Rapport de Projet Données Réparties

Un service de partage d'objets répartis et dupliqués en Java

Antoine Girard et Priscilia Gonthier
Groupe L2

Département Sciences du Numérique
2022-2023

Table des matières

Table des figures

Introduction

L'objectif principal de ce projet est réalisé en Java un service de partage d'objets par duplication reposant sur la cohérence à l'entrée. Les applications Java qui utilisent ce service peuvent accéder aux objets répartis et partagés efficacement puisque les accès sont majoritairement locaux. Pendant l'exécution, les applications communiquent au moyen de Java/RMI.

1 Architecture du système

1.1 Étape 1 :

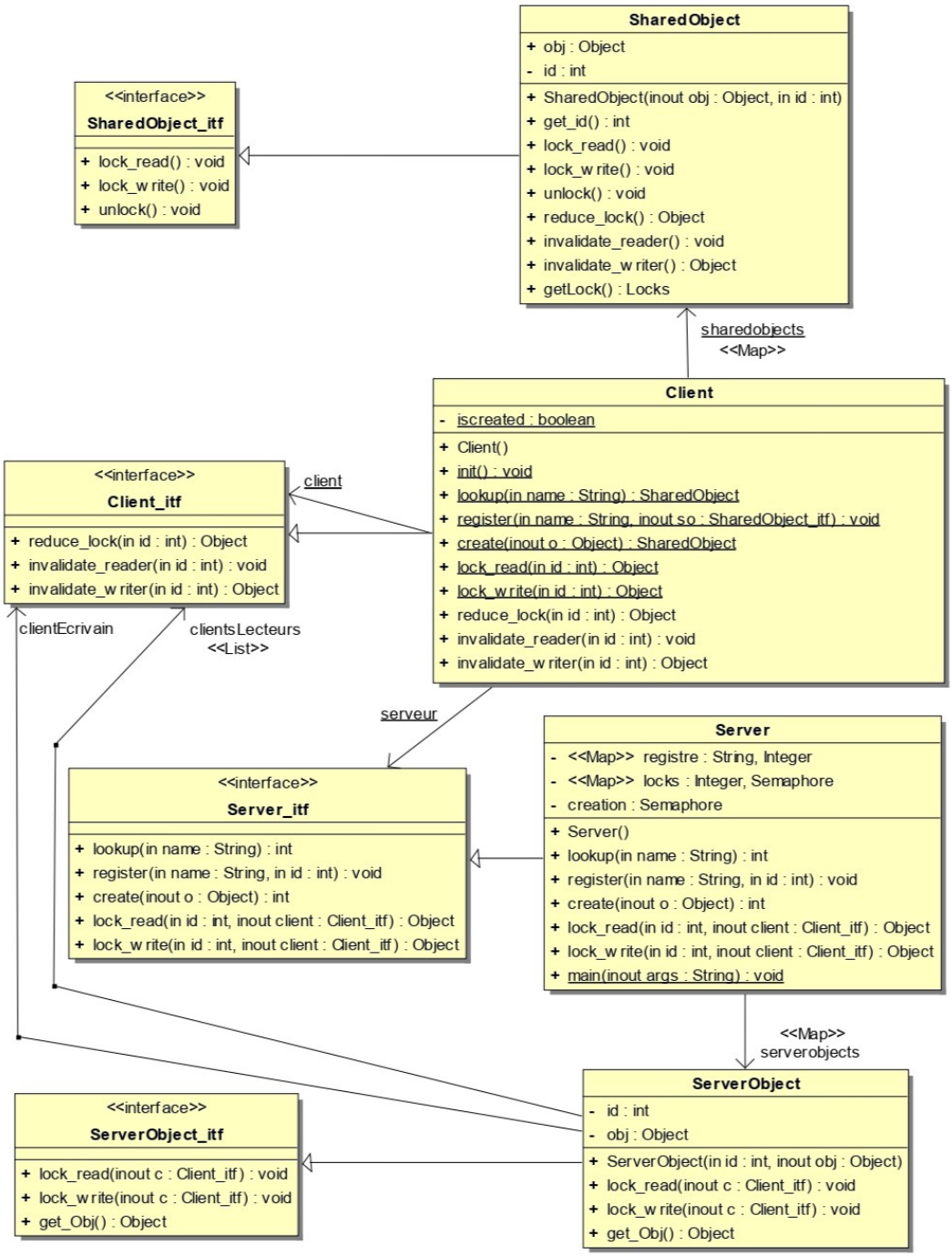

FIGURE 1 – Diagramme de classe de l'étape 1

Étape 1

Pour cette application, nous avons créé les classes Client, Server, SharedObject et ServerObject ainsi que leurs interfaces.

