Projet

Service de partage d'objets répartis et dupliqués en Java

Daniel Hagimont, Philippe Mauran, Philippe Queinnec, Alain Tchana

ENSEEIHT Département Informatique, 2ième année, 2015-2016

Objectif du projet

Le but de ce projet est d'illustrer les principes de programmation concurrente et répartie vus en cours. Pour ce faire, nous allons réaliser sur Java un service d'accès transactionnel à des objets partagés par duplication, reposant sur la cohérence à l'entrée (*entry consistency*).

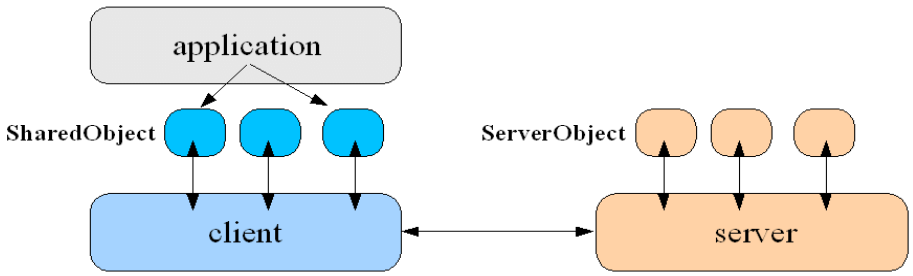
Les applications Java utilisant ce service peuvent accéder à des objets répartis et partagés de manière efficace puisque ces accès sont en majorité locaux (ils s'effectuent sur les copies (réplicas) locaux des objets). Durant l'exécution, le service est mis en œuvre par un ensemble d'objets Java répartis qui communiquent au moyen de Java/RMI pour implanter le protocole de gestion de la cohérence.

Présentation du service

Dans ce service, les objets sont représentés par des descripteurs (instances de la classe *SharedObject* d'interface *SharedObject_itf*) qui possèdent un champ *obj* qui pointe sur l'instance Java partagée. Toute référence à une instance partagée doit passer par une telle indirection. Une instance peut correspondre à une grappe d'objets.

Dans une première étape, cette indirection est visible pour le programmeur qui doit adapter son mode de programmation. Dans une seconde étape, on implantera des stubs qui masquent cette indirection.

Le service est architecturé comme suit.



L'utilisation d'un objet partagé se fait toujours avec une indirection à travers un objet de classe *SharedObject*. Cette classe fournit notamment des méthodes *lock_read()*, *lock_write()* et *unlock()* qui permettent de mettre en œuvre la cohérence à l'entrée depuis l'application. Notons que dans le cadre de ce service, on ne gère pas les prises de verrou imbriquées ; un *lock_read()* ou *lock_write()* sur un objet doit être suivi d'un *unlock()* pour pouvoir reprendre

un verrou sur le même objet. Il est par ailleur parfaitement possible de détenir des verrous sur plusieurs objets différents, mais le traitement des interblocages est laissé, pour cette version, à la charge du programmeur.

Un *SharedObject* contient notamment un entier (*id*) qui est un identifiant unique alloué par le système (ici un serveur centralisé) à la création de l'objet, ainsi qu'une référence à l'objet lorsqu'il est cohérent (*obj*).

Si l'objet *O* est accessible à travers un *SharedObject S*, alors l'appel de la méthode *M* avec verrouillage en lecture se fera ainsi :

```
S.lock_read() ;
Type_de_O var  = (Type_de_O )S.obj;
var.M() ;
S.unlock() ;
```

La couche (classe) appelée *Client* fournit les services pour créer ou retrouver des objets dans un serveur de noms (comme le *Registry* de RMI) :

- static void init() : initialise la couche cliente, à appeler au début de l'application.
- static SharedObject create (Object o) : permet de créer un objet partagé (en fait un descripteur) initialisé avec l'objet o. Le descripteur de l'objet partagé est retourné. A la création, l'objet n'est pas verrouillé.
- static SharedObject lookup (String n) : consulte le serveur de nom et retourne l'objet partagé enregistré.
- static void register (String n, SharedObject_itf so) : enregistre un objet partagé dans le serveur de noms.

Le fichier Irc.java vous donne un exemple d'application utilisant un objet partagé de classe Sentence (mais elle n'utilise pas des structures complexes d'objets partagés comme des graphes). Cette application sera utilisée pour tester (dans un premier temps) votre projet, mais vous devrez implanter d'autres jeux de test.

