Projet JAVA
Service de partage d'objets répartis et dupliqués

Adrian COURRÈGES
Mohamed Jalal BRICHA
2A IN Groupes C & D

Janvier 2008

Table des matières

1 Introduction

La gestion de la cohérence d'objets partagés est un problème récurrent que l'on retrouve au cours de divers développements logiciels. Il est d'usage d'autoriser un accès en lecture à plusieurs clients simultanément, mais de ne permettre l'écriture qu'à un unique client à un instant donné.

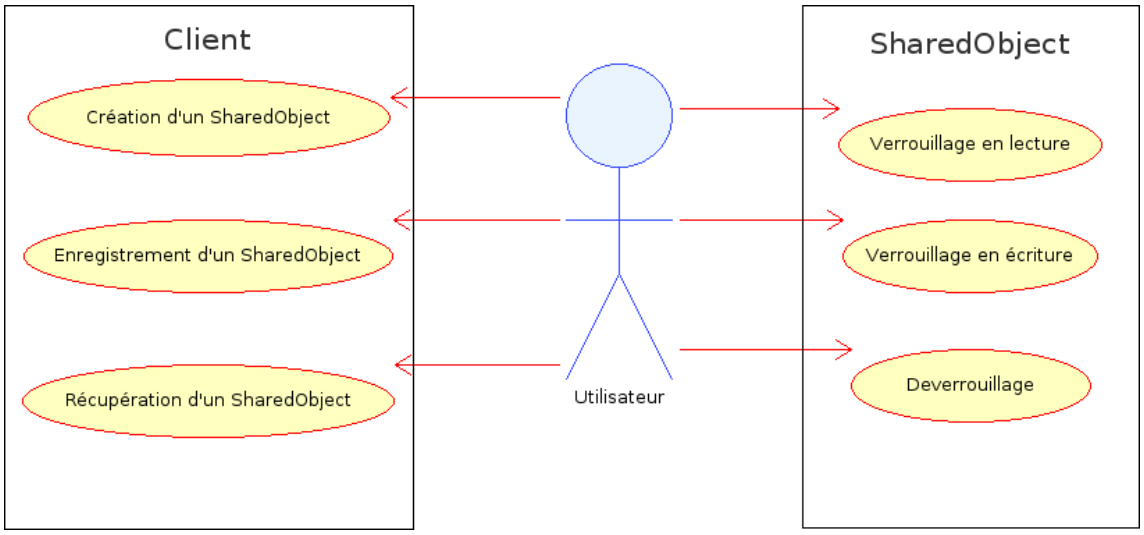
Le but de ce projet est d'élaborer un système de partage d'objets répartis en JAVA. Celui-ci doit permettre un accès concurrent aux ressources tout en garantissant leur cohérence. La communication entre nos différentes machines virtuelles sera effectuée grâce à un service de noms RMI.

Le développement d'un tel système nécessite un travail important de synchronisation entre les différentes entités. Il est notamment indispensable d'assurer la cohérence des copies locales des objets possédés par les clients.

1.1 Interface du service

Notre système doit permettre la création et la récupération d'objets partagés. Il doit en outre fournir, pour un objet donné, des méthodes de verrouillage en lecture, écriture, et de déverrouillage. Toutes les interactions de création ou d'enregistrement d'objets par l'utilisateur s'effectueront au travers d'une couche : le `Client`, propre à chaque application.
Ce `Client` possède une liste d'objets partagés connus de l'utilisateur : des `SharedObject`. Chaque `SharedObject` englobe le véritable objet que l'utilisateur a souhaité partager. C'est grâce au `SharedObject` que l'on appelle les méthodes de verrouillage.