La classe Client a accès aux SharedObject de l'application qui l'a initialisé pour pouvoir transmettre les demandes de suppression ou de réduction de droit. La classe Client a en plus un accès au Server en Java/RMI.

Nous avons en plus de ça une classe Irc qui permet de tester manuellement que notre application fonctionne globalement. En effet, certains cas ne peuvent pas être traités par l'Irc. Il nous a donc fallu créer par nous-même des tests qui nous ont permis de tester d'autres aspects de la conception.

Étape 2

Pour l'étape 2, nous avons rajouté une classe CreateurClassStub.java qui permet de générer une classe de stub à partir d'une classe objet.

1.2 Étape 2 :

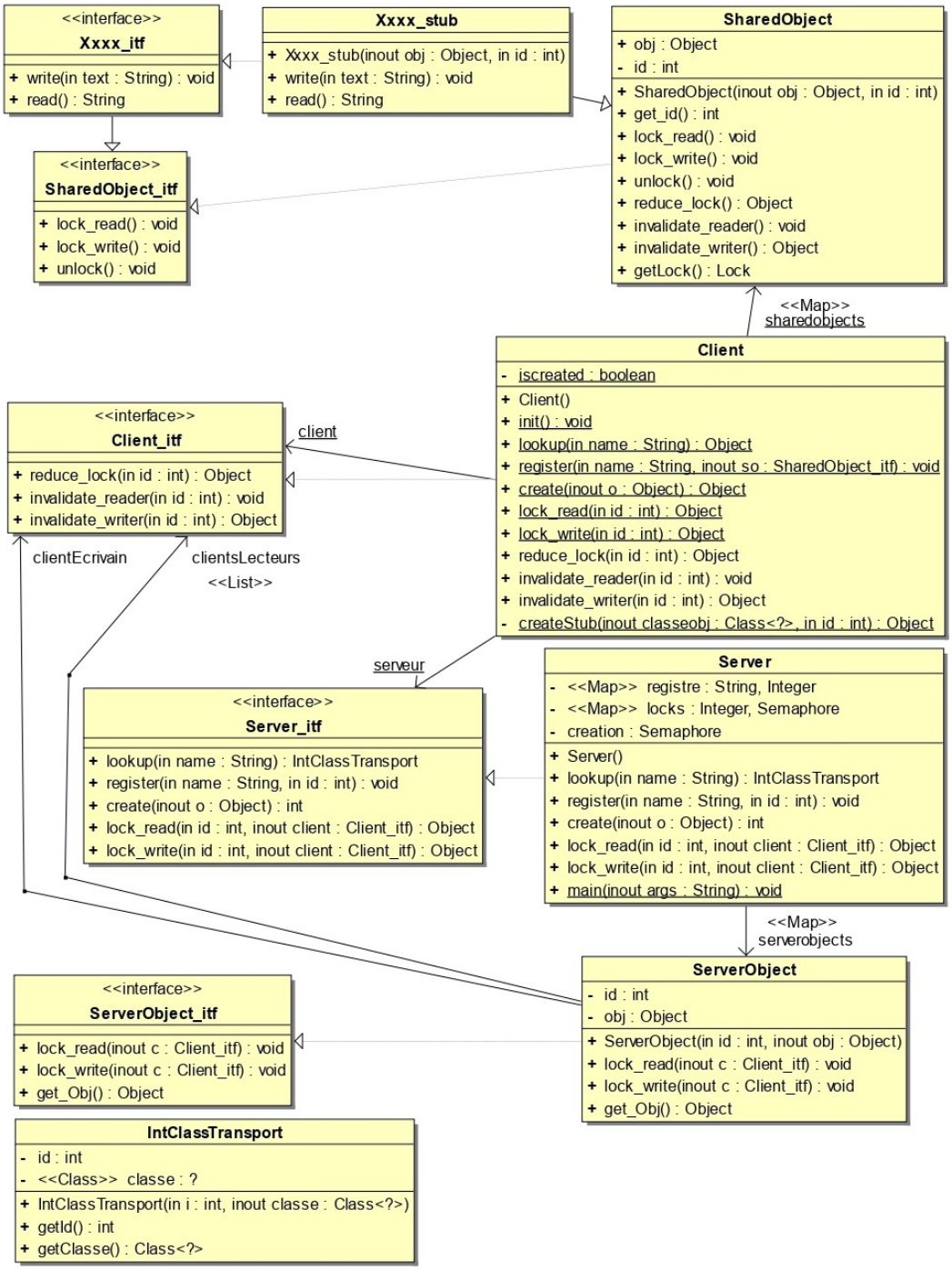

FIGURE 2 – Diagramme de classe de l'étape 2

2 Algorithmes des opérations essentielles

2.1 Connexion avec le Server

Le client et le server vont communiquer par RMI. Lors du lancement du Server il va créer un objet Server et un registry sur le port 2021. Il va ensuite utiliser la méthode statique de Naming pour binder l'objet server sur ce registry avec l'url *//localhost :2021/serveur*.

Le client quant à lui lors de l'init() va utiliser la méthode statique de Naming, lookup sur l'url *//localhost :2021/serveur* pour récupérer le server et ensuite pouvoir faire des appels aux méthodes de celui-ci.

2.2 Recherche et création d'un nouveau SharedObject

La première chose que va faire un utilisateur est de savoir si l'objet qu'il souhaite avoir existe déjà sur le Server (à partir du nom de l'objet).
Il va donc faire la requête Client.lookup(String nomObjet) pour obtenir le SharedObject s'il est déjà présent sur le Server, sinon la méthode lui retournera null et il fera donc les deux appels suivants successivement :

1. SharedObject so = **Client.create**(Object obj) ;
2. **Client.register**(String nom, SharedObject so) ;

La première commande va créer le ServerObject sur le Server et retourner le SharedObject associé (créé par le Client).