Implantation du service

Les *SharedObject*, qui sont utilisés par les programmes pour tous les accès aux objets, contiennent des informations sur l'état des objets du point de vue de la cohérence. On y trouve notamment une variable entière *lock* qui signifie :

```
int lock;          // NL : no local lock
                   // RLC : read lock cached (not taken)
                   // WLC : write lock cached
                   // RLT : read lock taken
                   // WLT : write lock taken
                   // RLT_WLC : read lock taken and write lock cached
```

Un verrou est dit *taken* s'il est pris par l'application. Il sera libéré au prochain *unlock()* et passera alors à l'état *cached*. Le verrou est dit *cached* s'il réside sur le site sans être pris par l'application. Un verrou *cached* peut alors être pris par l'application sans communication avec le serveur. L'état hybride RLT_WLC correspond à une prise de verrou en lecture par l'application alors que le SharedObject possédait un WLC

En fonction de l'état de l'objet sur le site client, la demande d'un verrou nécessitera (ou pas) de propager un appel au serveur.

Pour mettre en œuvre la cohérence, les méthodes *lock_read()* et *lock_write()* de la classe *SharedObject* ont besoin d'appeler des méthodes du serveur qui gèrent la cohérence. Ces appels passent par la couche cliente (classe *Client*) qui fournit les méthodes :

- *static Object lock_read (int id)* : demande d'un verrou en lecture au serveur, en lui passant l'identifiant unique de l'objet (le SharedObject devait être en NL). Retourne l'état de l'objet.
- *static Object lock_write (int id)* : demande d'un verrou en écriture au serveur, en lui passant l'identifiant unique de l'objet (le SharedObject devait être en NL ou RLC). Retourne l'état de l'objet si le SharedObject était en NL.

Ces méthodes (statiques) de la classe *Client* ne font que propager ces requêtes au serveur, en ajoutant aux paramètres de la requête une référence au client (instance) lorsque cela est nécessaire.

L'interface du serveur (distante) comprend donc les méthodes suivantes (d'autres étant ajoutées, notamment pour implanter le service de nommage évoqué ci-dessus) :

- Object lock_read(int id, Client_itf client)
- Object lock_write(int id, Client_itf client)

Ces méthodes incluent une référence au client afin de pouvoir le rappeler ultérieurement.

Sur le site serveur, la gestion de la cohérence d'un objet est attribuée à une instance de la classe *ServerObject*. La classe *ServerObject* indique l'état de la cohérence de l'objet du point de vue du serveur, avec notamment le client écrivain si l'objet est en écriture ou la liste des clients lecteurs si l'objet est en lecture (afin d'être en mesure de propager des invalidations). Les appels au serveur sont transférés au *ServerObject* concerné, ce *ServerObject* implantant une interface similaire:

- Object lock_read(Client_itf client)
- Object lock_write(Client_itf client)

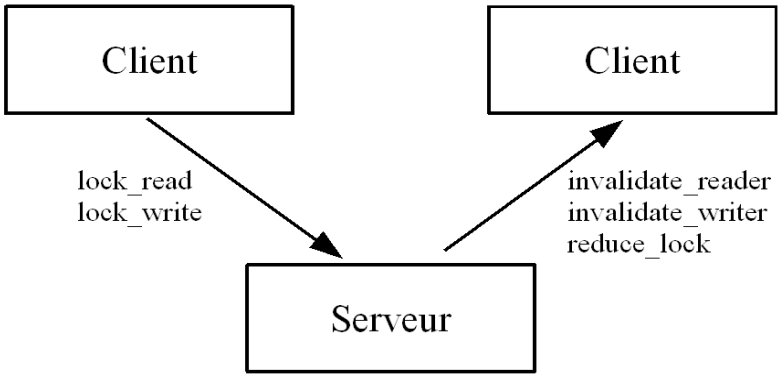
La référence au client reçue par le serveur lui permet de réclamer un verrou et une copie de l'objet au client qui la possède. L'interface du client (distante) permettant cette réclamation est la suivante :

- *Object reduce_lock(int id)* : permet au serveur de réclamer le passage d'un verrou de l'écriture à la lecture.
- *void invalidate_reader(int id)* : permet au serveur de réclamer l'invalidation d'un lecteur.
- *Object invalidate_writer(int id)* : permet au serveur de réclamer l'invalidation d'un écrivain.

Une telle réclamation est propagée au descripteur d'objet concerné (*SharedObject*) sur le site client qui implante les mêmes méthodes:

- Object reduce_lock()
- void invalidate_reader()
- Object invalidate_writer()

Les appels de méthodes d'un client au serveur et du serveur à un client sont représentés sur la figure suivante.



