Projet Données Réparties
Sciences du Numérique – Semestre 7

Rapport provisoire

Enzo PETIT Nam VU

12 décembre 2021

Architecture actuelle

Nous avons à l'heure actuelle implémenté la version mémoire partagée de Linda avec la gestion des callbacks.

La classe `linda.shm.CentralizedLinda` a ainsi été complétée et des tests unitaires sous JUnit 5 ont été rédigés dans `linda.test.CentralizedLindaTest`.

Une classe auxiliare `linda.shm.Event` représente un *event* enregistré lors d'un appel à `registerEvent`.

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 :

`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.

`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).

`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`).

`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.

`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 `SynchronousQueue`, 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.

Tests

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

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

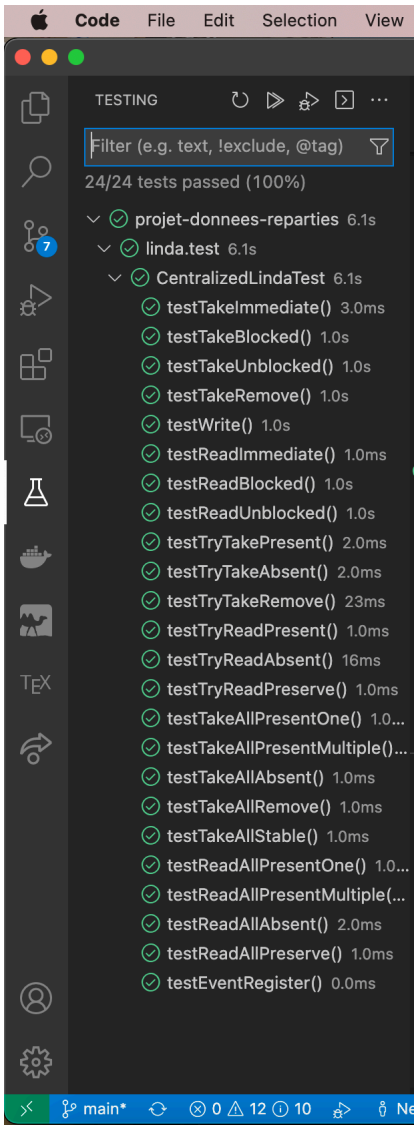

FIGURE 1 – Résultats des tests définis dans `linda.test.CentralizedLindaTest` (Visual Studio Code + Extension Pack for Java)

Suite du projet

Nous n'avons pas encore réfléchi à la version client-serveur à l'heure actuelle…