Données réparties : Rapport Provisoire

SOUCHON Théo, ROGARD Diego, NEYENS Pablo
(12 / 12 / 21)

1. Architecture

Nous avons réalisé la version de Linda en mémoire partagée (fichier *CentralizedLinda.java*).
Afin de stocker efficacement les différent callbacks, nous avons utilisé la structure de données suivante :

Map<Tuple, Map<eventMode, Vector<Callback>>>

Ainsi, nous pouvons associer à chaque tuple modèle, deux listes (implémentées par des *Vector*) de callbacks correspondantes aux deux modes (*eventMode*).
Afin de manipuler plus facilement cette structure un peu complexe, nous avons créé deux fonctions addCallback et removeCallback qui permettent, comme leur nom l'indique, d'ajouter et d'enlever des callbacks.

2. Algorithmes des opérations essentielles

La fonction *write* ajoute le tuple dans l'espace des tuples du noyau. Ensuite, elle réveille tous les *read* en attente afin qu'ils puissent récupérer leur tuple. Puis elle vérifie les callbacks en mode *read*, ceux en mode *take*, et enfin permet aux *take* de récupérer le tuple et donc de l'enlever de l'espace.

La fonction *eventRegister* vérifie d'abord le timing du callback. S'il est immédiat, on tente de réaliser l'action (*read* ou *take*) et on enregistre un callback seulement si on échoue. Sinon, on enregistre directement le callback dans notre structure de données. Le callback enregistré ne sera donc appelé que par la fonction *write* si celle-ci rajoute un tuple correspondant au modèle.

3. Points délicats et leurs résolutions envisagées

Pour éviter d'utiliser des *signalAll()*, nous avons utilisé une file (sous forme de *Vector*) de variables conditions. Chaque variable correspond à un *read* ou un *write* en attente et elle sera utilisée pour les réveiller le cas échéant. Comme on réveille les processus dans l'ordre de la file, la priorité s'assimile à un ordre FIFO. Ce n'est pas exactement le cas car les processus réveillés dans un ordre précis ne se bloqueront pas forcément dans le même ordre.

4. Tests

Nous avons également réalisé une série de tests unitaires. Ces tests sont sous la forme de mini scénarios, nous permettant de tester les fonctions écrites. Un de ces tests, regroupant les différentes fonctions *read()*, *take()*, et *eventRegister()* (en mode *read* et *take*), a mis en évidence un problème dans le code que nous n'avons pas pû résoudre dans le temps imparti. Cela restera donc un point à régler par la suite.