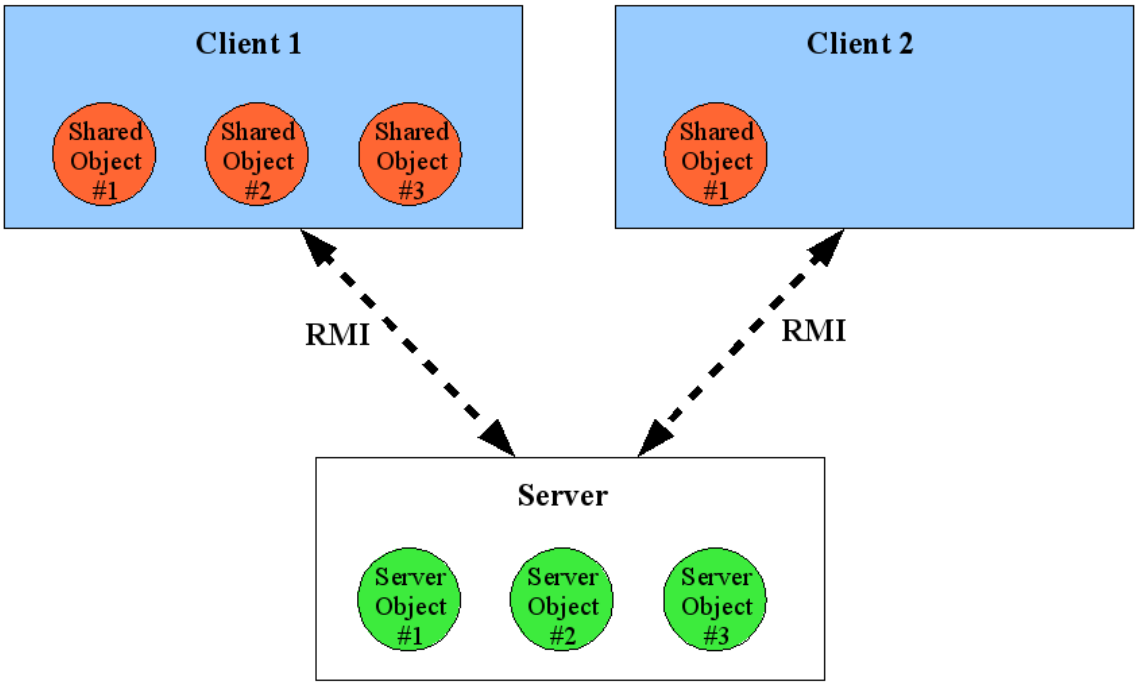
Voici un diagramme des cas d'utillisation pour une application accédant à notre service :



1.2 Structure du middleware

Notre modèle est centralisé autour d'une application serveur possédant sa propre machine virtuelle : le `Server`. C'est lui qui se charge d'initialiser le registre de noms RMI. Il contient également des `ServerObject` (l'équivalent des `SharedObject`, mais côté serveur) qui assurent une partie de la synchronisation.

Nous supposerons que chaque application possède une couche `Client` qui lui est propre, ainsi qu'une machine virtuelle dédiée.



2 Etape 1

La première étape consiste à développer notre système en laissant à l'utilisateur le soin de manipuler directement les `SharedObject`. Il s'agit donc d'implanter uniquement la gestion de la cohérence des objets partagés, et la synchronisation entre les clients.

2.1 Gestion de la cohérence

Un objet a le droit d'être lu simultanément par plusieurs clients, par contre l'accès en écriture doit être exclusif. Afin de savoir dans quel état se trouve un `SharedObject`, nous assignons à chacun une variable de verrouillage conformément aux notations du sujet (No Lock, Read Lock Cached...).

Voici une première version de la méthode `lock_write` de la classe SharedObject.

Cette version n'est pas encore correcte, car elle ne fait intervenir aucun mécanisme de synchronisation.

```
switch (this.lock) {
   case RLC : this.obj=Client.lock_write(this.id);
              this.lock = WLT;
              break;

   // RLT -> impossible

   case WLC : this.lock = WLT;
              break;

   // WLT,  RLT_WLC -> impossible

   case NL  : this.obj=Client.lock_write(this.id);
              this.lock = WLT;
              break;

   default : break;
}
```

La version ci-dessus de la méthode `lock_write` a été écrite de façon naturelle. Même si elle semble théoriquement correcte, sa mise en pratique ne fonctionne pas, car elle ne fait intervenir aucune synchronisation.