Attention : les requêtes peuvent se croiser, par exemple un client peut envoyer un lock_write au serveur pour augmenter son verrou (il possède déjà un RLC) et en même temps, le serveur envoie un invalidate_reader à ce client.

Travail demandé

Etape 1

On vous demande d'implanter le service de gestion d'objets partagés répartis. Dans cette première version, les SharedObject sont utilisés explicitement par les applications.

Plusieurs applications peuvent accéder de façon concurrente au même objet, ce qui nécessite de mettre en œuvre un schéma de synchronisation globalement cohérent pour le service que vous implantez.

On suppose que chaque application voulant utiliser un objet en récupère une référence (à un SharedObject) en utilisant le serveur de nom. On ne gère pas le stockage de référence (à des objets partagés) dans des objets partagés.

Attention : vous devez scrupuleusement respecter les interfaces qui sont spécifiées, surtout pour Client et SharedObject_itf qui sont utilisées par les applications dont Irc.java est un exemple. Nous vous demandons de ne pas gérer de package pour faciliter l'utilisation de nos applications de test lors de l'évaluation finale.

Etape 2

Cette étape consiste à implanter un service d'accès transactionnel aux objets dupliqués. Ce service sera utilisé à travers l'interface et le protocole d'usage suivants :

- Une classe *Transaction* est définie, qui fournit les méthodes *start(), commit(), abort(),* et *isActive()* ainsi que la méthode statique *getCurrentTransaction()*. Ces deux dernières méthodes sont fournies afin de permettre de différencier le traitement de *lock()/unlock()* selon que l'application appelante est en mode transactionnel ou non.
- Une application peut passer en mode transactionnel en créant une transaction, et appelant la méthode *start()* de cette transaction. Dès lors, tous les accès aux objets qui auront été verrouillés après l'appel à *start()* seront gérés de manière transactionnelle, jusqu'à ce que l'application revienne en mode non transactionnel. La **gestion des transactionnelle** des accès pour une application consiste à garantir que l'ensemble de ces accès est **atomique** (c'est-à-dire est soit complètement exécuté soit sans aucun effet) et **isolé** (sans interférences) p**ar rapport aux autres applications accédant à ces objets en mode transactionnel**. Une application en mode transactionnel revient en mode non transactionnel en appelant

- soit la méthode *commit()* qui demande la validation atomique des écritures réalisées en mode transactionnel. Cette méthode renvoie un booléen, qui indique le succès ou l'échec de la validation. En cas d'échec, le résultat est le même que celui obtenu par l'appel à la méthode *abort()*.
- soit la méthode *abort()*, qui annule les effets de la transaction, *pour les autres applications en mode transactionnel*.

- On pourra supposer/imposer que les transactions ne peuvent être imbriquées (récursives ou décomposables en sous-transactions concurrentes) dans un même client, et donc que deux appels à *start()* ne peuvent se suivre sans être séparés par un appel à *commit()* ou à *abort()*.
- On pourra également supposer/imposer qu'il est interdit de verrouiller/libérer plusieurs fois un même objet au sein d'une même transaction.

Remarques

- L'implémentation demandée n'a pas à couvrir les pannes de clients ou de serveurs.
- L'atomicité et l'isolation ne doivent être gérées que pour les applications concurrentes en mode transactionnel, et seulement sur les objets accédés dans ce mode ; aucune garantie de cohérence (autre que celle liée à la cohérence des copies dupliquées) n'est requise vis-à-vis des applications en mode non transactionnel, ou des objets verrouillés avant le passage en mode transactionnel. Pour accéder en mode transactionnel à un objet verrouillé avant le passage en mode transactionnel, il faut le libérer *avant* de passer dans ce mode.
- Les deux formes de cohérence qui sont abordées (cohérence à l'entrée pour la réplication et sérialisabilité pour les transactions) sont clairement distinctes, voire orthogonales : pour la première, il s'agit de garantir la cohérence d'accès concurrents sur un même objet, alors que pour la seconde, il s'agit de garantir la cohérence de suites d'accès à des objets distincts. Il faudra veiller lors de la seconde étape à ne pas remettre en cause les choix de conception de la première étape, lesquels visent à favoriser des accès aux données indépendants, locaux à chaque application.

Bonus

Le service transactionnel, comme le service de réplication ne prennent pas en compte les interblocages. Il serait utile de disposer de mécanismes permettant de les traiter ou de les éviter à ce niveau. Si vous envisagez ce développement, il faudra veiller à rester

- simple, tant dans la conception que dans l'interface proposée,
- compatible avec les choix de conception précédents.

