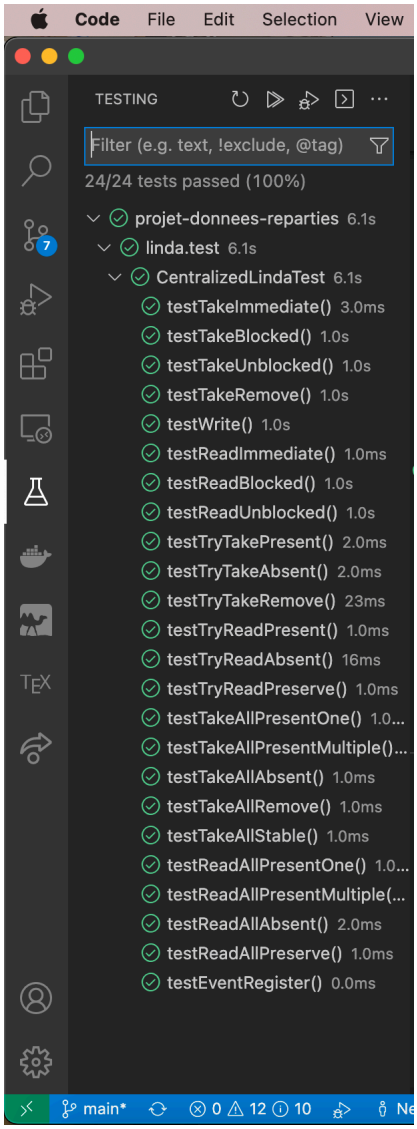

FIGURE 1 – Résultats des tests définis dans `linda.test.CentralizedLindaTest` (Visual Studio Code + Extension Pack for Java)

3 Version client / mono-serveur

Package ***`linda.server`***

3.1 Serveur

Une interface `LindaServer` a été définie, étandant `Remote`, et reprend les méthodes définies dans l'interface `Linda` à la différence que toutes les méthodes `throws RemoteException` et que `eventRegister` prend en paramètre un `eventRegister`.

`eventRegister` est une interface qui étand aussi `Remote` et représente un callback à exécuter sur le client.

L'implémentation de `LindaServer` est rédigée dans `LindaServerImpl`. Cette classe initialise un noyau Linda en mémoire partagée et transmets directement tous les appels distants au noyau à l'exception du `eventRegister` : pour cette méthode, on transmet au kernel un callback local qui résoudra le callback distant. Ainsi tout est transparant pour le noyau en mémoire partagée.

3.2 Client

De la même manière `LindaClient` récupère un `LindaServer` en RMI puis fonctionne de manière transparente pour les appplications en local en ne faisant que transferer directement toutes les requêtes au serveur distant.

Exception est encore faite pour `eventRegister` qui est un peu plus complexe : on envoie au serveur un `RemoteCallback` via `RemoteCallbackAdapter` qui s'occupera de répondre au callback local côté client.

4 Applications

4.1 Calcul des nombres premiers

Package ***`linda.primes`***

4.1.1 Séquentiel

La classe `Sequential` implémente une version sequentielle basique du crible d'Erathosthène en utilisant Linda. On parcours les nombres de 2 à n : si il n'est pas présent dans Linda (par `tryRead`) alors le nombre est premier et on rajoute tous ses multiples dans Linda.

4.1.2 Parallèle

On fournit dans `Parallel` une tentative de parallèlisation de cet algorithme peu parallèlisable.

On commence par enregistrer dans Linda tous les nombres en les marquant premiers (initialisation du crible).

Autant de `Worker`s que de nombres sont ensuite créés : ils s'occupent de retirer leurs multiples de Linda et abandonnent leur tâche si leur nombre de départ s'avère non premier (retiré par un autre `Worker`). Tous ces `Worker`s sont exécutés dans des threads grâce à un `ExecutorService` initialisé par `Executors.newWorkStealingPool`.

Enfin, dans le thread principal on se bloque jusqu'à l'éxecution complète de tous les `Worker`s.

`Résultats`

En testant la recherche jusqu'à 10000 il est clair que la parallèlisation du crible d'Erathosthène est peu efficace. On obtient un temps de traitement triple par rapport à la

version séquentielle qui s'explique par l'overhead généré par la création de threads et la répétitions d'opérations : ne pouvant savoir si un nombre et premier dès le départ, on commence malgré tout à marquer ses multiples. Aussi notre implémentation de Linda fait que les lectures sont peu efficaces car on itère sur toute la liste à chaque accès.

