ENSEEIHT
IMA – 2ème Année

Rapport de projet
Intergiciels
Gestion d'objets dupliqués avec Java RMI
Janvier 2008

Introduction

Le but de ce projet est de mettre en place un service de gestion d'objets dupliquées en s'appuyant sur Java RMI pour la communication entre l'ensemble des objets Java répartis. Des applications Java peuvent choisir de partager un objet en l'enregistrant avec un nom unique auprès du serveur. Une application peut par la suite demander au serveur si tel objet a été partagé et le cas échéant en récupérer une copie.
Pour assurer la cohérence d'un tel système, nous utiliserons des méthodes de verrou (en écriture ou en lecture) et pour faire face aux problèmes inhérent à la communication à distance, nous aurons besoin de gérer la synchronisation entre toutes les méthodes.

1. Étape 1 :

1.1. Implémentation de la gestion des objets partagés :

1.1.1. Côté Client :

La classe `Client` permet de faire l'intermédiaire entre l'application et le Serveur. C'est elle qui stocke l'ensemble des objets partagés ou utilisés par l'application. Or, lors de l'enregistrement d'un objet partagé auprès du Serveur, celui-ci lui attribue un identifiant unique, nous avons donc choisi d'utiliser une `HashMap` pour stocker ces objets partagés et leur associer à chacun leur identifiant.

Une partie des méthodes de la classe Client est appelée par l'application et lui permet de créer un nouvel objet partagé (`create`), de l'enregistrer auprès du Serveur (`register`) et de rechercher un objet partagé au Serveur (`lookup`). Ces méthodes sont déclarées comme `static`. Ceci impose donc qu'a chaque application correspond une seule instance de `Client`, et donc que chaque application doit être lancée dans une JVM différente.
Les méthodes qui permettent de demander un verrou sur un objet partagé (`lock_read`) et (`lock_write`) sont appelées par le SharedObject. Des méthodes qui permettent de réduire un tel verrou (`reduce_lock`), et d'invalider un verrou en écriture (`invalidate_reader`) ou en écriture (`invalidate_writer`) peuvent être appelées par le Serveur.

Quant à la classe `SharedObject`, elle possède une référence vers l'instance de l'objet effectivement partagé et permet d'assurer sa cohérence par le biais de ses méthodes de demande / réduction / invalidation de verrou. Ces méthodes étant appelées directement par l'application.

1.1.2. Côté Serveur :

La classe `Server` permet à 2 applications de partager des objets en faisant le lien entre les Clients. A l'instar de la la classe `Client`, elle stocke l'ensemble des objets qui sont partagés à l'aide un `HashMap` permettant d'associer à chaque objet, son identifiant. Cependant, lors de l'enregistrement d'un objet auprès du Serveur, un nom unique est utilisé, nous avons donc ajouter une `HashMap` associant à chaque identifiant, le nom de l'objet partagé.

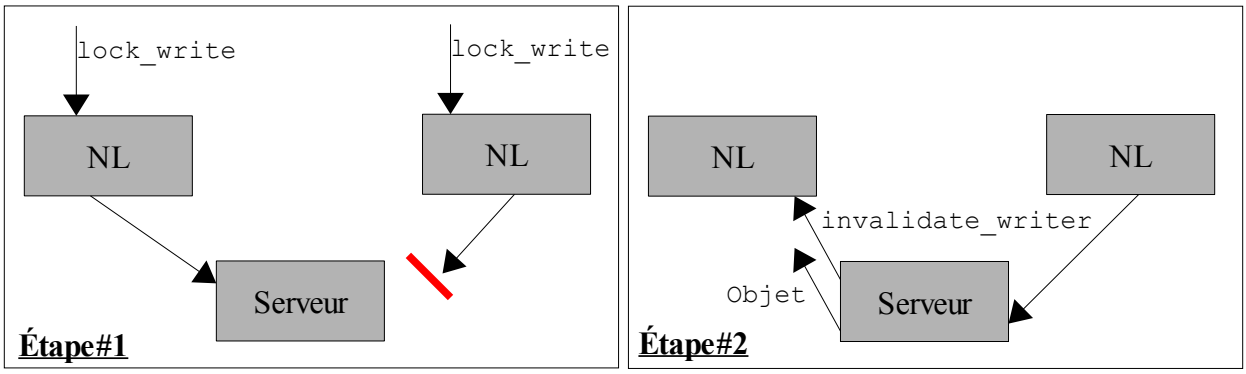
1.2. Les problèmes de synchronisation et leur résolution :

