Projet Données Réparties

Rapport de suivi #1

Paul Anaclet

Guohao Dai

Théo Desprats

2A SN

2021

1 Architecture

1.1 Classes et structures de données

Stockage des tuples :

- `tuples = CopyOnWriteArrayList<Tuple>`

Pour stocker l'ensemble des tuples nous avons décidé d'utiliser des `CopyOnWriteArrayList` qui sont des variantes thread-safe des ArrayLists permettant de gérer les accès concurrents. Cependant, nous ne savons pas si nous allons garder cette structure de données car elle pourrait être trop coûteuse et moins pertinente dans le cadre de l'exercice de synchronisation.

Stockage des callbacks :

- `Read/TakeCallbacks = Map<Tuple, CopyOnWriteArrayList<Callback>>`

Nous avons associés les callbacks READ/TAKE à leur template correspondant. Nous avons placés les callbacks dans une `CopyOnWriteArrayList` pour les mêmes raisons que précedemment.

Exclusion :

- `accesTuples/Callbacks = ReentrantLock`

La cohérence des tuples et des callbacks de l'espace partagé est protégée grâce à deux `ReentrantLock` utilisés dès qu'une primitive accède respectivement à la collection des tuples et aux maps des callbacks.

Primitives bloquantes :

- `read(Tuple template) / take(Tuple template)`

Pour que les primitives soient bloquantes nous avons décidé d'instancier un nouveau callback associé à la requête puis de l'enregistrer (`eventRegister`) en mode `IMMEDIATE` : si le tuple correspondant existe déjà dans la base, alors le callback reçoit le tuple, sinon il est en attente d'un tuple (signalé à l'appel de la primitive `write`).

Afin que le callback puisse se mettre en attente/se faire réveiller, nous avons créée une nouvelle implémentation de l'interface `Callback` : `SynchronousCallback`. L'attente (méthode `wait()`) et le réveil (méthode `notify()` dans `call()`) se font donc dans des blocs `synchronized` dépendant du `SynchronousCallback` lui même. Il est à noter que le réveil d'un `SynchronousCallback` n'a lieu qu'à l'appel de son `call()`, invoqué seulement s'il y a correspondance.

1.2 Difficultés identifées

Maps de callbacks :

- `Read/TakeCallbacks = Map<Tuple, CopyOnWriteArrayList<Callback>>`

Pour chaque `write(tuple)` nous avons du construire une clé pour pouvoir effectuer la recherche dans les maps de callbacks : les callbacks réveillés par un `write` ne sont accessibles que par ces maps dont les entrées sont par exemple de forme `[Integer.class String.class]: [callback1, callback2]` et non de la forme `[4 «foo»]:[callback1, callback2]`.

Une autre question que nous nous sommes posés concernait la suppression, ou non, d'une entrée des maps si la liste associée à un template devenait vide. Nous avons pour l'instant décidé de ne pas retirer l'entrée si la liste devenait vide.

2 Tests

Primitives :

- `TestReadAll / TestTakeAll`

Pour tester le bon fonctionnement de toutes les primitives de Linda, nous avons ajouté ces deux fichiers de tests qui vérifient les méthodes `readAll` et `takeAll`.

Autres :

Nous n'avons malheureusement pas encore eu le temps de mettre au point beaucoup de tests différents, notamment ceux concernant nos choix arbitraires et ceux impliquant plusieurs callbacks.