Rapport Projet
Linda

ARNOLD Cyprien

BARROSO Pierre
DUONG Tom

Département Sciences du Numérique - 2ème année - 2021-2022

Table des matières

1 Introduction

Le projet a pour but de réaliser un espace partagé de données typées. Cette approche est inspirée du modèle Linda (ou TSpaces). Dans ce modèle, les processus partagent un espace de tuples qu'ils peuvent manipuler à l'aide d'un jeu de primitives spécifiques.
Les tuples sont des n-uplets pouvant contenir des valeurs quelconques ou des types. On parle alors de motif.
Le modèle linda correspondant à l'interface *Linda* était accompagné de méthodes à implémenter.
Nous avons implémenté de deux manières ce modèle : une version en mémoire partagée et une version client/mono-serveur.
Il fallait enfin tester nos implémentations de linda par le biais de quelques applications. Nous avons réalisé une version du crible d'Eratosthène en version parallélisée puis un algorithme de recherche approximative dans un fichier.

2 Architecture

Un des enjeux de ce projet était de réaliser deux implémentations différentes de l'interface **Linda**, qui pouvaient être toutes deux utilisées sans avoir à apporter de modifications au code les appelant.
Nous avons commencé par réaliser la version en mémoire partagée : **CentralizedLinda**. Cette classe implémente directement l'interface **Linda**, tout comme **LindaClient** qui correspond au client dans la version client / serveur. En revanche, **LindaClient** hérite de la classe **UnicastRemoteObject** car elle transmet des objets **LindaClient** au serveur.
Nous avons par ailleurs dû créer une nouvelle interface **LindaReparti** fortement similaire à l'interface Linda mais apportant tout de même quelques modifications. Comme cette interface hérite de l'interface **Remote**, toutes ses méthodes doivent comprendre un *throws RemoteException* dans leur signature. On ne pouvait donc pas simplement hériter de l'interface **Linda** car les signatures n'auraient alors plus été correctes. Il n'était par ailleurs pas non plus possible d'utiliser un **Callback** comme dans la version en mémoire partagée. Nous avions ici besoin d'un **Callback** qui héritait de l'interface **Remote** afin de pouvoir utiliser sa méthode *call*. Nous nous sommes donc basés sur l'interface **Callback** à laquelle nous sommes venus ajouter la mention *throws RemoteException* à la signature de son unique méthode *call* pour créer la nouvelle interface **CallbackRemote**. Des objets de type **CallbackRemote** sont utilisés dans la méthode *eventRegister* de l'interface **LindaReparti**.
La classe **LindaServer** représente le serveur dans la version client / serveur. Son code est identique à celui de la version **CentralizedLinda** à l'exception près qu'on vient utiliser des **CallbackRemote** et non plus des **Callback** "classiques". On utilise deux *Map* prenant pour clé un template et pour valeur une *List* de **Callback** : une pour ceux en lecture et une pour ceux en écriture. On utilise également une *List* permettant de stocker les **Tuples**. Ces trois collections sont synchronisées, ce qui permet d'éviter des accès concurrents sur leur contenu. On utilise également un sémaphore pour bloquer les méthodes *read* et *take* afin d'éviter qu'elles monopolisent les ressources partagées.
La classe **LindaClient** implémente l'interface **Linda** pour pouvoir être appelée comme **CentralizedLinda** sans avoir de modification de code à apporter. Elle implémente également l'interface **CallbackRemote** et hérite de la classe **UnicastRemoteObject** car sa méthode *eventRegister* vient transmettre *this* au serveur. Elle a pour attributs un **LindaReparti** et un **Callback**, correspondant respectivement au serveur auquel elle est connectée et au **Callback** appelé par le serveur. Le serveur ne peut pas en réalité appeler directement ce **Callback** car il n'est pas de type **Remote** : c'est pourquoi on transmet plutôt un **LindaClient** au serveur dans lequel on a enregistré le **Callback** d'origine. Lorsque le serveur appelle la méthode *call*, il appelle donc celle d'un **LindaClient** qui vient finalement appeler la méthode *call* de son attribut *cb*.