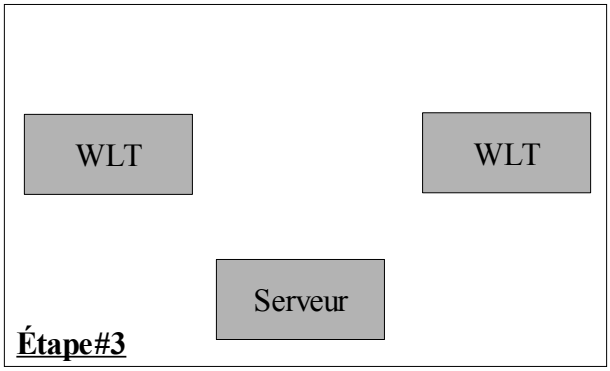
Les 2 catégories de problèmes sont les « deadlock » et les problèmes de cohérence sur les objets partagés.

- L'exemple du sujet

Le sujet fourni présente, dans le cas de figure 10, la première difficulté que nous avons rencontrée. Lorsque le client envoie une demande de verrou, et que le serveur lui envoie en même temps un message d'invalidation, les deux applications se mettent en attente d'une réponse de l'autre. Pour résoudre ce problème, la solution est de libérer le verrou sur le client lorsqu'il envoie une demande d'objet au serveur.

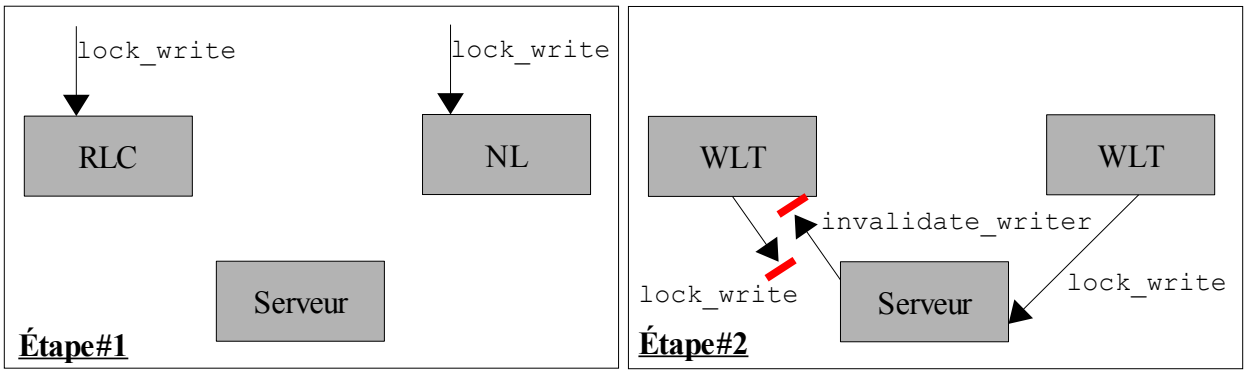
- Problème de cohérence : lorsque deux messages se croisent





Pour résoudre le problème précédent, nous avons dû relâcher la synchronisation sur le client lorsqu'il émet une demande d'objet. En conséquence, dans le cas de figure ci-dessus, il y a une perte de cohérence. Ici, deux écrivains demandent un verrou en même temps. La deuxième requête est mise en attente pendant que le serveur traite la première. Puis, pendant que l'objet est envoyé au premier client, la deuxième requête est traitée, et génère un envoie d'`invalidate_writer`. Si ce message d'invalidation arrive avant l'objet, le premier client sera invalidé alors qu'il n'a pas de verrou en écriture. Puis, lorsqu'il recevra l'objet, se mettra en WLT. Pour éviter ce problème, nous avons décidé que le client passerai en WLT avant d'envoyer la demande au serveur. Ainsi, le message d'invalidation sera mis en attente s'il arrive avant l'objet.

