Projet systèmes concurrents/intergiciels

2IR

4 mars 2025

1 Linda

Le projet a pour but de réaliser un espace partagé de données typées. Cette approche est inspirée du modèle Linda (ou TSpaces). Dans ce modèle, les processus partagent un espace de *tuples* qu'ils peuvent manipuler à l'aide d'un jeu de primitives spécifiques.

1.1 Qu'est-ce qu'un tuple ?

Tuple de valeurs : un tuple de valeurs est un n-uplet ordonné de valeurs. Les éléments sont de type quelconque (entier, booléen...ou même tuple). Par exemple, les données suivantes sont des tuples :
`[ 10 'A' true ]`, `[ 1 2 3 ]`, `[ "azerty" 1 ]`, `[ 'A' [ 10000 false ] 23 ]`

Motif : un tuple motif (ou *template*) est un n-uplet dont certaines des composants sont des types, qui représentent « n'importe quelle valeur de ce type ». Par exemple `[ ?Integer true ]` est un motif et les tuples `[ 6 true ]` et `[ 2 true ]` correspondent (« match ») au motif. Dans notre cas, un type est une classe Java. On peut utiliser `?Object` qui correspond à n'importe quelle valeur de n'importe quel type.

Par exemple le tuple `[ 'A' [ 10000 false ] 2 ]` correspond aux motifs `[ 'A' ?Tuple ?Integer ]`, `[ ?Character [ ?Integer false ] ?Object ]` et bien d'autres. La propriété « correspondre » est une propriété d'inclusion et forme un ordre partiel : on peut comparer des motifs (cf figure 1).

1.2 Les primitives du modèle Linda

- `write(tuple)` : dépose le tuple dans l'espace partagé ;
- `take(motif)` : extrait de l'espace partagé un tuple correspondant au motif précisé en paramètre ;
- `read(motif)` : recherche (sans l'extraire) dans l'espace partagé un tuple correspondant au motif fourni en paramètre ;
- `tryTake(motif)` : version non bloquante de `take` ;
- `tryRead(motif)` : version non bloquante de `read` ;
- `takeAll(motif)` : renvoie, en extrayant, tous les tuples correspondant au motif (vide si aucun ne correspond) ;
- `readAll(motif)` : renvoie, sans extraire, tous les tuples correspondant au motif (vide si aucun ne correspond) ;

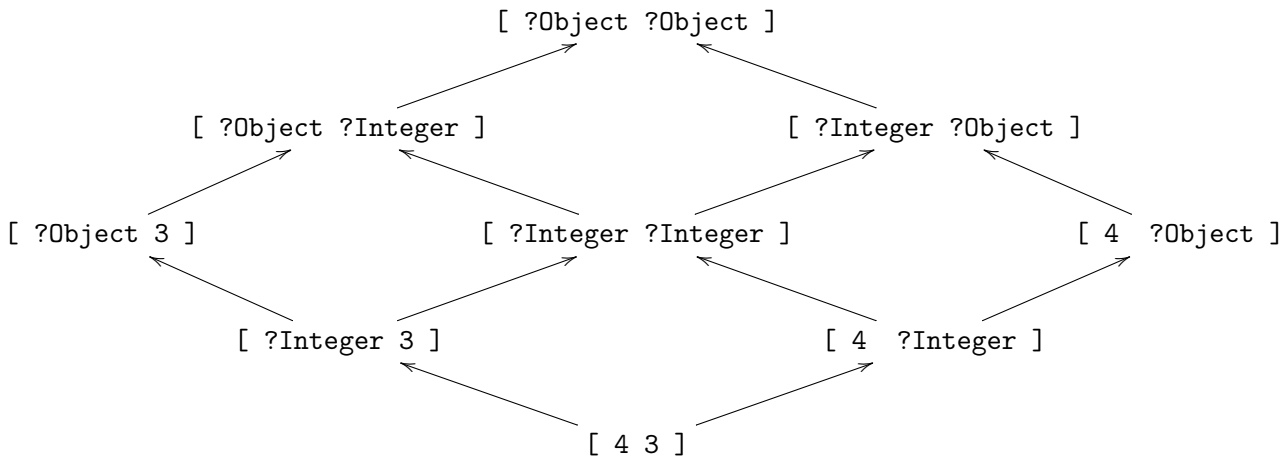

FIGURE 1 – Ordre partiel de la propriété de correspondance

— `eventRegister(mode, timing, motif, callback)` : s'abonner à l'événement d'existence/d'apparition d'un tuple correspondant au motif.