2.2 Gestion de la synchronisation

Afin d'éviter qu'une méthode d'un `SharedObject` ne soit interrompue au milieu de son exécution, ou que deux demandes de verrouillage ne se produisent simultanément, il a été nécessaire d'implanter de la synchronisation.

Une première partie de cette synchronisation consiste à mettre en attente des demandes d'invalidation (`invalidate_reader`, `invalidate_write`, `reduce_lock`) qui parviennent au `SharedObject` alors que celui-ci est encore verrouillé par l'utilisateur. Pour cela nous avons utilisé la méthode `wait` qui met un thread en attente. Lorsque la méthode `unlock` est invoquée par l'utilisateur, un appel à `notify` réveille le thread endormi.

Le mot-clé `synchronized` permet à une portion de code de se retrouver en exclusion mutuelle par rapport à une instance de sa classe.
Une première idée consiste à déclarer la totalité des méthodes du `Server`, des `ServerObject`, de `Client`, de `SharedObject`.
Malheureusement, on tombe très vite sur des cas d'interblocage dûs à un croisement de messages.

Il ne faut donc pas déclarer les méthodes `lock_read` et `lock_write` entièrement `synchronized` mais simplement en partie, afin de ne pas "conserver le verrou" lorsque l'on est mis en attente.

Nous décrivons dans la partie suivante certains problèmes de synchonisation qui nous ont forcé à adapter notre code.

3 Problèmes de synchronisation rencontrés

Dans cette section, nous ne nous intéresserons pas au cas classique du deadlock qui se produit lorsque toutes les méthodes sont déclarées `synchronized`. Le sujet décrit suffisamment bien ce cas particulier.

Par contre, nous allons présenter les autres problèmes de synchronisation qui sont survenus lorsque nous avons tenté d'obtenir des méthodes de lock qui ne seraient que *partiellement* `synchronized`.

3.1 Le cas du "Voleur de Verrou"

Ce cas concerne les méthodes `lock_read` et `lock_write` de notre classe `SharedObject`.

Afin d'éviter le deadlock décrit dans le sujet, nous avions dans un premier temps décidé de sortir l'appel à `Client.lock_read` de la section critique. Ainsi, la méthode pouvant être interrompue, on évitait tout risque d'interblocage.
Voici la méthode dans sa première version :

```
synchronized(this) {
   propagerAppel = faux ;
   faire le switch sur la valeur courante en mettant à jour propagerAppel ;
}

si (propagerAppel) {
   récupérer l'objet avec un Client.lock_read ; // peut être bloquant
   verrou = RLT ;
}
```

Le problème est qu'au moment même où l'on sort de la section critique, on donne à n'importe quel appel d'invalidation le droit de nous "voler le verrou".

Par exemple : entre la ligne de récupération de l'objet et celle de la mise à jour du verrou en RLT, il peut très bien survenir un `invalidate_reader` qui va rendre `null` le champ `obj` du `SharedObject`.

Notre `SharedObject` croit alors posséder un verrou de lecture alors qu'il vient de se le faire voler immédiatement après son obtention !
Qui plus est, on se retrouve avec un `SharedObject` en RLT, avec un champ `obj` qui est `null` (à cause de l'invalidation).