- Problème de verrouillage



L'implantation de la solution au problème précédent introduit un nouveau problème. Comme le montre le schéma ci-dessus, le fait de passer en WLT avant d'envoyer la demander d'objet implique que le client ne peut plus se faire invalider, même dans ce cas justifié. Nous avons donc modifié le code pour que le client puisse se faire invalider s'il est en WLT et qu'il reçoit un message `invalidate_reader`.

1.3. Les tests :

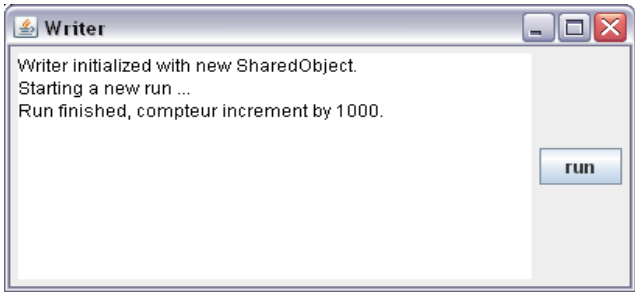
1.3.1. Le module IRC modifié :

Afin de tester notre projet, nous avons rapidement été amené à modifier le module `IRC` pour nous permettre de demander nous même les verrous, l'état de l'objet partagé, à l'aide de l'interface graphique. Nous avons donc complété l'interface graphique et les tests ont rapidement été concluants puisque cette application ne permet pas réellement de mettre en avant les problèmes de synchronisation.

1.3.2. Les classes de tests de la synchronisation :

Afin de mettre durement notre projet à l'épreuve, nous avons donc implémenter plusieurs classes. Le principe était de partager un unique compteur. Un classe `Writer`, réalise uniquement des écritures, elle incrémente simplement le compteur un certain nombre de fois. Un classe `Reader`, réalise uniquement des lectures de la valeur du compteur. Cette classe permettant de contrôler la valeur finale du compteur et de mettre en avant des conflits entre lectures et écritures concurrentes.

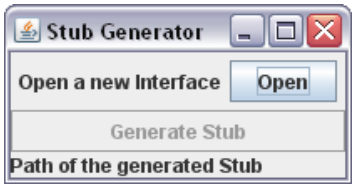
Entre chaque accès en écriture (respectivement lecture), le processus attend un temps aléatoire entre 0 et 50 millisecondes.
En lançant simultanément, plusieurs `Writers` et plusieurs `Readers`, nous avons pu avancer dans la résolution de nos problèmes de synchronisation pour finalement arriver à une version concluante.

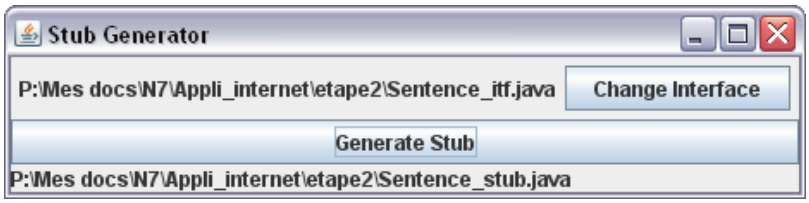


2. Étape 2 :

2.1. Le générateur de Stubs :

Nous avons choisi de réaliser une interface graphique permettant à l'utilisateur de choisir l'interface dont il veut générer le Stub en explorant ses fichiers. Après avoir choisi son fichier Interface, l'utilisateur peut donc générer le Stub correspondant. Des messages lui indiquent un éventuel problème (le fichier n'est pas une interface...).
Ensuite, la génération du Stub utilise le package `java.lang.reflect` pour déterminer le nom de la classe, et les méthodes à implémenter. Pour implémenter chacune des méthodes de l'interface, on récupère l'objet encapsulé dans le SharedObject et en le « castant » avec le nom de la classe. Puis on appelle la méthode correspondante sur cet objet.
Remarque : il faut regarder si la méthode est typée, et le cas échéant, retourner le résultat de l'appel.
Pour écrire le fichier Stub, nous avons utilisé un objet `PrintWriter`.