La seconde va donner un nom à l'objet qu'elle vient d'enregistrer pour faire en sorte que tous les utilisateurs puissent le retrouver et que l'on puisse gérer la synchronisation.

Nous avons rédigé un test qui créait beaucoup de thread à la fois et qui permettait de tester la robustesse de notre code face à un grand nombre de demandes de lookup/création à la fois et nous nous sommes rendu compte que l'on créait plusieurs fois le même objet avec des identifiants différents à chaque fois.
Nous en sommes donc venus à instancier un système de sémaphore pour régler ce problème, on traîte ce problème dans une section suivante. (cf : 3.3)

2.3 Synchronisation interne au SharedObject

Lorsque le SharedObject est en état de lecture ou d'écriture en cours (RLT, WLT, RLT_WLC) il faut attendre qu'il rendre le lock avant de pouvoir le réduire ou l'invalider. Cela évite qu'en plein milieu de l'utilisation de l'objet on lui enlève le lock.

Pour cela nous avons utilisé les méthodes **wait()** et **notify()**. Lorsque le SharedObject va recevoir un **invalidate_reader()**, un **invalidate_writer()** ou un **reduce_lock()** il va regarder quel est l'état de son lock. S'il n'est pas pris il va pouvoir continuer l'exécution des

méthodes, par contre si le lock est pris il va se mettre en attente (**wait()**) avant de pouvoir continuer.

C'est la méthode **unlock()** qui va indiquer la fin de la lecture ou écriture grâce au **notify()**.

2.4 Choix des stockages de SharedObject et de ServerObject

2.4.1 Les Clients

Sur Client nous avions à stocker les SharedObject et de les lier à leur identifiant pour pouvoir propager les demandes de **invalidateReader** / **invalidateWriter** / **reduceLock**. Pour ce faire nous avons choisi l'objet Map qui permet de lier une classe à une autre. Nous avons donc mis comme clé l'identifiant et comme élément le SharedObject associé :
Map<Integer, SharedObject> sharedobjects ;
On crée un nouvel objet dedans à chaque fois que l'utilisateur demande un objet qui n'y est pas encore (par create ou lookup).

