Rapport Projet PDR

I Introduction

Le projet a pour but de réaliser un espace partagé de données. Cette approche est inspirée du modèle Linda. Dans ce modèle, les processus partagent un espace de tuples qu'ils peuvent manipuler. Cet espace à été implémenté de deux manières distinctes. La première solution est un **espace en mémoire partagée**, et la deuxième est un **espace client/serveur**, avec plusieurs clients pouvant accéder a l'espace de manière concurrente.

II Centralized linda sans callback

Sans callback, la méthode **Write** se contente de copier le tuple et de l'ajouter à l'espace de tuple. La méthode **Take** supprime le tuple de l'espace en en gardant une copie, La méthode **Read** copie un tuple commme **Take** mais sans le supprimer de l'espace de tuple. **TryTake** et **TryRead** sont des équivalents de **Take** et **Read** Non Bloquants, c'est à dire que si le **Tuple** n'est pas trouvé, on ne fait rien aulieu d'attendre qu'il soit présent. **readAll** et **TakeAll** sont des équivalents de**Read Take** mais s'appliquent juste sur tout l'espace de tuples.

II Centralized linda avec callback

Avec **callbacks**, seule la méthode **Write** nécéssite des modifications. **Pour la méthode Write**, avant d'ajouter un tuple à notre espace de tuple, la méthode **write** vérifie d'abord qu'il y a un tuple qui correspond au tuple en paramètre, si c'est le cas, on exécute le **callback** correspondant (exécution de la méthode **call** de l'interface **Callback**), avant de passer à la suite. Maintenant, si le mode du tuple correspondant est **Read**, aucun ajout dans l'espace de tuples n'est réalisé, et si c est **Take**, nous l'ajoutons.

III event register

Pour cette partie, on utilise **eventTiming**. **eventTiming** présente deux états distincts, à savoir **eventTiming.IMMEDIATE**, qui traduit la considération immédiate de l'état courant. Si un tuple correspondant existe, alors le **CallBack** sera déclanché.
Le deuxième état est **eventTiming.FUTURE**, qui précise que seul les futurs " **Write**" seront pris en compte.
Nous avons procédé comme suit en distinguant les deux cas précédemment définis : Si **EventTiming = IMMEDIAT**, alors nous appelons la méthodes call de l'interface Callback si le résultat retourné par Read ou Take est différent de nul, autrement dit, si Read ou Take renvoie un tuple. Si non, alors **EventTiming = FUTURE**, nous ajoutonsdonc un nouvel élément à la liste des **callbacks** selon les autres paramètres de la méthode **EventRegister**.

IV Implémentation client/serveur de Linda.

Nous avons ajouté une interface **LindaInterfaceRMI** qui est **hérité** de la classe **Remote** pour pouvoir appeler ses méthodes depuis le serveur. Le serveur doit bien évidemment implémenter cette interface pour pouvoir être appelé à distance.

IV/ A. Partie Serveur

La classe serveur nécessite un **comportement à distance**, elle est donc **héritée** de **UnicastRemoteObject** et **implémente** l'interface qui étend **Remote** pour pouvoir être appelée à distance. Il est important de remarquer que que toutes les méthodes peuvent renvoyer une **RemoteException**. Le constructeur permet d'instancier un **CentralizedLind**a. Ensuite, nous avons créé l'objet **Linda** (Lindaserveur) et nous l'avons inséré dans le registre. A ce niveau-là, le serveur

est prêt. Nous avons Override les **méthodes** de l'interface qui étend **Remote**, en appelant simplement les méthodes de **CentralizedLinda**. En ce qui concerne l' **eventRegister**, le callback en paramètre doit pouvoir appeler à distance le callback sur la méthode **write**, d'où la nécessité de la classe **CallbackImplForServer** qui implémente l'interface **Callback** et transforme le callback en callback appelable à distance.

Nous avons ajouté une interface **LindaInterfaceRMI** qui étend la classe Remote pour pouvoir appeler ses méthodes depuis le serveur. Le serveur doit bien évidemment implémenter cette interface pour pouvoir être appelé à distance.