4.2 Recherche approximative dans un fichier

Package `linda.search.evolution`

Cette application recherche dans un fichier de mots celui qui est le plus *proche* du mot donné en entrée.

Une base de l'application était déjà faite, mais celle-ci devait être étendue avec d'autres fonctionnalités.

Pour chaque nouvelle fonctionnalité, une explication de l'implémentation se trouve dans ce qui suit.

4.2.1 Simultanéité des chercheurs

On voulait permettre à l'application d'autoriser plusieurs chercheurs en simultané sur une même requête. Cela a été fait en modifiant plusieurs choses.

```
Ready to do a search
Ready to do a search
Search 95b64eab-41eb-47c8-a8c2-0f058a57c4a6 for abbey
Looking for: abbey
Looking for: abbey
New best (4): "a"
New best (4): "a"
New best (3): "aba"
New best (2): "abb"
New best (0): "abbey"
query done
PS E:\PROG\2021-2022\projet-donnees-reparties>
```

FIGURE 2 – Exemple d'exécution avec plusieurs chercheurs sur la même requête - 'abbey' dans un dictionnaire anglais

Tout d'abord, le tuple de requête contient un nouveau paramètre : le nombre de chercheurs ayant pris la recherche en charge. Le manager initialise ce paramètre à 0.

Ensuite, quand un chercheur trouve un tel tuple, il le prend, regarde si il reste de la place pour d'autre chercheurs sur cette recherche, et le redépose avec une valeur mise à jour dans l'espace le cas échéant. Dans tous les cas, le chercheur va ensuite travailler normalement, mais au lieu de notifier la fin de son travail, il notifie qu'il est toujours opérationnel.

Enfin, le manager va effectuer une attente semi-active pour savoir si il reste encore des chercheurs affectés à sa requête. Quand il n'y en a plus, on considère la recherche comme terminée.

Le choix de l'attente active a été faite dans le cas où le chercheur viendrait à *planter*, car la consigne spécifiait un *arrêt arbitraire de chercheurs*, que nous avons interprété en tant que tel. Dans ce cas, toute sorte de notification de fin de travail est impossible.

Si cela n'était pas ce qui était désiré, nous avons pris l'autre partie pour la rétraction d'un manager (voir la partie concernée ci-dessous).

FIGURE 3 – Exemple d'exécution avec arrêt prématuré des `Searchers` - Le `manager` détecte l'absence de chercheur restants et termine

4.2.2 Simultanéité des managers

L'on souhaitait ici avoir plusieurs managers déposant des requêtes différentes.

La simple solution pour permettre cela était de marquer les données avec l'UUID de la requête.

Il fallait également attendre la prise en charge de la recherche par un des chercheurs avant de considérer la fin de celle-ci.

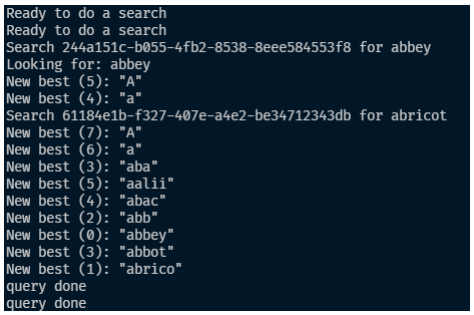

FIGURE 4 – Exemple d'exécution avec 2 `Searchers` et 2 `Managers` - 'abbey' (présent) et 'abricot' (absent)

4.2.3 Linda version serveur

Ici, rien n'est à modifier, le code marchait à la perfection en étant simplement transposé, même si les managers et les chercheurs sont dans des clients différents.

4.2.4 Retrait de managers

En contraste avec la vision prise pour les chercheurs, on considère ici que les managers se *retirent* moins brusquement.

Au lieu de faire une attente active du côté des chercheurs, on décide alors la présence dans l'espace linda d'une variable *de garde* que l'on retire quand on souhaite interrompre la recherche.

Pour permettre d'indiquer un temps d'attente maximum, l'on surcharge le constructeur avec un nouveau paramètre.

On peut alors déclarer un objet partagé et attendre un temps maximal à l'aide de la fonction `wait()`, et reprendre l'exécution plus tôt à l'aide de la fonction associée `notify()`.

De plus, si un chercheur est interrompu dans sa recherche, il est réinitialisé et attend une nouvelle requête.