La solution consiste à effectuer des appels supplémentaires à `Client.lock_read` tant que l'on a subi des invalidations. Pour cela, nous avons besoin d'une sorte de `TestAndSet` atomique qui nous permet de faire passer notre `SharedObject` en RLT dès lors que l'on sait qu'aucune invalidation n'est survenue.
Voici la solution à laquelle nous avons abouti :

```
synchronized(this) {
   propagerAppel = faux ;
   faire le switch sur la valeur courante en mettant à jour propagerAppel ;
}

si ( ! propagerAppel ) {
   return ;
}

tant que (verrou != RLT) {
   récupérer l'objet avec Client.lock_read ;
   synchronized(this) {
      si (obj != null) { //il n'y a pas eu d'invalidation
         verrou = RLT ;
      }
   }
}
```

Les deux lignes de code rajoutées en section critique dans la boucle `tant que` constituent un test tout à fait efficace. En effet, en observant la valeur du champ `obj` on peut savoir si une invalidation est survenue. Si ce n'est pas le cas, on change (toujours atomiquement) le verrou en RLT. **A partir de ce moment-même, il n'est plus possible de subir de "vol de verrou"** : toutes les invalidations vont être mises en attente du `unlock`.

Bien sûr, tant que des invalidations surviennent, il faut refaire un appel `Client.lock_read`, avec le risque de se faire voler le verrou immédiatement après l'appel.

Il existe donc un risque de famine potentiel, cependant nos tests n'ont pas mis à jour de problème majeur à ce sujet.

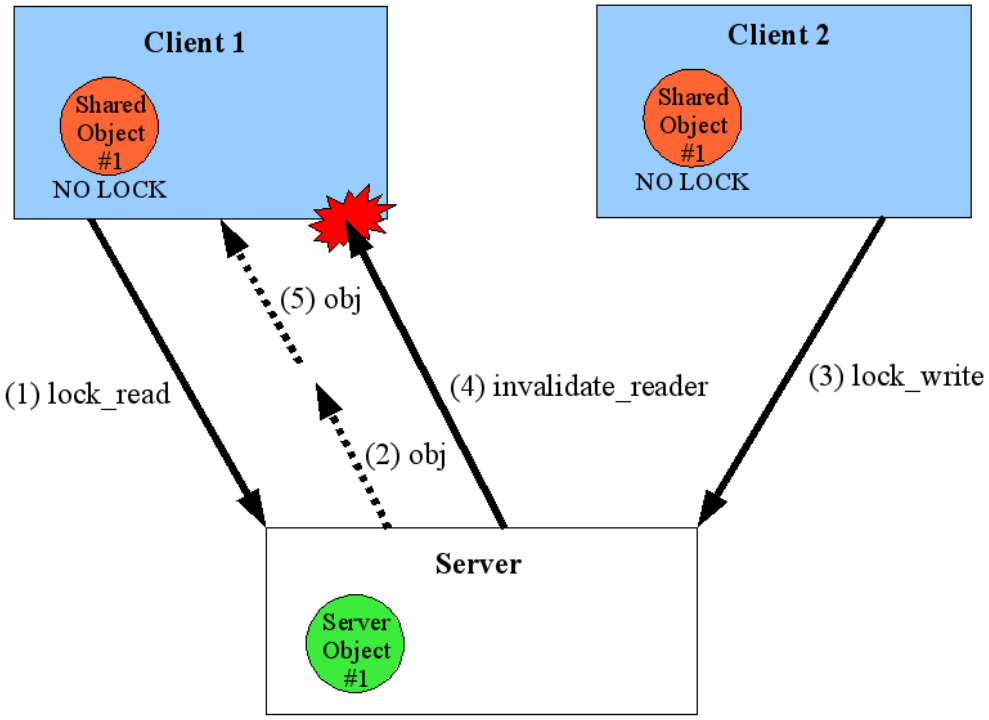
3.2 Le cas du "Quiproquo spatio-temporel"

Sous cette appellation se cache un problème d'ordre d'arrivée des messages qui transitent dans le RMI.
Cela peut conduire à un quiproquo temporaire entre l'univers du client, et celui du serveur. Par exemple, le serveur peut avoir accordé un verrou à un client, et avoir envoyé l'objet à retourner dans le RMI. Cet objet va mettre une certain durée à parvenir au `Client` (transit au sein de la liaison RMI).

