PROJET : OBJETS DUPLIQUÉS
Rapport Final

Audric SCHILTKNECHT Denis VILLAND

Année 2009

Résumé

Réalisation d'un service d'objets dupliqués et répartis en *Java*.

Table des matières

1 Objectif **2**

2 Étape 1 **2**
2.1 Analyse 2
2.2 Réalisation 2
2.3 Tests 5

3 Étape 2 **6**
3.1 Analyse 6
3.2 Réalisation 6
3.3 Tests 8

4 Étape 3 **9**
4.1 Analyse 9
4.2 Réalisation 9
4.3 Tests 10

5 Conclusion **13**

1 Objectif

Le but de ce projet est d'implanter un service de partage d'objets dupliqués. Notre service mettra en œuvre un service de cohérence à l'entrée.

Le service sera implémenté au dessus de la couche RMI de `Java`. Nous utiliserons l'architecture décrite sur le schéma 1.

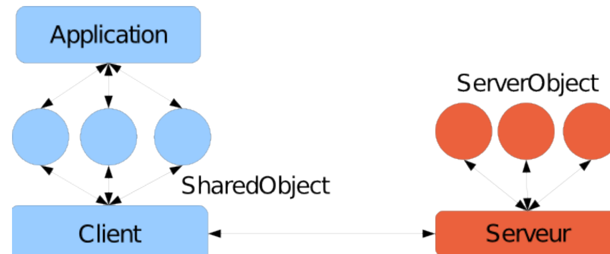

FIG. 1 – Architecture du service

On admettra qu'il ne peut y avoir qu'un seul couple (CLIENT,APPLICATION) par *JVM*.

2 Étape 1

2.1 Analyse

Cette étape nous demande de mettre en place la gestion du service d'objet répartis. Les difficultés qui jalonnent cette étape sont la mise en place de la cohérence de la réplication et la synchronisation des appels à notre service.

2.2 Réalisation

2.2.1 Client

Comme indiqué par le schéma de description de l'architecture (schéma 1), le CLIENT sert de relai entre les SHAREDOBJECT et le SERVER

Il aura donc une table **oids** du type HASHMAP<INTEGER,SHAREDOBJECT>, qui permettra de retrouver le SHAREDOBJECT associé à une valeur de l'identifiant. Il possèdera donc aussi une référence **server** vers le SERVER d'objets dupliqués. D'autre part, il faudra aussi stocker une référence d'une instance de CLIENT, référence qui sera transmise au SERVER pour les futurs appels de *callback* (`invalidate_reader`, etc.).

Le rôle du CLIENT est vraiment basique. Dans le cadre d'une communication :

SHAREDOBJECT → SERVER : Le CLIENT transmet l'appel au SERVER en utilisant sa référence **server**.

SERVER → SHAREDOBJECT : Le CLIENT va préalablement chercher dans la table **oids**, puis transmettre l'appel au SHAREDOBJECT idoine.

2.2.2 SharedObject

C'est cette classe qui a posé le plus de problèmes à la réalisation du service d'objets répartis, nottament du point de vue de la synchronisation. Étudions la en détail.

