Projet :
Projet systèmes concurrents/intergiciels

Table des matières

Remarque : Ceci est le document réponse, il ne reprend pas l'intégralité du sujet.

1. Introduction

1.1. Membres de groupe

Nous sommes trois étudiants du groupe M2 :

- ❏ Ying LIU
- ❏ Philippe NEGREL-JERZY
- ❏ Sébastien PONT

1.2. Présentation

Nous ne représentons pas tout le sujet (que vous pouvez retrouver ici [1]), mais voici un résumé.

Linda est un service permettant de partager des données sous formes de `Tuple`. Dans ce projet nous allons implémenter deux manières de partager et gérer ces ressources à travers plusieurs clients :

— une version dite "locale" à base de mémoire partagée (`shm` package Figure 1)
— une version dite "distante" à base de clients / monoserveur (`server` package Figure 1)

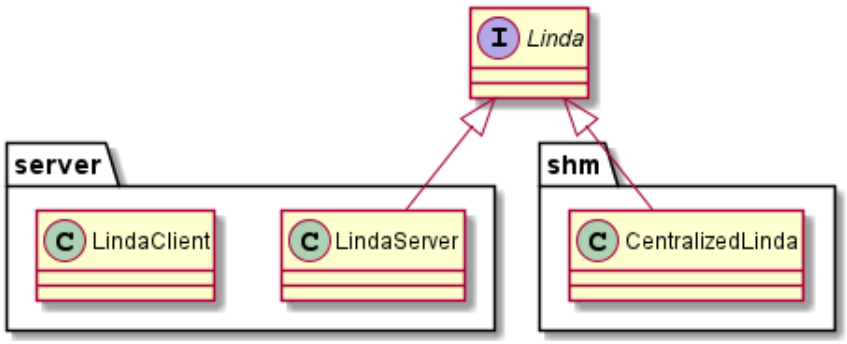

FIGURE 1 – Diagramme de classe général du projet

- ❏ Version mémoire partagée (version locale) : Chaque client est un nouveau *thread*.
- ❏ Version client / monoserveur (version distante) : Un serveur tourne et communique via *RMI* aux différents clients.

1.3. Gradle

Pour simplifier la gestion des dépendances et faciliter le développement nous avons choisi d'utiliser l'outil *Gradle* [2]. Ainsi, pour compiler le programme, il suffit par exemple de taper dans un terminal à la racine du projet :

</> Lancer l'application java via Gradle

```
./gradlew run
```

2. Version *mémoire partagée*

2.1. Version naïve

Pour implanter la version mémoire partagée, nous construisons l'architecture dans `CentralizedLinda` ci-dessous, on fait une première version sans se préoccuper d'`eventRegister`. Les interfaces et classes `Linda`, `Tuple`, `Callback`, `AsynchronousCallback` sont figées et ne doivent en aucun cas être modifiées.

1. https://spont.me/mjxoog
2. gradle.org

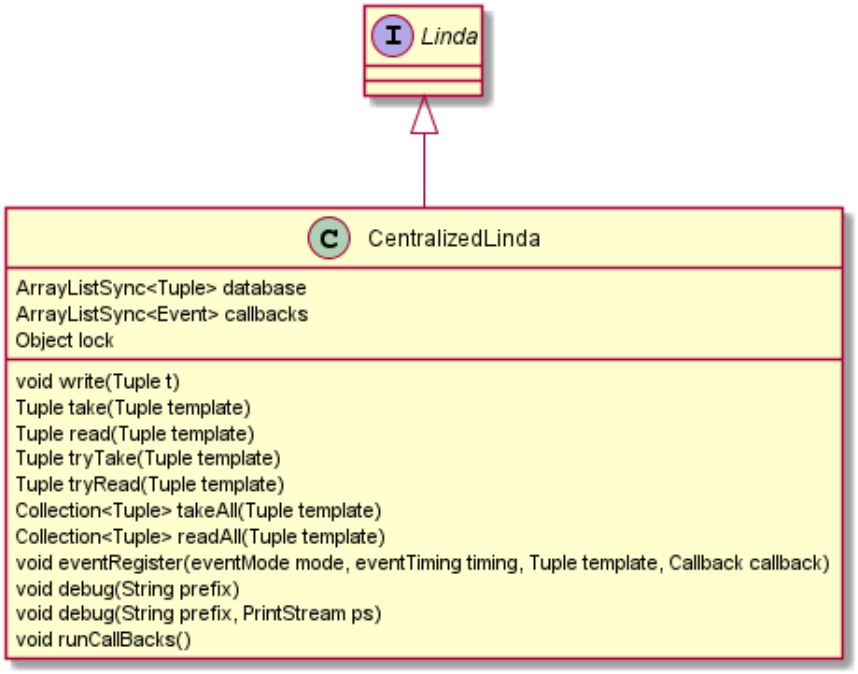

FIGURE 2 – CentralizedLinda

Pour l'`eventRegister`, on fait l'algorithme ci-dessous :

- On met en réserve le *callback*
- Si `timing` vaut `immediate`, on supprime le *callback*

</> Code 1 : CentralizedLinda.eventRegister

```
1  public void eventRegister(eventMode mode, eventTiming timing, Tuple template, Callback callback) {
2     // store the callback
3     Event e = new Event(template, callback, mode);
4     callbacks.add(e);
5
6     // if the event is immediate, remove the callback, then run it
7     if (timing == eventTiming.IMMEDIATE) {
8       for (Tuple t : database.clone()) {
9         if (t.matches(template)) {
10          callbacks.remove(e);
11          if (mode == eventMode.TAKE) {
12            database.remove(t);
13          }
14          callback.call(t);
15        }
16      }
17    }
18  }
```

2.2. Version multithreadée

3. Version *client/monoserveur*

Pour implanter la version client / monoserveur, nous construisons l'architecture des `LindaClient` et `LindaServer` ci-dessous. La classe `linda.server.LindaClient` doit être une implantation de l'interface Linda.