2.2. Les modifications des classes de l'étape 1 :

Grâce au `Stub` généré, l'application récupère directement un `Stub` par les méthodes `create` et `lookup` de `Client`. Nous avons donc modifier ces dernières. Pour la méthode `lookup`, le problème était que l'application donne simplement un nom d'objet partagé qu'elle recherche. Le Client demande donc lui même au Serveur, qui, s'il connaît cet objet, lui renvoie son identifiant unique. Le Client doit donc créer une instance du `Stub` correspondant, il doit donc connaître le nom de la classe de l'objet que l'application recherche. Pour se faire, nous avons choisi d'ajouter une méthode à l'interface du `Server` qui renvoie le nom d'un objet partagé en fonction de son identifiant.

2.3. Les tests :

Pour tester cette étape de notre projet, nous avons simplement utilisé le module `IRC` fourni et nos classes de tests de l'étape 1 en les adaptant.
Ainsi, nous avons dû ajouter une fonction à l'interface du `Server` permettant de connaître la classe d'un objet en fonction de son identifiant. En effet, dans la fonction `lookup` du `Client`, nous avions besoin de créer une instance du Stub correspondant à l'objet demandé par l'application, mais pour ce faire, nous avions besoin de u nom de la classe de cet objet.

3. Étape 3 :

3.1. Surcharge des primitives spécifiant la sérialisation / désérialisation :

Nous avions besoin de spécifier la désérialisation d'un `SharedObject` pour pouvoir conserver la cohérence du côté `Client`. En effet, pour chaque `SharedObject`, doit correspondre un unique `Stub`.
Ainsi, nous avons utiliser la surcharge de la méthode `readResolve()` dans la classe `SharedObject`, cette méthode étant appelée à chaque désérialisation d'un `SharedObject`.
Nous avons dû spécifier la désérialisation côté `Client` et côté `Server`, pour ce faire, nous avons utilisé un « état de sérialisation » au `SharedObject`, un Client ne communiquant qu'avec le `Server`, et le `Server` ne communiquant qu'avec des `Clients`, à chaque désérialisation, cet état change.
Ainsi, lors d'une désérialisation côté `Server`, nous ne faisons que modifier l'« état de sérialisation » du `SharedObject` ; et lors d'une désérialisation côté `Client`, nous vérifions si le `Client` possède déjà une copie locale de cet objet, si oui, un pointeur vers cet objet est renvoyé pour la désérialisation, sinon c'est un nouveau `SharedObject` avec le bon identifiant qui est retourné.
Une nouvelle méthode de la Classe `Client` est donc nécessaire.
Finalement, on assure bien la cohérence même avec des références à des objets partagés dans des objets partagés.

3.2. Les tests :

Nous avons enrichi nos classes de tests utilisant un compteur en ajoutant dans la classe `Compteur` une référence à un objet partagé, son historique. Lors de la création d'un nouveau `Compteur`, celui-ci crée un `SharedObject` pointant vers une nouvelle instance d'un objet `Historique`. Puis à chaque opération sur le compteur, l'historique est mis à jour.
Nous avons donc également ajouter une classe nommée `Spy`, qui utilise une interface graphique pour afficher l'historique complet du compteur, ou simplement la dernière opération grâce à des boutons.
Ainsi, nous avons pu vérifier que tout fonctionnait correctement.

Conclusion

Ce projet nous a permis d'appréhender des problèmes nouveaux, survenant lorsque l'on réalise un système qui ne s'exécute pas simplement en local. En effet, la majeure partie des problèmes auxquels nous avons dûs faire face étaient ceux liés à la synchronisation, le debuggage s'avéra d'autant plus délicat que lorsque plusieurs applications s'exécutent simultanément dans des JVM distincts, ils est très difficile de comprendre la chronologie réelle des événements.

Nous avons également découvert un nouveau domaine de Java qui est l'introspection. Nous avons trouvé son exploitation agréable et surprenante dans ses résultats, en permettant de traiter des problèmes de manière générique, comme nous avons pu nous en servir dans l'étape 2 de ce projet.

Nous avons globalement apprécié ce projet et trouvé que les séances de TP qui lui étaient consacré étaient une bonne idée dans la mesure où elles permettaient de faire le point sur l'avancement du projet et de discuter des problèmes que nous rencontrions avec les professeurs.

De plus, nous avons trouver agréable que ce projet fasse appel à ce que nous avions vu, à la fois en systèmes concurrents pour la gestion des accès concurrents, et en intergiciels, pour l'utilisation de Java RMI et la gestion de la cohérence.