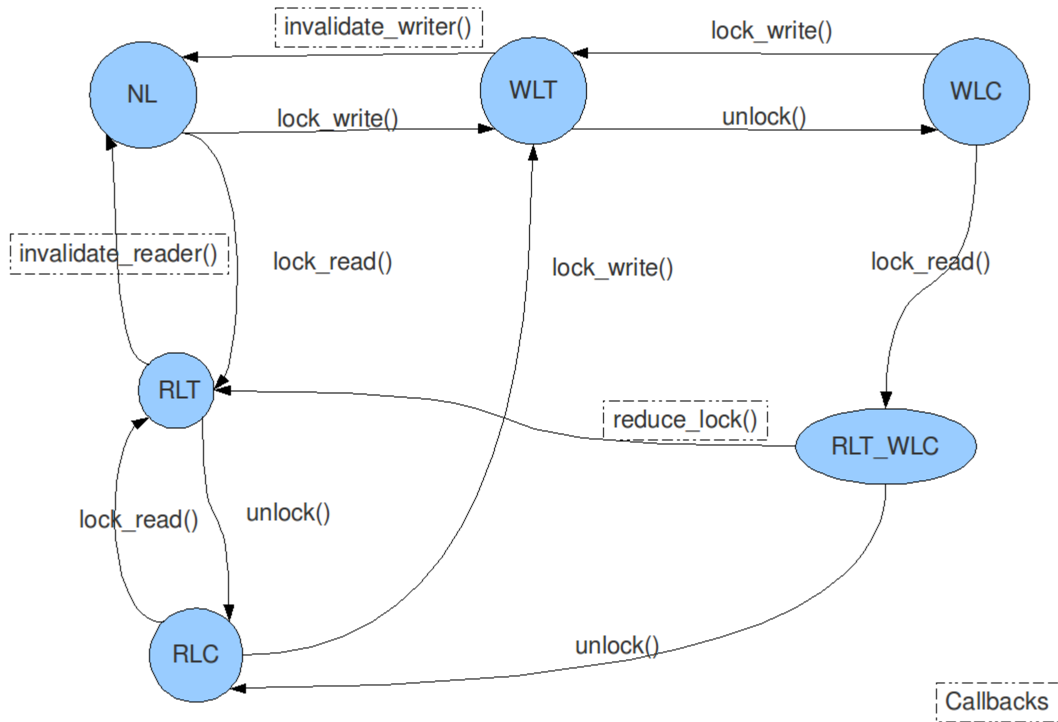

FIG. 2 – États d'un SHAREDOBJECT

Attributs : Voici la liste des attributs dont est doté un SHAREDOBJECT :

id : INTEGER L'identifiant (unique) du SHAREDOBJECT, fourni par le SERVER.

lock : LOCK Le mode de verrou pris sur l'objet. La classe LOCK est une énumération des divers modes possibles.

obj : OBJECT La référence vers le « véritable »objet.

A sa création, un SHAREDOBJECT possède un verrou en mode *NL*, et sa référence vers l'objet n'est pas initialisée. Elle le sera lors du premier appel à une fonction de verrouillage.

Fonctionnement : Le schéma 2 décrit sommairement le fonctionnement d'un SHAREDOBJECT sans se préoccuper (en premier lieu) d'une quelconque synchronisation.

Synchronisation : Voyons maintenant comment appliquer le schéma de synchronisation. Pour cela, les méthodes appellables par le SERVER seront passées en `synchronized`, de telle façon qu'un verrou sera pris lors de l'execution de ces méthodes.

Cependant, il peut y avoir un problème d'interblocage. Ce cas est décrit sur le schéma 3. Ce problème se produit car le client est en état **WLC**, donc le SERVER reste bloqué dans la procédure `invalidate_reader`, et garde donc son mutex associé.

La solution que nous avons trouvée est de « permettre » au SERVER de continuer son execution dans le cas d'un appel à la fonction `invalidate_reader` alors que le SHAREDOBJECT est dans l'état **WLT**.
Pour ce faire, il faut, dans la méthode SHAREDOBJECT.`lock _write` sortir l'appel à la méthode homonyme de la classe CLIENT du bloc `synchronized`. Ainsi, lorsque l'on devra appeller le SERVEUR, le *mutex* sera relâché sur le SHAREDOBJECT (même si cet appel sera bloqué en attente du *mutex* du SERVEUR). Cela permettra au SERVER d'effectuer son appel à `invalidate_writer` . Le SHAREDOBJECT se trouvera donc dans un état « non-conforme » (puisque dans un état