C'est précisément pendant ce laps de temps que se crée une situation de quiproquo : du point de vue du `SharedObject` le verrou n'a pas été obtenu car on n'a pas reçu l'objet, mais du point de vue du serveur, comme l'objet a été envoyé, le `SharedObject` est considéré comme possesseur du verrou.

Ce quiproquo est temporellement très court, mais il est dans certains cas assez important pour faire tomber toute la synchronisation du système.

Voici un exemple concret pour lequel le problème survient.



Initialement, nos deux `Client` possèdent un même `SharedObject` en état `no lock`. Le premier `Client` va vouloir y accéder en lecture, et le second en écriture.

Voici le déroulement des actions :

- 1 : le `Client` 1 fait un `lock_read` sur le `Server` afin d'obtenir une copie de l'objet sur laquelle il aura un verrou en lecture.
- 2 : le `Server` traite la demande, accepte le verrouillage en lecture et envoie l'objet à retourner sur le service RMI pour qu'il soit acheminé au `Client` 1.

A ce stade précis, le message de retour est en transit dans le service RMI, il n'est pas parvenu à destination. Pour le serveur, le verrou a été accordé au `Client` 1. Pour le `Client` 1, le `SharedObject` n'a toujours pas été verrouillé et est encore en `no lock`.

- 3 : le `Client` 2 fait un `lock_write` sur le `Server` afin d'obtenir une copie de l'objet sur laquelle il aura un verrou en écriture.
- 4 : le `Server` traite la demande de `lock_write`, et envoie un message `invalidate_reader` au `Client` 1 (qui, du point de vue du `Server`, possède un verrou de lecture).

Le Client 1 reçoit alors un `invalidate_reader`, bien qu'il soit en état No Lock !
Un tel cas ne devait jamais se produire dans le modèle qui avait été élaboré à la base.

Ce phénomène pose problème car le `Server` pense avoir invalidé le `Client` 1, alors qu'il n'en est rien car le message de confirmation de `lock_read` est encore en transit et va finalement lui parvenir (étape 5).
Chaque Client va alors travailler en parallèle dans son coin, certain d'avoir obtenu son verrou.

Pour corriger ce problème, il convient de modifier le code des méthodes d'invalidation de `SharedObject` (c'est-à-dire `invalidate_reader`, `invalidate_reader` et `reduce_lock`) afin de les faire prendre en compte un tel phénomène.

Voici par exemple le code modifié de la méthode `invalidate_writer` de `SharedObject`.

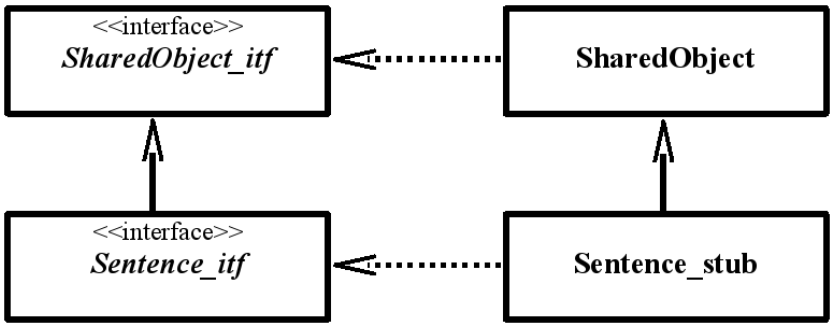
```
switch (lock) {
   case RLT : erreur ; break ; // impossible

   case WLC :
              lock = NL ;
              break ;

   case WLT :
   case RLT_WLC :
              while (lock != WLC) {
                 wait() ;
              }
              lock = NL ;
              break;

   case RLC :
   case NL  :
              // Le SharedObject a déjà fait un lock qui a
              // été accepté mais il ne le sait pas encore.
              while (lock != WLC) {
                 wait() ;
              }
              lock = NL ;
              break;

   default : break;
}
```

Une telle adaptation corrige le problème de synchronisation du "Quiproquo spatio-temporel".