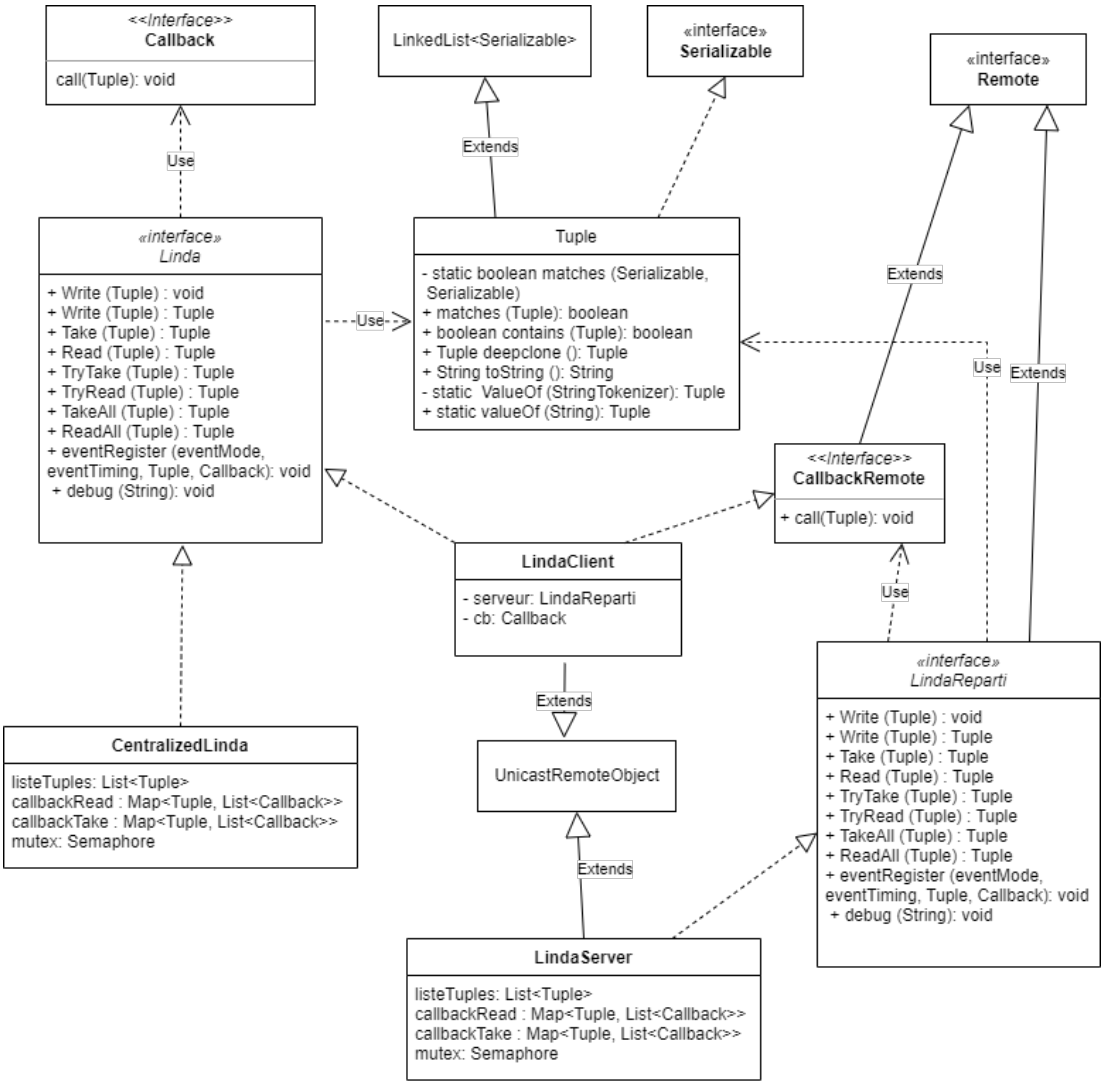

FIGURE 1 – Diagramme de classe Linda

3 Algorithmes utilisés et points délicats

Tout d'abord, afin de rendre les méthodes **Read** et **Take** bloquantes, nous avons utilisé un **mutex**. La méthode **Write** génère un jeton pour le **mutex** si une ou plusieurs méthodes sont en attente sur ce **mutex**. Le fonctionnement est similaire pour les méthodes **Read** et **Take**. Si la lecture/prise est accomplie, et qu'une méthode est en attente de libération du **mutex**, celui-ci est rendu, afin que cette dernière puisse être réveillée. Sinon, comme pour **Write**, si aucune méthode est en attente sur le **mutex**, celui-ci n'est pas rendu. Cette idée d'implantation est idéale pour deux choses : d'une part, on conserve le fonctionnement théorique d'un **mutex** (son nombre de jeton ne peut pas dépasser 1). Ensuite, cela évite qu'une méthode qui doit bloquer car l'accès à un Tuple n'a pas pu être effectué, de ne pas l'être dès la première itération car le **mutex** possédait un jeton ou plus, et boucle plusieurs fois avant de bloquer, ce qui évite une perte de temps.
Pour gérer l'exclusion mutuelle, nous avons utilisé des listes et hashmaps **synchronisées**. En effet, la liste de Tuples et les Hashmaps de Callbacks sont ainsi accessibles par une unique méthode à la fois, évitant les conflits.