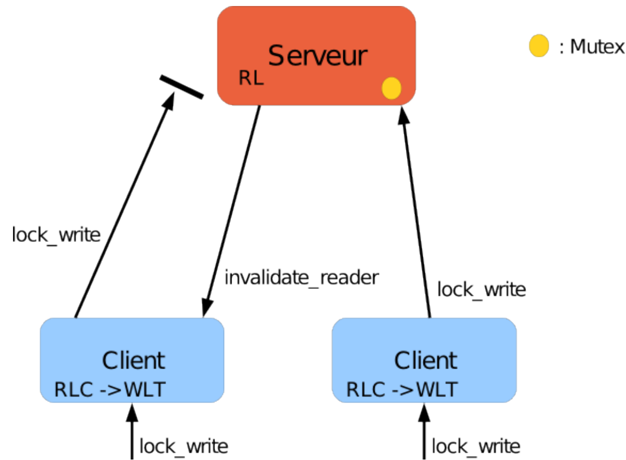

FIG. 3 – Cas d'interblocage

WLT, mais sans **obj** cohérent avec celui du SERVEUR), qui sera rétabli lorsque le CLIENT obtiendra l'accès au SERVER et que le verrou sera récupéré (de même que l'objet à jour).

2.2.3 Server

Le SERVER est semblable à la classe CLIENT : il s'occupe de transmettre les appels de fonction entre les SERVEROBJECT et le CLIENT. On ne s'étendra donc pas sur les points communs.

À l'instar du CLIENT, il possède une table **oids** du type HASHMAP<INTEGER, SERVEROBJECT>. La classe SERVER doit néanmoins réaliser un « serveur de nom », semblable à celui réalisé en RMI. Pour cela, elle sera aussi dotée d'une table **registry**, de type HASHMAP<STRING, SERVEROBJECT>.

2.2.4 ServerObject

Attributs : Les attributs du SERVEROBJECT permettent de représenter l'état de l'objet associé du point de vue de la concurrence. On dispose ainsi d'un pointeur **obj** vers l'objet en question, de son identifiant du point de vue du serveur et d'un attribut **lockMode** qui représente son état courant, à savoir lecture ou écriture (par défaut, nous avons initialisé **lockMode** à *READ*). Le SERVEROBJECT doit gérer la concurrence sur son objet référencé : il est donc nécessaire de connaitre à tout instant les lecteurs, ou l'écrivain, de l'objet. L'ensemble **readers** (nous avons choisi un HASHSET<CLIENT_ITF>) référence ainsi tous les CLIENTs ayant un verrou en lecture sur l'objet, et **writer** remplit le même rôle avec l'éventuel écrivain.

Méthodes : Les méthodes `lock_read` et `lock_write` prennent en compte les différents cas de figure présentés dans le sujet.

Pour le `lock_read` : s'il y a un écrivain en cours, on l'invalide. Le CLIENT demandeur est ensuite ajouté à la liste des lecteurs et l'objet est considéré comme « lu ». On est assuré que la version de l'objet est bien la dernière en date, le `reduce_lock` appelé sur l'écrivain retournant l'objet mis à jour. L'ancien écrivain éventuel conserve un droit de lecture sur l'objet : il est donc lui aussi ajouté à la liste des lecteurs.

Pour le `lock_write` : on demande l'invalidation de tous les lecteurs (ou de l'écrivain) courants. Nous n'avons pas explicitement distingué les cas où l'objet est en mode *READ*/*WRITE* :

s'il est en *READ*, il n'y aura pas d'écrivain à invalider, et vice-versa (Protocole 1 rédacteur - N lecteurs).

Le `lock_write` comme le `lock_read` peuvent faire l'objet de *deadlock*, les opérations d'invalidation étant bloquantes.