4 Etape 2

Pour cette partie, le but est de soulager la tâche de l'utilisateur, en lui permettant de ne manipuler que le `SharedObject`. En plus des méthodes de verrouillage, l'utilisateur peut appeler directement les méthodes natives de l'objet sur le `SharedObject` qui l'englobe. Il n'y alors plus besoin de passer par le champ `obj`, toutes les méthodes sont automatiquement redirigés vers l'objet.

Pour cela, nous avons besoin de définir pour chaque classe, deux nouvelles classes correspondant à l'interface et au stub de la classe source.
Par exemple dans le cas de `Sentence.java` :



4.1 Générateur de stub

Nous avons réalisé une classe qui, à partir d'un fichier java d'une autre classe, va se charger d'écrire les fichiers `_itf.java` et `_stub.java`.
Nous utilisons des `PrintWriter` pour écrire le code.

Par exemple, si on s'intéresse à la génération du fichier `stub`, voilà la procédure suivie :

- Ecrire : `public class` nomDeLaClasse_stub `extends SharedObject implements` nomDeLaClasse_itf `, java.io.Serializable {`

Puis, pour chaque méthode définie dans le fichier d'origine :
- `public` typeDeRetour nomDeLaMéthode `(` [liste des paramètres] `) {`
- nomDeLaClasse `s =(`nomDeLaClasse`)obj ;`
- *si void est le type de retour* : `return`
- `s.`nomDeLaMéthode `(` [liste des paramètres] `) ; }`

Et on referme la première accolade : `}`

4.2 Modifications apportées au code

Il suffit de reprendre le code de l'étape 1 en y apportant de légères modifications.

Dans le cas du `Client`, voici quelques lignes de code modifiées de la méthode `create` :

```
//Ask the server for a new ID
int id = Client.server.create(o);
//We create an instance of the "stub" class
Class stubClass = Class.forName(o.getClass().getName() + "_stub");
so = (SharedObject)stubClass.newInstance();
so.setId(id);
```

La modification est similaire pour la méthode `lookup`. Il a simplement été nécessaire de modifier le type de retour de la méthode `lookup` du `Server`, afin qu'il retourne, en plus de l'identifiant unique du `SharedObject`, le nom de la classe de l'objet englobé.

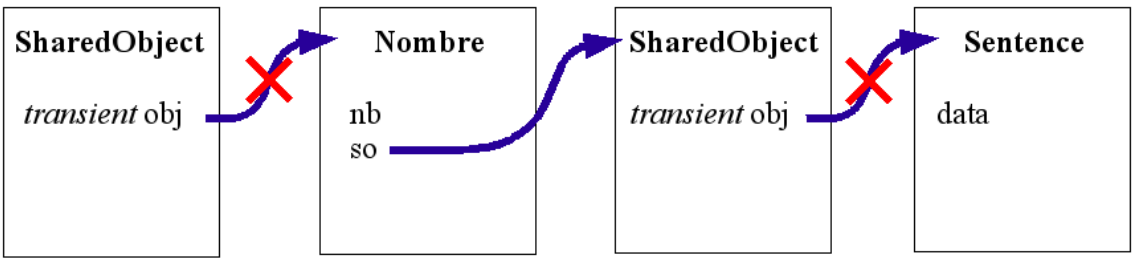
5 Etape 3

Il s'agit dans cette partie de prendre en compte la possibilité pour un objet partagé de stocker une référence vers un `SharedObject`.

Par exemple, créons une classe `Nombre` qui aurait pour attributs :
- `public int nb ;`
- `public SharedObject so ;`

Et instancions-la avec `Nombre(0,SharedSentence)` où `SharedSentence` est un `SharedObject` englobant un objet `Sentence`.

Nos différents objets vont donc avoir entre eux les relations suivantes :



Sur notre schéma, on peut remarquer que le champ `obj` de `SharedObject` est marqué `transient`, c'est-à-dire qu'il ne sera pas sérialisé lors de son passage dans le RMI. A l'arrivée, `obj` vaudra `null`.

```
// Shared object
public transient Object obj;
```

Mais il est également nécessaire d'avoir une cohérence des différents stubs au niveau de notre `Client`, tout en évitant d'avoir des stubs dupliqués ou absent.