Concernant justement les callbacks, leur gestion est principalement réalisée dans la méthode **Write** et bien evidemment dans la méthode eventRegister. Nous avons décidé de créer deux hashmaps **CallbackRead** et **CallbackTake**, dont les clés sont des Tuples (template), et les valeurs sont des listes de Callbacks. La méthode **Write**, après avoir écrit le Tuple passé en argument dans la liste des Tuples, recherche une correspondance parmi les clés des deux hashmaps, afin de reveiller tous les clients présents dans la liste des callbacks associée à ce Tuple. Nous avons décidé de donner la priorité aux clients abonnés pour un **Read**, afin qu'ils puissent tous lire le Tuple qu'ils recherchent avant que les client abonnés pour une prise de Tuple ne puissent le faire. Concernant justement les prises, la méthode **Write** ne reveille les clients qu'un par un, jusqu'à ce que le Tuple recherché ne soit plus disponible, ce qui évite de reveiller puis de réenregister un client dont la demande n'a pas pu être satisfaite dans l'**eventRegister** pour rien. Quant à **eventRegister**, il enregistre donc dans l'une des hashmaps, selon le paramètre **eventMode**, le nouveau client abonné. Cette méthode effectue également une tentative d'écriture/lecture si le paramètre **eventTiming** est **IMMEDIATE**.