2.4.2 Le Server

Il nous a fallu aussi 3 Map dans le Server pour :

1. Enregistrer et lier les ServerObject avec leur identifiant. (**Map<Integer, ServerObject> serverobjects**)
2. Enregistrer les noms associés à chaque ServerObject identifiés par leur identifiant. (**Map<String, Integer> registre**)
3. Enregistrer les sémaphores qui permettent d'éviter d'avoir plusieurs demandent en même temps sur le même ServerObject. (**Map<Integer, Semaphore> locks**)

On enregistre les ServerObject et leur identifiant à leur création (dans la méthode create du Server)
Les noms sont quant à eux enregistrés lors des appelles aux méthodes register du Server.
Pour ce qui est des sémaphores, on les crée et initialise lors des appels à la méthode create du Server. On parle plus précisément de ces sémaphores dans une partie suivante. (cf : 3.2)

2.4.3 Les ServerObject

Dans les ServerObject, il est uniquement question de stocker une liste des clients en lecture sur l'objet ou de stocker l'unique client lecteur. Nous avons donc créé :

1. Un attribut **Client_itf clientEcrivain** qui stockera le client en écriture s'il y en a un.
2. Une liste de clients lecteurs (**List<Client_itf> clientsLecteurs**)

Ces attributs nous permettent de faire les appels aux méthodes **invalidateWriter** / **invalidateReader** / **reduceLock** en fonction des demandent qui sont faites au ServerObject.

2.5 Création et utilisation des stubs de l'étape 2

Lors de la seconde étape, pour faciliter les appels aux méthodes pour les applications, le client ne va pas renvoyer directement un SharedObject mais plutôt un stub qui l'étend et qui va implémenter une interface définissant les méthodes de lecture et d'écriture. On parle plus précisément de la façon dont on génère la classe de ce stub dans une partie suivante. (cf : 3.4)

Ce qui change par rapport à l'étape précédente au niveau du Server c'est que lors du lookup au lieu de renvoyer seulement l'id de l'objet, il va aussi renvoyer la classe de celui-ci car le client a besoin de la classe de l'objet pour pouvoir générer le stub. Pour cela nous avons créé la classe IntClassTransport qui est sérialisable et va contenir l'attribut de classe et l'attribut d'id.

Dans le client nous avons créé une méthode **createStub** qui prend en paramètre la classe de l'objet pour lequel on crée un stub et son id et renvoie le stub créé. Il va tout d'abord récupérer la classe du stub correspondant à celle de l'objet. Si cette classe n'existe pas encore le client va appeler la méthode statique **createclass()** de **CreateurClassStub** afin de créer le fichier java définissant le stub. Il va ensuite compiler ce fichier et attendre la fin de la compilation pour charger la classe nouvellement créée dans le **ClassLoader**. A la fin on a donc accès à la classe du stub et on va pouvoir générer une instance de celui-ci qui va être retournée par la méthode.

La méthode **lookup** du client va donc récupérer la classe et l'id de l'objet depuis le server, créer le stub, l'enregistrer dans la liste des SharedObjects avant de renvoyer le stub à l'application (null si l'objet n'existe pas encore sur le Server).

La méthode **create** va quant à elle récupérer l'id depuis le server, récupérer la classe de l'objet donné en paramètre et de la même manière que pour **lookup** créer le stub, l'enregistrer dans la liste des SharedObjects avant de renvoyer le stub à l'application.

3 Points délicats et résolution

3.1 Envoi de la référence du Client au Server

Il est nécessaire de fournir une référence au Client lors de l'appel aux méthodes **lockRead** / **lockWrite** pour que le ServerObject puisse les stocker et faire par la suite des appels aux méthodes **invalidateReader** / **invalidateWriter** / **reduceLock** pour gérer les droits d'écriture et de lecture.

Nous ne savions pas quoi faire car pour le moment, les méthodes définient étaient static et ne demandait donc pas une instance du client. Nous avons donc pensé à une première solution :

Créer un nouveau client pour chaque appel aux méthodes lockRead et lockWrite qui est envoyé au Server. Bien que cette méthode nous ait donné une idée assez précise de comment faire par la suite, elle n'était clairement pas la bonne solution puisque cela créait trop d'objets

inutilement et donc était trop consommateur de ressources.