Il est donc nécessaire d'agir à chaque fois qu'un `SharedObject` est désérialisé côté `Client`.

Pour cela nous allons utiliser la méthode `readResolve()` qui est invoqué automatiquement après chaque désérialisation d'un objet. C'est l'objet que cette méthode retourne qui sera considéré par le `Client` comme le `SharedObject` récupéré par RMI.
Afin de savoir si l'objet est désérialisé côté `Client` ou côté `Server`, nous avons rajouté un attribut de type booléen dans `SharedObject` qui est "inversé" à chaque désérialisation. Ainsi, on peut facilement savoir dans quel univers (`Client` ou `Server`) on se trouve.

Si on est côté `Server` : il n'y a rien à faire. Le serveur n'est pas concerné par les `SharedObject`, il ne possède que des `ServerObject`, et communique avec les clients à base de `int` ou de `Object`.

Si on est côté `Client` : dans le cas ou notre `Client` connaît déjà le `SharedObject`, on retourne directement une poignée vers le `SharedObject` déjà présent dans sa liste. Dans le cas contraire, on ajoute le nouveau `SharedObject` dans la liste, mais en état No Lock.

Voici le code de `readResolve` pour `SharedObject` :

```
public Object readResolve() throws java.io.ObjectStreamException {
   if (this.clientSide) {
      //we just got deserialized on Server
      this.setClientSide(false);
   } else {
      //we just got deserialized on Client
      this.setClientSide(true);
      if (Client.idObjects.containsKey(this.id)){
         //The SharedObject is already present
         return Client.idObjects.get(this.id);
      } else {
         //SharedObject is unknown : we create it (No Lock)
         this.lock = State.NL;
         Client.idObjects.put(this.id,this);
      }
   }
return this;
}
```

6 Programmes de test

Afin de vérifier que la synchronisation programmée était bien correcte, nous avons réalisé différents programmes de test.

Le premier consistait à avoir deux `Client` qui se partageaient un même objet `Sentence`, l'un essayant de lire en boucle, et l'autre d'y écrire. Un tel programme est basique et ne permet pas de détecter certains cas complexes. Il ne permet pas non plus de vérifier la cohérence de l'objet partagé (l'un ne faisant que lire et l'autre écrire). Il nous a cependant permis de prendre conscience de certains problèmes d'interblocage ou de `Null Pointer Exception`...

Notre second type de test consistait à lancer de nombreux `Client` en parallèle ayant le même

code d'exécution.
Chacun se partage un objet `Nombre` (contenant un `int`) et itère un certain nombre de fois une succession d'incrémentation de l'entier et de lecture. On peut alors vérifier facilement que l'objet est resté cohérent en comparant à la fin du test la valeur du nombre à celle attendue.
Un tel test nous a permis de traiter les cas du "Voleur de Verrou" et du "Quiproquo spatio-temporel".

En outre, notre modèle a résisté à 40 clients lancés en parallèle, essayant d'accéder en concurrence au même objet partagé.

7 Conclusion

Ce projet nous a permis de nous confronter concrètement et directement aux problèmes de synchronisation. Nous avons pu mettre en application des principes abordés dans le cours des systèmes concurrents, tout en nous familiarisant avec l'univers middleware de Java. Les étapes 2 et 3 nous ont poussé à utiliser des méthodes dont nous nous servions jusque là rarement, que ce soit les méthodes "d'introspection", de manipulation de classes, ou la méthode de désérialisation readResolve.

Nous avons également pu prendre conscience de toute la complexité du débugage d'une telle application : les différents états de notre système se sont montrés particulièrement difficiles à prévoir. Mais la difficulté relative du sujet ne l'a rendu que plus intéressant à réaliser.