Un problème de synchronisation se posant potentiellement, les deux méthodes ont été définies comme `synchronized`. On a ainsi la garantie que seul un Client à la fois pourra faire une demande d'invalidation. Cela permet de résoudre en partie le cas bloquant où 2 Clients demandent le `lock_write` de manière conflictuelle (cas 10 du sujet, par exemple) : le premier « arrivé » demandera l'invalidation de tous les lecteurs (dont ne fait pas partie le deuxième demandeur...), et pourra entamer son écriture normalement. Le deuxième demandeur ne pourra prendre le verrou sur le ServerObject qu'une fois que le premier l'aura relâché, ne passant effectivement écrivain qu'à la fin de la tâche de son prédécesseur. En pratique, l'accès à un object `synchronized` est géré par la JVM), le deuxième demandeur pourra être amené à attendre plus longtemps (cf section 2.3)

2.3 Tests

2.3.1 Synchronisation

Cette première étape a pour objet de s'assurer du bon fonctionnement du projet en terme de synchronisation, et en particulier de l'absence d'interblocage (cas de figure 10 du sujet).

La classe Stresser implémente une « moulinette » de base : pour un nombre d'itérations donné par l'utilisateur, on réalise des demandes de verrou aléatoires en lecture ou écriture sur un SharedObject unique (ici, l'objet référencé est une Sentence). Ainsi, avec l'exécution de plusieurs Stressers sur un grand nombre d'itérations, on confronte notre programme à tous les cas de figure présentés dans le sujet.

Le test Stresser s'exécute sans erreurs, et toutes les instances créées finissent par terminer (il n'y a donc pas, à priori, d'interblocage). Nous avons par exemple, via un script, exécuté 50 Stressers réalisant 10000 itérations, sans encombres. Notons toutefois la gestion particulière de l'équité entre les Clients : lorsque plusieurs Clients sont en concurrence, ils réalisent des séries d'accès au SharedObject à tour de rôle. Un Client pourra en général effectuer une séquence d'accès avant d'avoir à céder la main à un autre. Néanmoins, ce comportement ne semble pas remettre en cause le bon fonctionnement du projet et peut être dû à la gestion interne de l'équité dans les méthodes `synchronized`.

Le test MultiObjectRopper a un fonctionnement sensiblement équivalent au test sus-traité, si ce n'est qu'il s'applique cette fois à plusieurs SharedObjects. En pratique, chaque instance de MultiObjectRopper accède à un tableau de SharedObjects, en choisit un aléatoirement et réalise, à l'instar du Stresser, une demande de verrou en lecture ou écriture. Comme son pendant à un seul SharedObject, ce test fonctionne correctement.

2.3.2 Absence de famine

Comme indiqué au paragraphe précédent, nos tests de base nous ont permis de constater qu'un Client conservait souvent la main sur un SharedObject pour un certain nombre d'accès. Ainsi, après avoir vérifié l'absence de cas d'interblocage, il nous a paru nécessaire de nous assurer qu'un Client ne pouvait pas s'accaparer un SharedObject et laisser les autres Clients en situation de famine.

La classe StarvationTest permet de s'en assurer. Ce test est prévu pour fonctionner avec un nombre n déterminé d'instances de StarvationTest, chacune étant dotée d'un numéro distinct (inférieur à n-1). Le SharedObject est un tableau de n entiers faisant office de compteurs sur les accès : chaque Client va faire un nombre donné d'appels en écriture sur le SharedObject, et incrémenter le compteur correspondant à son numéro dans le tableau. La classe StarvationMonitor affiche l'état des différents compteurs en cours d'exécution : on peut ainsi s'assurer que les accès au SharedObject sont équitablement accordés (et c'est effectivement le cas lors de nos tests).

2.3.3 Cohérence

Nos jeux de tests nous ont permis de tester le bon fonctionnement apparent de l'étape 1 de notre projet, avec un ou plusieurs objets partagés. Le test CoherencyWriter a pour objectif de vérifier la cohérence des accès aux SharedObjects afin de voir si, au delà des problèmes de famine et d'interblocage, il n'y a pas de prises illégales de verroux par des Clients.

