RAPPORT
Projet de systèmes intergiciels

Édouard Lumet <edouard.lumet@etu.enseeiht.fr>
Bastien Pelissier <bastien.pelissier@etu.enseeiht.fr>

18 avril 2019

Sommaire

Première partie

Préambule

Merci de me lire !

Ce document a pour but de présenter les implantations proposées ainsi que les réflexions menées lors du projet de systèmes intergiciels implantant un modèle *Linda*. Vous y trouverez les architectures, les algorithmes essentiels ainsi que les difficultés rencontrées lors de ce projet. Le plan de test se trouve en annexe A.

Le projet consiste en la conception d'un système concurrent/intergiciel. On réalise un espace de données partagé inspiré du modèle *Linda*. Les données manipulées ici sont des tuples, à l'aide de primitives afin d'écrire, lire, retirer etc ces tuples sur la base de motifs. La concurrence vient du fait que l'espace de données est partagé (appelé ici *tuplespace* tandis que la partie *intergiciel* apparaît au niveau de la communication client-serveur(s).

Concernant le système concurrent, l'enjeu se trouve au niveau de l'accès au tuple, sachant que certaines opérations sont bloquantes tandis que d'autres ne le sont pas. Concernant le système intergiciel, l'enjeu est l'utilisation de *RMI* pour la communication entre un ou plusieurs client(s) et un ou plusieurs serveur(s).

Deuxième partie

Version en mémoire partagée

Solution implantée

On trouve ci-dessous l'architecture de notre système concurrent. La partie *intergiciel* n'est pas encore traitée. On réalise une implantation de l'interface *Linda* qui ne doit en aucun cas être modifiée. On doit donc implanter les différentes primitives et créer le *tuplespace*, l'espace partagé contenant les tuples.