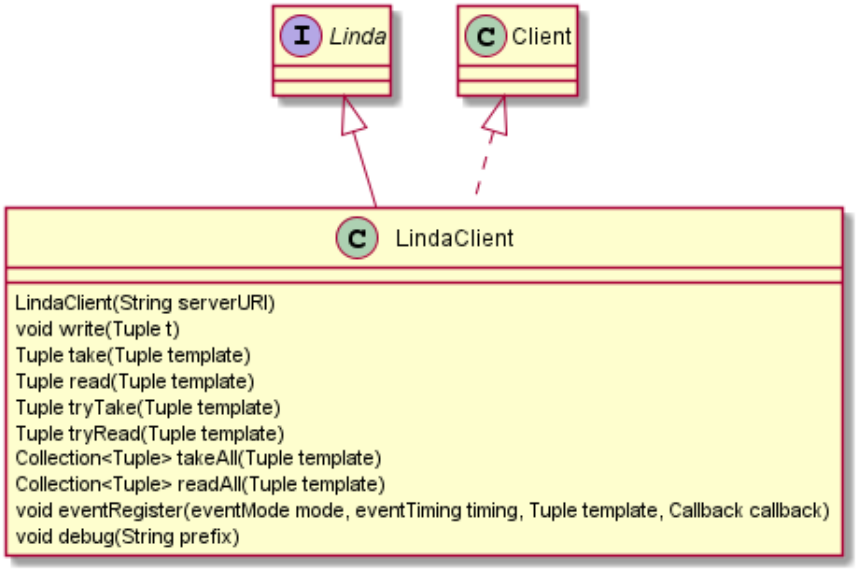

FIGURE 3 – Linda Client

`LindaServer` doit être une implantation de l'interface `LindaRemote` et extension de `UnicastRemoteObject`.

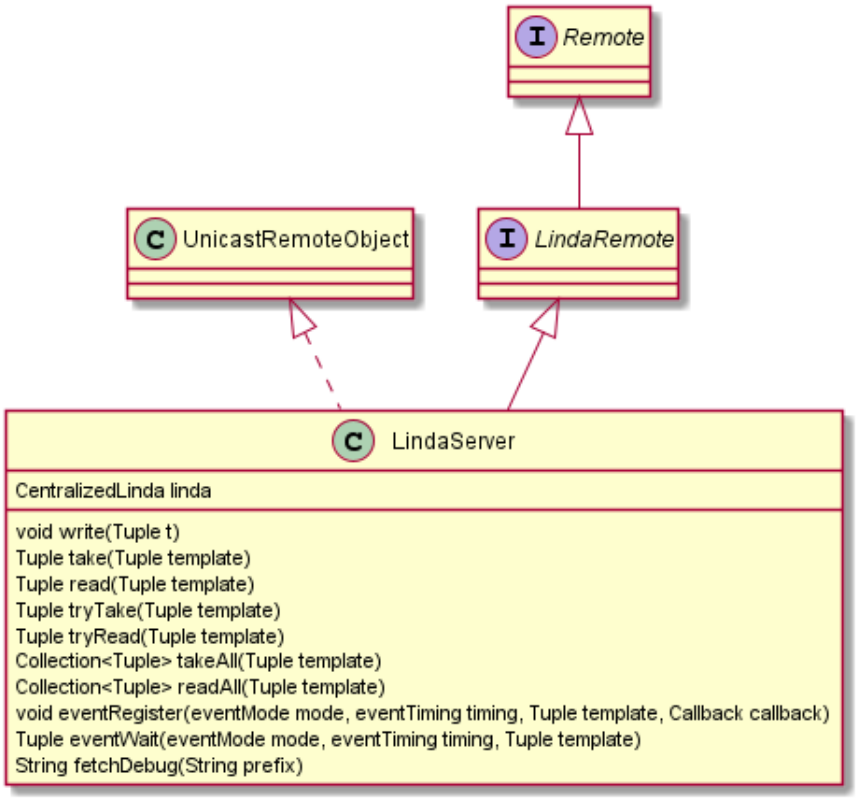

FIGURE 4 – Linda Server

4. Calcul des nombres premiers inférieurs à k

Pour calculer des nombres premiers inférieurs à k, on fait une version séquentielle basée sur la technique du crible d'Eratosthène, puis envisager et réaliser différentes formes de parallélisation de cet algorithme. (`PrimeSearch.java`) Une comparaison deux différentes versions :

— `SequentialSearch` : L'exécution séquentielle est monothread, l'efficacité de l'exécution du code est très faible.

— `ParallelSearch` : Plusieurs threads exécutés en parallèle ensemble, l'efficacité est relativement élevée.

5. Conclusion

5.1. Problèmes rencontrés

Voici une liste succincte des problèmes rencontrés lors du développement du projet :

— `eventRegister` : difficile de le faire tourner côté client et non pas serveur. Solutions envisagées :
  — ajouter une référence du client sur le serveur
  — le client exécute un nouveau thread bloquant avec `callback`
— `lock` plus on utilise de `lock`s, plus il y a de *deadlock*s dans le programme.