Le test met en jeu un tableau de n (quelconques) SharedObjects référençants des entiers, auquel on adjoint un SharedObject de « contrôle » (**checkSum**). En pratique, on exécutera un nombre quelconque de CoherencyWriters qui répéteront indéfiniment l'opération suivante : prise de verrou en écriture sur un SharedObject du tableau aléatoirement choisi et sur **checkSum**, puis addition d'un même nombre aux entiers pointés par ces 2 SharedObjects, et enfin libération du verrou. Ainsi, si tout se passe normalement, l'entier référencé par **checkSum** est à tout instant la somme des entiers référencés par les SharedObjects du tableau. La classe CoherencyMonitor permet de faire cette vérification à intervalle régulier, en demandant des verrous en lecture sur tous les SharedObjects mis en jeu.

L'étape 1 de notre projet passe également ce test. A noter la présence d'un CoherencyNaughtyWriter, qui réalise sciemment des opérations non conformes sur les objets : dans ce cas, le test échoue, ce qui est normal.

3 Étape 2

3.1 Analyse

Objectif de cette étape : implémenter un générateur de stubs, afin que les applications n'aient plus à manipuler de SharedObjects directement, en passant par les stubs associés à la place.

Prenons l'exemple de l'objet à partager Sentence. Le diagramme 4 illustre la relation entre son stub associé (Sentence_stub) et les différentes autres classes entrant en jeu.

Ainsi, on se munit d'une interface Sentence_itf, que n'implémente pas Sentence, qui dispose des méthodes de Sentence et hérite, en plus, de SharedObject_itf et donc de ses méthodes de verrouillages. A partir de l'étape 2, l'utilisateur pourra se contenter de manipuler des Sentence_itf et n'aura plus à passer par les SharedObects. Le but du générateur de stubs sera de fournir une implémentation de Sentence_itf : Sentence_stub, qui héritera en plus de SharedObject.

3.2 Réalisation

Aucun changement n'a été effectué en terme de synchronisation : on ne reviendra donc que sur les nouvelles classes et sur celles ayant fait l'objet de modifications pour prendre en compte

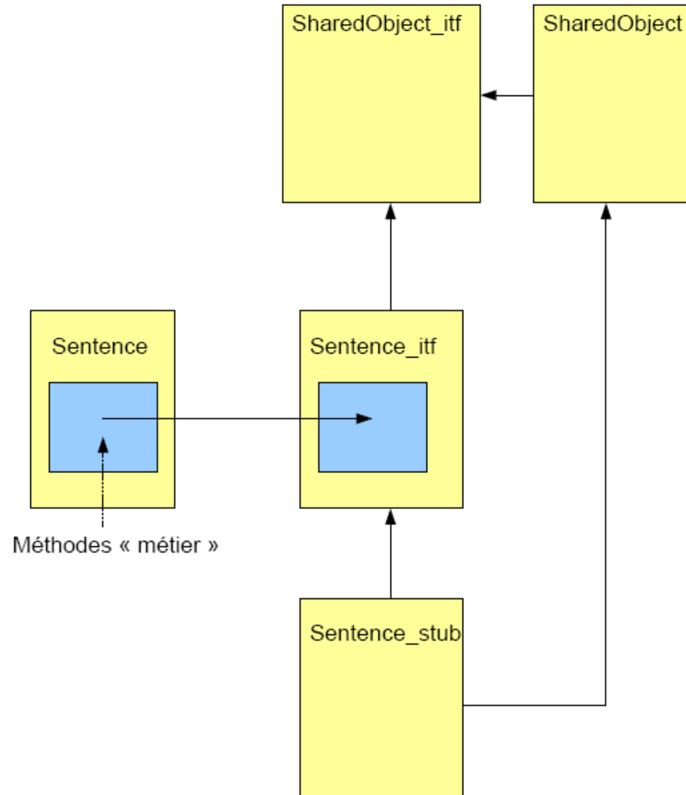

FIG. 4 – La classe _stub : exemple de Sentence_stub

le mécanisme de stubs.

3.2.1 StubGenerator

Dans un premier temps, nous avons codé une classe générant un fichier *.java*, conforme à l'utilisation en temps que stub. Ce fichier devra ensuite être compilé « à la main »pour pouvoir être utilisé.