Etape 3

On doit implanter un générateur de stub, destiné à soulager le programmeur de l'utilisation explicite des SharedObject.

Nous prenons les hypothèses suivantes:

- un objet partagé est une classe sérialisable (par exemple Sentence). Il s'agit de la classe métier de l'objet partagé.
- l'objet partagé est utilisable à partir de variables de type une interface (par convention, l'interface pour utiliser un objet partagé de classe Sentence s'appellera Sentence_itf) qui inclut les méthodes métier de Sentence auquel on ajoute les méthodes de verrouillage en héritant de SharedObject_itf. Mais Sentence

n'implémente pas Sentence_itf, car la classe métier ne définit pas les méthodes de verrouillage (c'est le stub qui le fait).

- un stub est généré, appelé Sentence_stub. Ce stub hérite de SharedObject (donc des méthodes de verrouillage) et il implémente l'interface Sentence_itf.

Avec ces hypothèses, les interfaces de la classe Client (notamment pour le serveur de nom) restent les mêmes. Et pour utiliser un objet partagé de la classe Sentence, on fera :

```
Sentence_Itf s = (Sentence_Itf)Client.lookup ("MySentence");
s.lock_read() ;
s.meth();
s.unlock() ;
```

On pourra également étudier l'annotation des méthodes dans les interfaces afin de leur associer un mode de verrouillage (@read ou @write). Ainsi, le verrouillage de l'objet partagé est réalisé par le stub généré et on obtient la séquence d'appel suivante :

```
Sentence_Itf s = (Sentence_Itf)Client.lookup ("MySentence");
s.meth();
```

Etape 4

On désire prendre en compte le stockage de référence (à des objets partagés) dans des objets partagés. Le problème qui se pose est la copie d'un objet partagé O1, qui inclut une référence à un objet O2, entre deux machines M1 et M2. O1 inclut une référence au stub de O2 sur la machine M1. Après la copie, il faut que la copie de O1 inclut la référence au stub de O2 sur la machine M2. Ceci nécessite d'adapter les primitives de sérialisation des stubs pour que, lorsqu'un stub est sérialisé sur M1, on ne copie pas l'objet référencé et lorsqu'il est désérialisé sur M2, le stub soit installé de façon cohérente sur M2 (sans installer plusieurs stubs pour un même objet sur la même machine).

Indication : vous devez exploiter (spécialiser) la méthode Object readResolve().

Modalités pratiques

Projet exécuté en *binôme*. Les binômes devront être constitués au sein d'un même groupe de TD. La composition des binômes devra être transmise par mail à l'adresse *mauran@enseeiht.fr*, pour le **vendredi 11 décembre avant midi**. Passé ce délai, les étudiants restant sans binôme seront constitués en binômes par tirage au sort. La liste des binômes sera disponible sur la page Moodle de l'UE.

Les fournitures et documents relatifs au projet sont disponibles sur la page Moodle de l'UE.

Le projet comporte quatre séances de suivi. Les séances de suivi sont obligatoires : un point d'avancement sera fait à chaque fois. Dans l'idéal, à titre indicatif :

- la séance de la semaine du 14 décembre portera sur la validation de la compréhension et de l'organisation du projet, ainsi que sur l'identification des problèmes à résoudre pour l'étape 1. Selon l'avancement, il sera aussi possible de discuter des idées d'architectures et solutions de principe.
- la séance de la semaine du 4 janvier devrait porter sur la présentation et la validation de l'étape 1, et permettre de discuter des solutions de principe de l'étape 2
- la séance de la semaine du 11 janvier devrait porter sur la présentation et la validation de l'étape 2, et aborder la conception des étapes 3 et 4
- la séance de la semaine du 18 janvier devrait porter sur la présentation et la validation de des étapes 3 et 4. Elle devrait identifier des difficultés restantes et présenter une proposition de plan de test.

Vous devez **rendre le 25 janvier 2016 avant midi** :

- 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.

Ces documents doivent être envoyés sous la forme d'une archive à l'adresse mail suivante : mauran@enseeiht.fr

Attention :

- **La date du lundi 25 janvier 2016 midi est une date ferme** : les documents remis après cette échéance ne seront pas pris en compte**.**
- 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.

Une **revue de projet**, par groupe de projet, aura lieu le **vendredi 29 janvier, entre 14 et 17h**.

- Cette revue durera de 10 à 15 minutes, et consistera en une brève présentation/démonstration (par les membres du groupe) du travail réalisé, suivie d'un retour et de questions sur le travail effectué.
- Les horaires de passage des différents groupes seront disponibles le mardi 26 janvier, sur la page Moodle de l'UE.