Ce pour quoi nous avons pensé à une autre solution qui est celle dont on se sert toujours : Créer un client lors de l'appel à **init** et on le stocke en attribut static de Client. Ainsi, chaque machine aura un client qui aura comme attribut un map de identifiants/sharedObject enregistré qui permettra de propager les reduceLock et invalidate.

Dans le cas où plusieurs applications seraient lancées sur la même machine (et donc plusieurs lancements de Client.init) nous avons créé un booléen static qui est à vrai si le Client a déjà été initialisé, faux sinon.

3.2 Eviter les demandes simultanées de lockWrite et réception de invalidateReader (cas n°11 du diapo)

Pour régler ce problème qui faisait que l'on pouvait avoir simultanément plusieurs objets en écriture, nous avons créé un système de verrous sur le Server permettant d'empêcher le fait qu'il y ait plusieurs demandes à la fois sur le même objet.

Dans un premier temps nous avions créé un seul verrou pour l'entièreté des demandes mais nous avons remarqué que cela restreignait inutilement les performances du Server. En effet, il n'y a aucun intérêt à bloquer une demande en lecture/ecriture sur un objet X s'il y a déjà une demande de lecture/ecriture en cours de traitement sur l'objet Y.

On est donc parti sur un map de identifiant(ServerObject)/lock et chaque demande de lecture ou d'écriture prendra le lock du ServerObject s'il est disponible, sinon il attendra que la demande déjà en cours soit finie.

Nous avons par la suite remarqué des problèmes avec les locks que nous utilisions (ReentrantLock). En effet, plusieurs tests menaient à la même conclusion : Il y a une probabilité non nulle qu'un lock soit pris plusieurs fois par deux personnes différentes s'il y avait plusieurs demandes au même moment. (notamment avec l'exception illegalMonitorStateException qui ressortait)

On a donc décidé de passer sur les sémaphores pour remplacer les verrous. On reste sur le même fonctionnement mais juste pas le même objet. Ce dernier fonctionne mieux dans notre cas et nous n'avons plus de problèmes.

3.3 Lookup et création d'objets concurrents

Nous avons rédigé un test qui créait beaucoup de thread à la fois et qui permettait de tester la robustesse de notre code face à un grand nombre de demandes de lookup/création à la fois et nous nous sommes rendu compte que l'on créait plusieurs fois le même objet avec des identifiants différents à chaque fois.

Pour régler ce problème, nous avons instancié un système de sémaphores qui permettait d'éviter la création de doublons. Pour ce faire nous avons créé un sémaphore unique (nommé creation) initialisé à 1 (comme un verrou) qui est utilisé comme suit sur le Server :

1. Prise du verrou au début de l'appel à lookup.
2. Si l'objet existe déjà :
   (a) on rend le verrou. Sinon on le garde.
3. Sinon :
   (a) L'utilisateur appelle la méthode create du Server. (on ne libère pas le verrou encore puisqu'il n'y a pas encore possibilité de trouver l'objet qui vient d'être créé)
   (b) L'utilisateur appelle la méthode register du Server. A la fin de cette méthode, on libère le verrou puisque l'objet est maintenant visible pour tous les autres utilisateurs lors du lookup.

3.4 Compréhension et création du générateur de stub

Nous avons eu des difficultés de compréhension au niveau de la création des stubs. Nous avions compris assez facilement que le stub devait être renvoyé à l'application à la place du SharedObject, cependant la façon de générer un stub restait très floue. Avec l'aide de notre professeur nous avons compris qu'il fallait créer une classe avec une méthode de création de la classe stub. Nous avons donc défini CreateurClassStub qui comprend une méthode statique createclass (appelée dans client) et un main pour générer la classe de stub depuis le terminal (**java CreateurClassStub** <**nomclass**>). On peut donc grace à cette classe générer un stub avant l'exécution du programme principal et aussi à la volée (si l'utilisateur à oublié de le générer avant).

Les étapes de la génération sont :

— Créer le fichier **nomclass_stub.java** qui contiendra la définition de la classe stub.

— Ecrire l'en-tête pour hériter de SharedObject et implémenter **nomclass_itf** et **Serialisable**.

— Ecrire le constructeur qui appelle le constructeur de la superclasse

— Ecrire les méthodes **write** et **read** définies dans **nomclass_itf**.