IV/ B. Partie Client

Cette classe implémente l'interface **Linda** et propage tout au serveur auquel le client est connecté c'est-à-dire : //localhost :4000/LindaServer Le constructeur permet de se connecter au bon serveur via l'url et le registre. Ensuite, on fait appel à toutes les méthodes de l'interface Linda. En ce qui concerne l' **eventRegister**, le callback en paramètre doit pouvoir appeler à distance l'interface Callback, c'est pour cela que nous avons créé **CallbackInterfaceRMI** qui étend la classe **Remote** pour pouvoir invoquer call depuis le client Linda. La classe **CallbackImplForClient** nécessite un comportement à distance d'où l'héritage depuis **UnicastRemoteObject** et implémente l'interface **CallbackInterfaceRMI** qui est héritée de **Remote** pour pouvoir être appelé à distance.

V Applications

V/ A. Crible d'Eratosthène

Concernant le Crible, **CribleSequentiel** correspond a l'algorithme classique, pas de parallélisation.

CriblePoolFixe : La version **PoolFixe** exécute l'algorithme standard avec un nombre de **threads** fixes, ici deux.

CribleParallele1 : La version **Parallèle1** crée un thread pour chaque nombre que entre 1 et **MAX**, ce qui consomme beaucoup de ressources inutilement, mais permet un bon niveau de parallélisation.

CribleExecutorSubmit : La version **ExecutorSubmit** constitue une version qui implémente une Exécutor, avec nombre d'activités adaptables (**newCachedThreadPool()**), cette méthode bloque en attente du résultat

CribleExecutorExec : La version **ExecutorExec** constitue une version qui implémente une Exécutor, avec nombre d'activités adaptables (**newCachedThreadPool()**),On paralèllise la recherche des multiples des nombres premiers donc le nombre de thread lancés dépend des nombres premiers présents.

Comparaison du temps d'exécution :

Version du Crible	Séquentiel	Poolfixe	Parralele1	Executorsubmit	Excecutorexec
Temps d'exécution (en ms pour n= 100000)	3844	3001	2649	2904	2726

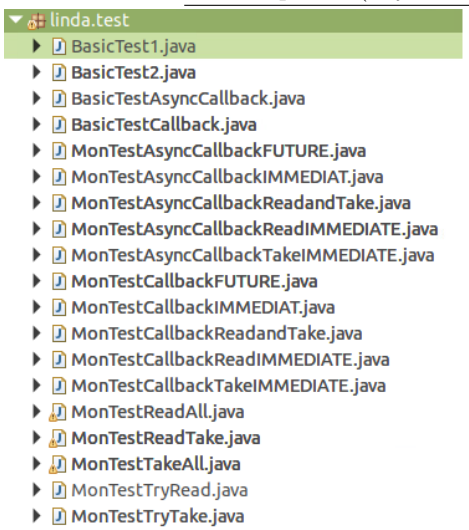
Sans suprise, la version qui est la plus consomatrice de ressources est la plus rapide, et la version séquetielle est la plus lente.

V/ B. Recherche approximative dans un fichier

Dans cette partie, il s'agissait de faire de la recherche de mot par distance de **Levenshtein**. La distance de **Levenshtein** donne une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre. On peut faire calculer cette distance a plusieurs **Threads** en parallèle, on les appelle les **chercheurs**. Les données a explorer sont gérer par un **Thread**, qu'on appelle **Manager**. Par un soucis d'économie de ressources, on limitera le nombre maximum de **Chercheurs** à 10.Cette application est fonctionnelle, cependant, Les parties suivantes n'ont pas été traitées : **le démarrage dynamique et arbitraire de chercheurs**, l'**arrêt arbitraire de chercheurs**, le **retrait de la recherche par le manager après un certain délai**.

VI tests

Concernant les tests, nous en avons ajouté pour vérifier et appuyer le bon fonctionnement du projet. Tous nos tests sont commentés et précisent à chaque étape le résultat attendu. Nous avons ajouté un test avec **Read** et **Take**, un test avec **ReadAll**, un avec **TakeAll**, et deux avec les **Take** et **Read** non bloquants (TryRead, TryWrite).



Passant maintenant aux callbacks, nous avons testé **IMMEDIAT** ainsi que **FUTURE**. Un test avec deux **eventRegister** Read et Take en meme temps. **Read** et **Take** ont aussi été testés de manière indépendante. Les derniers tests cités ont été fait sur des **callbacks Asynchrones** également.

VII Conclusion

Pour conclure, dans l'ensemble, le sujet à été compris et traité, il nous aurait cependant fallu un peu plus de temps pour implenter les dernières fonctionnalités de **Recherche approximative dans un fichier**, qui ne présentaient pas de difficultés particulières.