L'opération `write` ajoute une copie du tuple déposé dans l'espace des tuples partagés. L'espace des tuples est un multi-ensemble : un même tuple peut être présent en plusieurs exemplaires. Noter aussi que l'on peut déposer aussi bien des tuples de valeur et des tuples motifs.

Les primitives `take` et `read` sont bloquantes : l'appelant reste bloqué tant qu'aucun tuple de l'espace partagé ne satisfait le motif demandé.

L'opération `take` consiste à rechercher un tuple présent dans l'espace des tuples partagés. La recherche s'effectue d'après un tuple motif. Lorsqu'un tuple satisfaisant le motif est finalement trouvé (après une attente éventuelle), celui-ci est extrait de l'espace des tuples partagés.

L'opération `read` recherche un tuple correspondant au motif fourni en paramètre, et retourne à l'appelant une copie du tuple trouvé. Autrement dit, le tuple trouvé reste dans l'espace des tuples partagés.

Les opérations `tryTake` et `tryRead` sont les versions non bloquantes des opérations `take` et `read`. Elles renvoient le tuple trouvé (une copie si consultation) ou une valeur nulle si aucun tuple ne correspond.

Les opérations `takeAll` et `readAll` renvoient la collection de *tous* les tuples qui correspondent au motif, avec ou sans extraction de l'espace des tuples. Si aucun tuple ne correspond, ces opérations renvoient une collection vide : elles ne bloquent jamais.

Enfin, l'opération `eventRegister` permet de s'abonner à l'occurrence d'existence/d'apparition (selon le timing eventTiming.IMMEDIATE/FUTURE) d'un tuple correspondant au motif spécifié. Le `callback` est exécuté avec le tuple identifié. Ce tuple est retiré de l'espace de tuple (comme un `take`) si le mode est `eventMode.TAKE`, il est laissé dans l'espace si le mode est `eventMode.READ`. Le callback n'est déclenché qu'une fois ; s'il le souhaite, il peut se ré-engistrer lui-même. Noter que le callback peut être déclenché immédiatement à son enregistrement si un tuple correspondant au motif existe déjà dans l'espace de tuple et que le timing est `eventTiming.IMMEDIATE`.

Spécification libérale : la spécification des opérations est volontairement assez libérale :

- quand plusieurs tuples correspondent, `take` en retourne un arbitraire ;
- quand plusieurs `take` sont en attente et qu'un dépôt peut en débloquer plusieurs, le choix est arbitraire (pas nécessairement le plus ancien) ;
- quand des `read` et un `take` sont en attente et qu'un dépôt peut les débloquer, il n'est pas spécifié si, outre le `take`, tous les `read` doivent être débloqués, ou aucun, ou seulement certains ;
- quand il y a à la fois un `take` et un callback enregistré pour un même motif, le choix entre déblocage du `take` et déclenchement du callback n'est pas spécifié.

L'implantation pourra, au choix, définir précisément le comportement (par exemple FIFO) ou pas. La correction de l'exécution d'un exemple ne doit pas dépendre de ces choix.

1.3 Manipulation des tuples

Le type tuple est défini dans la classe `Tuple`. Il s'agit d'une liste d'objets (sérialisables à cause de la deuxième partie du projet).

Outre les méthodes de l'interface `List` (telles `add`, `get`...), la classe fournit des méthodes pour créer un tuple, pour tester si un tuple correspond à un motif ainsi que pour lire ou écrire des tuples.