3.5 Etape 3

Nous avons essayé d'implémenter l'étape 3, et nous nous sommes frotté à des difficultés qui ne sont pas toutes résolues. La première a été au niveau de la compréhension. Il a été difficile de comprendre comment étaient reliés les différents SharedObjects. Nous ne sommes pas sûrs de nous mais au final nous avons créé une classe qui est presque identique à sentence sauf qu'elle contient un attribut suivant qui est le SharedObject chainé. Il a fallu ensuite récrire la méthode **readresolve()** dans SharedObject car c'est cet objet qui est sérialisé. Nous avons rencontré un problème au niveau du bind avec le server puisque la deserialisation du server utilisait notre nouvelle fonction **readresolve()**. Nous avons résolu ce problème en créant un

booléen qui indique si le server est déjà créé ou non. S'il n'est pas créé on renvoie juste le server sans faire autre chose.

Nous avons par la suite eu un problème que nous n'avons pas réussi à résoudre. C'est que lorsqu'un client crée un objet chainé si un autre client récupère cet objet il n'a pas accès au sharedobject suivant. En débuggant nous avons remarqué que la méthode readresolve() n'était en réalité pas appelée et nous ne comprenons pas pourquoi.

Pour nos tests sur cette étape nous avons créé IrcChaineCreateur et IrcChaineRecepteur. Le premier va créer un objet chainé et le second va le récupérer et chercher à avoir accès au sharedObject qui est en 2ème position de la chaine.

4 Exemples développés pour tester le système

4.1 IrcMatraquage

IrcMatraquage est une classe qui va envoyer successivement des demandes de lecture et d'écriture au server avec un temps aléatoire autour du temps donné en argument. Cela permet de tester les problèmes de synchronisation des SharedObject. Pour lancer le test, il faut lancer un server dans un terminal puis ensuite lancer au moins 2 IrcMatraquage dans des terminaux (**java IrcMatraquage <nom> <temps>**).

4.2 IrcMatraquagePlusieursObjets

IrcMatraquagePlusieursObjets va faire la même chose qu'au-dessus mais avec des objets différents pour pouvoir vérifier que tout se passe bien si on a plusieurs objets dans chaque application (**java IrcMatraquagePlusieursObjets <nom> <temps>**).

4.3 IrcMatraquageThread

L'objectif de ce test était de créer un certain nombre de processus demandant au Server si un objet existe déjà ou non.

Ceci permet de voir si on a plusieurs demandes simultanées de création d'un même objet si oui ou non le Server arrivera à gérer synchronisation pour éviter de créer plusieurs fois un objet.

Ceci a permis de nous rendre compte que l'on avait un problème de synchronisation que l'on a réglé avec des verrous.

Ce test ne permet pas de bien tourner sur le long terme, il est utile uniquement à la création car il met en place plusieurs applications pour un unique client. Ainsi, lorsque le Server invoquera une des méthodes **invalidateReader** / **invalidateWriter** / **reduceLock**, il fait un appel au Client qui va recherche un seul et unique SharedObject qui a le même

identifiant ! Or dans notre cas, il y a autant de sharedObject avec cet identifiant que de threads créés. Cela pose donc rapidement un problème.

On part donc du principe qu'il ne peut y avoir qu'une seule application par Machine donc une seule application par Client.

4.4 TestsEtape

TestsEtape1 et TestEtape2 permettent de tester les cas 1 à 10 du sujet respectivement pour les étapes 1 et 2. Afin de lancer les tests il faut lancer un server dans un terminal. Dans un nouveau terminal il faut lancer le premier test (java TestsEtape1 1) et ensuite dans un autre terminal on lance le second test (java TestsEtape1 2). Il faut ensuite suivre les indications dans les terminaux pour lancer les tests un à un.

Il est important de noter que ce test permet de tester les 10 premiers cas disponibles dans les slides du sujet en un seul fichier. Pour réaliser cela, nous avons dû faire un SharedObject par cas pour être sûr d'être dans les bonnes conditions pour le test côté Client et Server.

Conclusion

Avec ce projet, nous avons réussi à mieux maîtriser Java/RMI, nous avons bien mieux compris comment fonctionnaient les échanges entre les Clients et le Server et nous avons aussi mieux compris la manipulation des objets.

Nous avons rencontré de nombreux problèmes lors de la réalisation de ce projet mais nous avons réussi à résoudre une très grande partie d'entre eux.