Pour cela, nous nous appuyons sur les méthodes fournies par le `package java.lang.reflect`. Il fourni des méthodes permettant de réaliser l'introspection d'une classe java. Ainsi, en passant en paramètre de notre méthode principale le nom de la classe (qui doit donc se trouver dans le `classpath`), on va récupérer la liste des méthodes qui y sont définies.
Pour chacune de ces méthodes, on appliquera le même schéma : on va récupérer le champ **obj** du SHAREDOBJECT, et on appliquera sur cet objet (grâce à un transtypage approprié) la méthode demandée.

3.2.2 AwesomeStubGenerator

Par la suite, nous avons voulu simplifier l'utilisation du service en tentant de générer le *stub* à l'exécution. Après quelques recherches, nous avons découvert une API nommée JAVASSIST[1], qui permet la création et la modification de classes au runtime. Comme cela semblait intéressant, nous nous sommes lancé dans cette tâche.

Au final, l'API proposée par les concepteurs de JAVASSIST a permis d'utiliser notre code quasiment tel quel. En effet, il a simplement fallu étudier comment l'on pouvait créer une

[1]http ://www.csg.is.titech.ac.jp/˜chiba/javassist/

classe *ex-nihilo*, et définir les interfaces qu'elle implémente, ainsi que les classes dont elle hérite. Une fois cette étape franchie, il ne suffit plus que de rajouter des méthodes, le corps de ces dernière étant déjà obtenu suite au travail pour réaliser la génération de stub de la classe StubGenerator.

3.2.3 Client

Client se voit munir de 2 nouvelles méthodes :

getStub : Cette méthode permet de récupérer le stub associé à un objet s'il existe, et de le créer le cas échéant. En paramètre, on retrouve le nom de l'objet sous forme d'un String et son identifiant. Le Client va consulter sa table de SharedObject : si l'identifiant demandé y figure, on retourne le SharedObject associé (en fait : un stub) . Dans le cas contraire, le Client fait appel à sa méthode `createStub`, décrite ci-dessous, pour récupérer un stub sur l'objet, qu'il ajoutera à sa table sous l'identifiant demandé.

createStub : Étant donnés le nom de type « réel » d'un objet (par exemple : Sentence, pas Sentence_stub...) et son identifiant, cette méthode va générer la classe _stub associée[2] via un appel au `generate` du AwesomeStubGenerator, puis en créer une nouvelle instance. C'est cette instance, le stub vers l'objet, qui est retournée.

Notons par ailleurs des modifications dans les méthodes suivantes, nécessaires pour adapter le projet à l'utilisation des stubs :

Dans la méthode `lookup`, l'appel au `lookup` du Server renvoie à présent l'identifiant de l'objet demandé au serveur et une chaine de caractères représentant son type. Ainsi, là où on créait un nouveau SharedObject à l'étape 1, on appelle maintenant `getStub` pour créer un stub sur l'objet et l'enregistrer dans la table du Client. Il en va de même dans la méthode `create` : au lieu de créer un SharedObject à partir de l'objet passé en paramètre puis de l'enregister, on fait un appel à `getStub`.

3.2.4 Server

Seule modification apportée au Server : le méthode `lookup` ne renvoie plus un `int` mais un Identity, classe que nous avons définie. Un Identity comporte l'identifiant (entier) de l'objet, mais aussi un String représentant son type. L'interface Server_itf a été modifiée en conséquence. Ce changement était nécessaire car le Client a besoin de l'identifiant et du type de l'objet demandé pour en créer un stub.

3.3 Tests

Nous n'avons pas réalisé de tests particuliers pour cette deuxième étape : on constate que le StubGenerator génère bien un fichier .java contenant le code attendu dans le cas, par exemple, de Sentence. De même, le AwesomeStubGenerator génère des fichiers .class à l'emplacement attendu. Par ailleurs, aucun changement n'ayant été apporté à la synchronisation, le fonctionnement général du projet reste le même.