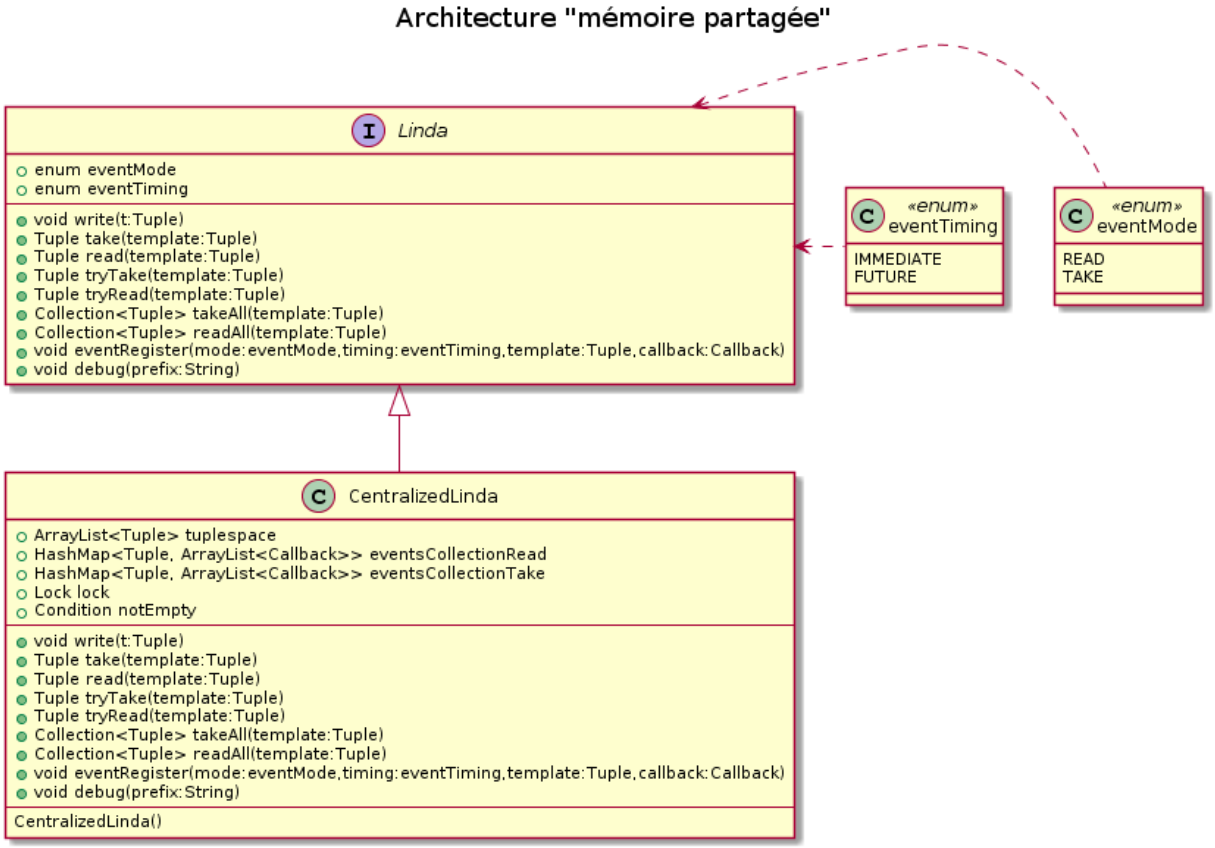

FIGURE 1 – Diagramme UML de la version en mémoire partagée

Algorithme de eventRegister

```
Si timing est IMMEDIATE:
    Cas mode.READ:
        tentative de lecture non bloquante
        Si échec:
            Si la collection d'évènements contient le motif:
                ajout du callback à la liste correspondant au motif
            Sinon:
                création du motif et de la liste de callbacks associée
                ajout du callback à la liste créée
        Sinon:
            appel du callback
    Cas mode.TAKE
        tentative d'extraction non bloquante
        Si échec:
            Si la collection d'évènements contient le motif:
                ajout du callback à la liste correspondant au motif
            Sinon:
                création du motif et de la liste de callbacks associée
                ajout du callback à la liste créée
Sinon:
    Cas mode.READ:
        Si la collection d'évènements contient le motif:
            ajout du callback à la liste correspondant au motif
        Sinon:
            création du motif et de la liste de callbacks associée
            ajout du callback à la liste créée
    Cas mode.TAKE
        Si la collection d'évènements contient le motif:
            ajout du callback à la liste correspondant au motif
        Sinon:
            création du motif et de la liste de callbacks associée
            ajout du callback à la liste créée
```

Algorithme de write

```
verrouillage
écriture du tuple dans le tuplespace
Si la collection d'eventRead contient le motif ajouté:
    appel du callback
    suppression du callback
    Si la liste correspondant au motif est vide:
        suppression de la liste
Sinon:
    Si la collection d'eventTake contient le motif ajouté:
    appel du callback
    suppression du callback
    Si la liste correspondant au motif est vide:
        suppression de la liste
réveil des threads en attente
déverrouillage
```

Pour gérer les accès concurrents entre les *threads*, nous avons utilisé un verrou commun de type *ReentrantLock* et une condition permettant de mettre les opérations bloquantes en attente jusqu'à écriture.

signalAll() est utilisé lors de l'opération d'écriture pour notifier l'ensemble des *threads* en attente.

Nous avons choisi de regrouper les *callbacks* par motif dans des listes. Ces listes sont ensuite stockées dans une *HashMap* avec pour clé le motif correspondant.

Deux *HashMap* différentes sont utilisées, une pour le mode *take* et une pour le mode *read*.

Problèmes rencontrés

Nous avons rencontré différents problèmes lors de l'implantation de cette version. Certains ont pu être résolus, d'autres non. Dans tous les cas nous avons des pistes de solutions pour pallier lesdits problèmes.

Premièrement, nous pouvons noter le manque de finesse dans l'utilisation des verrous en termes de concurrence. Cela permet d'éviter les bugs entre les différentes primitives sur un grand nombre d'opération mais on perd en performance. Malheureusement nous n'avons pas eu le temps d'implanter des *ReadWriteLock* pour affiner les blocages.

Ensuite, l'utilisation de *HashMap* paraissait judicieuse au départ pour gagner du temps dans la manipulation des collections d'évènements en attente. Cependant, cela pose un problème lorsque nous devons parcourir la collection. Utiliser une *LinkedHashMap* (ou *LinkedList*) aurait été plus judicieux.

Troisième partie

Version client / mono-serveur

Solution implantée

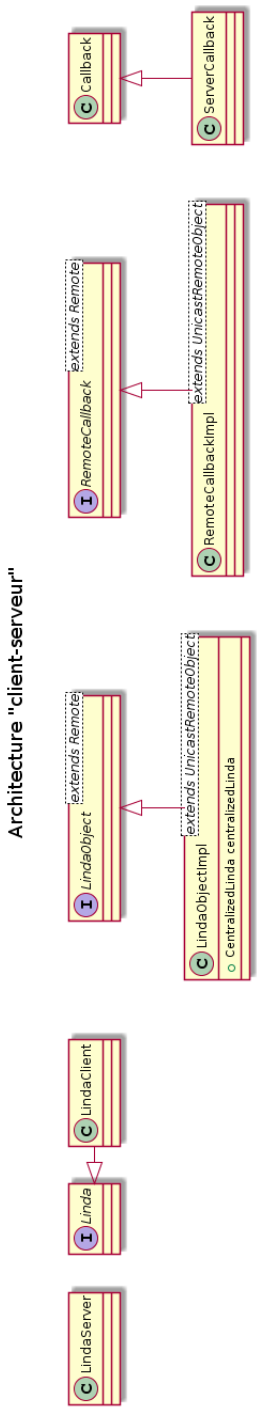

Figure 2 – Diagramme UML de la version client-serveur

Annexe A

Plan de tests

Annexe B

Sources