4 Application à des exemples concrets

4.1 Calcul des nombres premiers inférieurs à K

Pour le calcul des nombres premiers inférieurs à une certaine borne K, nous nous sommes basés sur l'algorithme du crible d'Eratosthène et nous avons créé une interface **Eratosthene** contenant une méthode *getPremiers*. Nous avons dans un premier temps réalisé une implémentation séquentielle de cet algorithme, puis nous avons essayé de mettre en place une version parallèle utilisant Linda.
Le principe de cet algorithme est de créer un tableau de booléens où chaque case correspond à un entier. On va venir dans un premier temps mettre à jour ce tableau en bouclant sur les entiers tant que leur carré est inférieur à la borne K, et modifier la valeur des cases correspondantes aux multiples de cet entier. Une fois qu'on a obtenu la version finale du tableau, on vient le parcourir entièrement pour récupérer les entiers non modifiés, c'est-à-dire les nombres premiers.
Comme cet algorithme dépend à chaque itération des résultats de la précédente, il n'était pas possible de le paralléliser à ce niveau là. L'idée a alors été de découper le parcours final du tableau en sous parties. On vient alors créer des Threads qui vont chacun parcourir une partie du tableau, enregistrer les nombres premiers dans un Tuple et le transmettre au Linda (centralisé ou serveur). On vient finalement faire un *readAll* avec un motif correspondant aux Tuples précédemment enregistrés et on concatène toutes les listes en une seule avant de la renvoyer.
La classe **Comparaison** permet d'évaluer la différence de temps d'exécution entre la version séquentielle et la version parallèle. On obtient dans ce cas là de bien meilleurs résultats avec la première méthode mais on pouvait s'y attendre car c'est un algorithme non adapté à la parallélisation. Les temps de lancement des Threads, d'écriture dans des Tuples, de transmission au Linda puis de récupération font que cette version ne propose pas de bonnes performances. On aurait pu améliorer très légèrement ce temps d'exécution en utilisant un Callback concaténant les listes de nombres premiers au fur et à mesure de leur écriture dans le Linda avant de se ré-inscrire lui-même mais ça n'aurait pas permis de venir concurrencer la version séquentielle tout en compliquant la lisibilité du code.
Cependant, l'écart de performances entre les deux algorithmes tend à se réduire plus on augmente la taille de la borne car plus tous ces temps deviennent négligeables devant le temps total de l'exécution. Nous avons ici pris simplement des entiers et non des doubles et l'avantage reste toujours à la version séquentielle à la limite.

4.2 Recherche approximative dans un fichier

Nous avons rajouté 3 fonctionnalités :
— Le fait de pouvoir utiliser linda client/serveur
— Le fait de pouvoir avoir plusieurs Searcher
— Le fait d'avoir plusieurs Manager et donc plusieurs queries ainsi mots à analyser

Pour cela, nous lançons n **Manager** et m **Searcher**. Les **Searcher** s'occupent d'une *query* à la fois. Une fois qu'un des **Searcher** a trouvé le mot de distance 0, il envoie un Tuple particulier au Manager correspondant qui retire sa *query* et écrit que sa *query* est terminée.
Tous les **Searcher** regardent tous les i mots si la *query* est terminée, si elle l'est, ils attendent la *query* suivante. Cela permet qu'une fois un mot trouvé par un **Searcher**, les autres arrêtent de chercher.
Le **Manager** suivant est en *read* sur le tuple annonçant la fin de la *query* précédente. Une fois reçu, il va attendre un petit peu avec un *sleep* (pour que les **Searcher** aient le temps de savoir que la *query* précédente est terminée et donc se mettre en attente de la prochaine) et *write* le **Linda** avec les mots du dictionnaire et faire sa *query* et ainsi de suite.

5 Conclusion

Ce projet nous a permis de mettre en application les différents concepts vus en cours de systèmes concurrents et d'intergiciels. Il était intéressant de mettre en relation ces 2 matières car cela permet d'avoir une vision plus globale de ce que peut être un programme complexe concurrent.
Les applications nous ont aussi permis de visualiser à quoi peut servir le modèle Linda. Bien que complexe, la recherche d'un mot dans un fichier nous a montré l'utilité de Linda par sa possibilité de mélanger plein de types d'informations différentes dans un même espace et de pouvoir récupérer ceux qui nous intéressent par des templates. Cependant, il est difficile d'avoir un grand nombre de Threads utilisant Linda car les méthodes de Linda prennent un certain temps, et assez rapidement une queue se crée pour utiliser ces méthodes limitant ainsi la parallélisation. Par ailleurs nous avons bien constaté que la parallélisation pouvait s'avérer une bonne solution mais qu'elle n'était pas adaptée à tous les problèmes.