[2] En fait, on vérifie que le stub n'a pas été préalablement créé

4 Étape 3

4.1 Analyse

Objectif de cette étape 3 : être capable de gérer le passage de références à des SharedObject dans un autre SharedObject. Si on s'en tient à notre étape 2, un SharedObject pointé par un autre SharedObject sera en effet inutilement copié lors des échanges de verrous sur son « réferant »Ċoncrètement : une machine M1 demandant un verrou sur un SharedObject O1 dont le verrou est détenu par M2 recevra une copie de l'objet référencé par O1. Si cet objet est lui même un SharedObject O2, M1 se retrouvera donc potentiellement avec une copie du stub de O2 sur la machine M2. On voudrait que O1 soit déserialisé de manière cohérente sur M1, à savoir qu'O1 pointe après déserialisation sur le stub de O2 sur M1.

4.2 Réalisation

Comme suggéré par le sujet, la résolution du problème est passée par une spécialisation du `readResolve` et, dans notre cas, du `writeReplace` des SharedObjects. Là encore, nous n'avons effectué aucun changement en terme de synchronisation générale, et ne nous reviendrons donc que sur les changements effectués et les nouvelles classes.

4.2.1 Classe Util

Dans la résolution du problème telle que nous l'avons envisagée, il est nécessaire de connaitre l'environnement à partir duquel s'effectue la sérialisation/déserialisation du SharedObject. En effet, les considérations de cohérence des stubs n'interviennent qu'au niveau de l'utilisateur (donc : couche Client). Le Server n'a pas à se préoccuper des objets qu'il manipule, et cette distinction Client/Server devait pouvoir être faite.

C'est l'objet de cette nouvelle classe Util : permettre de savoir si on se trouve au niveau du Client ou du Server. On trouve donc un attribut statique **typeOfService** de type *TOS* (soit : CLIENT ou SERVER). Ainsi, on créera un Util « SERVER » à la création d'un Server, et un Util « CLIENT » à l'initialisation d'un Client. On pourra donc, à partir d'un SharedObject, savoir dans quel type de service on se trouve à un moment donné (cf ci-après).

4.2.2 SharedObject

Les seules modifications que nous avons apportées au SharedObject sont une spécialisation des méthodes `writeReplace` et `readResolve` :

writeReplace : L'intérêt d'une spécialisation du `writeReplace` est d'éviter la transmission superflue d'objets lors de la sérialisation d'un SharedObject. En effet, sans modification, un SharedObject serait sérialisé avec son objet pointé, ce qui est inutile : le nom du SharedObject et son id suffisent pour construire le stub associé côté « récepteur » , l'état de l'objet pointé étant géré par la suite lors des prises de verrous.

Le `writeReplace` modifié va donc remplacer le SharedObject à sérialiser par un stub créé pour l'occasion via un appel au `createStub` du Client, décrit à l'étape 2. Ainsi, on ne transmettra que les informations nécessaires : puisque le stub est nouvellement créé, son champ **obj** est `null`, ce qui est le but recherché. En revanche, on conserve l'**id**, et bien sûr les

informations sur le nom, du SharedObject d'origine : la reconstruction sera possible côté récepteur, via `readResolve`. Ceci ne sera fait que si on se trouve côté Client, ce dont on s'assure grâce à un Util, défini précédemment. Dans le cas du Server, on retourne l'objet en l'état.

readResolve : C'est le `readResolve` qui va se charger de rêgler les problèmes de cohérence des stubs. Pour répondre au problème posé par l'étape 3, sa seule spécialisation suffisait, le `writeReplace` ne faisant qu'améliorer le procédé.