2 Version en mémoire partagée

Il s'agit de réaliser une implantation de l'interface `Linda` qui s'exécute directement dans la même machine virtuelle que les codes utilisateurs.

Contrat : les interfaces et classes `Linda`, `Tuple`, `Callback`, `AsynchronousCallback` sont figées et ne doivent en aucun cas être modifiées. L'implantation doit être dans la classe `linda.shm.CentralizedLinda` avec un constructeur sans paramètre. Les exemples fournis doivent compiler et s'exécuter correctement sans qu'il soit nécessaire d'y faire le moindre changement.

Conseil : faire une première version sans se préoccuper d'`eventRegister`.

Attention : les exemples fournis ne sont pas des tests suffisants, loin de là !

3 Version client / mono-serveur

Il s'agit de réaliser une implantation de l'interface `Linda` qui accède à un serveur distant qui centralise l'espace de tuples.

Contrat : La classe `linda.server.LindaClient` (à écrire) doit être une implantation de l'interface `Linda`. Le constructeur doit prendre un unique paramètre chaîne qui est l'URI du serveur linda à utiliser (par exemple `//localhost:4000/LindaServer`).

Conseil : l'implantation d'`eventRegister` est le morceau acrobatique de cette partie.

Bonus si l'implantation du serveur réutilise en interne la version linda en mémoire partagée sans y toucher une virgule (et sans tricher en y rétroportant des aspects client/serveur).

4 Tolérance aux fautes

1. On souhaite pouvoir arrêter et redémarrer le serveur, en récupérant l'état de l'espace des tuples : il faut donc sauvegarder et recharger l'espace des tuples. Utiliser la sérialisation pour ajouter cette fonctionnalité.
2. On souhaite rendre le service plus résistant. Le serveur primaire est doublé d'un serveur de secours auquel il transfère régulièrement (tous les x secondes, ou tous les x changements) l'état de l'espace des tuples. Quand un client observe que le serveur primaire est inaccessible, il bascule sur le serveur de secours. Définir l'architecture, en particulier de manière à rendre la bascule totalement transparente pour le code applicatif. Prendre soin au traitement des callbacks.

Note : cette partie est volontairement sous-spécifiée. Garder séparée une version sans ces nouvelles fonctionnalités, pour que les tests d'origine compilent et fonctionnent.

5 Modalités pratiques

Projet exécuté *en binôme*. Les séances de suivi sont obligatoires : un point d'avancement est fait à chaque fois. Un test aura lieu le 4 juin 2025.

Squelette du code : `/home/queinnec/Projet-IR.tar`

Vous devez rendre le 4 juin 2025 :

— un *petit* rapport présentant l'architecture, les algorithmes des opérations essentielles, une explication claire des points délicats et de leur résolution, et un mot sur les exemples originaux développés ;

— le code complet, y compris le plan de test et les tests que vous avez développés.

Attention : le code des exemples fournis doit compiler et s'exécuter correctement *sans que vous y touchiez le moindre caractère*. Ces tests valident votre respect de l'API. Ils ne sont cependant absolument pas exhaustifs et vous devez développer vos propres tests.

Organisation des codes sources

Package `linda` : `Tuple`, `TupleFormatException`, `Linda`, `Callback`, `AsynchronousCallback` (intégralement fourni, classes et interfaces pour les codes utilisateurs, ne pas modifier) ;

Package `linda.shm` : classe `CentralizedLinda` (implantation de `linda.Linda` en mémoire partagée, à écrire sous ce nom). Vous pouvez ajouter ici d'autres classes/interfaces, mais elles seront cachées aux codes utilisateurs ;

Package `linda.server` : classe `LindaClient` (implantation de `linda.Linda` pour accéder à un serveur distant). Vous pouvez ajouter ici d'autres classes/interfaces, mais elles seront cachées aux codes utilisateurs ;

Package `linda.test` : nos tests élémentaires ;

Package `linda.whiteboard` : un exemple d'application ;

Package `linda.`*`autre`* : ce que vous voulez.