Le fonctionnement est symétrique à celui du `writeReplace`. Si on se trouve côté Server, là encore, on ne fait rien. Côté Client : l'objet à déserialiser est un stub, généré depuis la machine expéditrice. On dispose de son nom et de son identifiant, ce qui est suffisant pour récupérer un stub se référant au même objet mais sur la machine réceptrice, via un appel au `getStub` du Client. Ainsi, si un stub sur l'objet voulu existe déjà sur la machine, on le retourne. Sinon, on en crée un nouveau et on l'enregistre dans la table du Client. Dans tous les cas, le problème de cohérence du stub ne se pose plus : les références à des stubs de la machine émettrice sont remplacés par des références à des stubs de la machine réceptrice.

4.3 Tests

4.3.1 Référence simple

Ce premier test visait à tester le fonctionnement "de base" de l'étape 3 : bonne sérialisation/déserialisation d'un SharedObject passé en référence dans un autre SharedObject. Nous avons pour cela créé une classe StoreReference : il s'agit simplement d'une référence à un SharedObject_itf (attribut **ref**), avec les accesseurs ad-hoc.

Le test associé s'effectue en 2 temps : on commence par lancer un TestReferenceInit. On crée ainsi un objet partagé sur une Sentence et un StoreReference pointant sur cet objet. On va ainsi créer et enregistrer deux objets partagés sur le serveur. Après une première modification sur la Sentence, le TestReferenceInit se met en pause quelques secondes (`Thread.sleep(5000)` : suffisant pour lancer un TestReference en parallèle, qui va demander un verrou sur le StoreReference. On constate ainsi qu'un Sentence_stub est créé côté TestReference, alors même que la Sentence n'avait pas été enregistrée de son côté au préalable. Le TestReference modifie la Sentence à son tour, et le TestReferenceInit le constate en se « réveillant » : l'ensemble est cohérent.

4.3.2 Référence double

Le principe est le même que le précédent, mais avec des références doubles permettant des tests plus poussés en matière de cohérence (Classe StoreDoubleReference). Là encore, on dispose d'un TestDoubleReferenceInit à exécuter préalablement à un TestDoubleReference. Le principe est d'inverser les références du StoreDoubleReference) en cours d'exécution, et de vérifier que tout reste cohérent (cf schémas ci dessus).

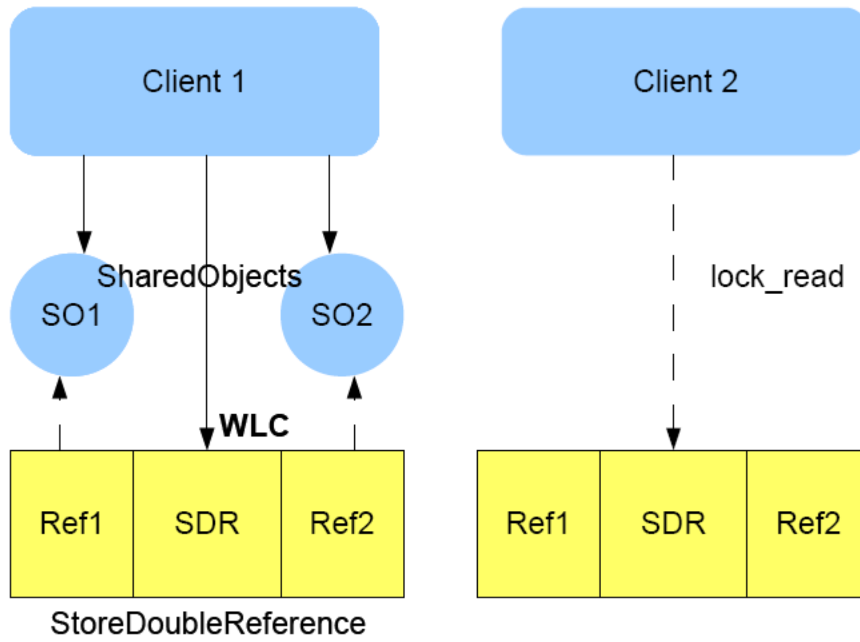

Fig. 5 – Étape 1 : Initialisation par le Client1

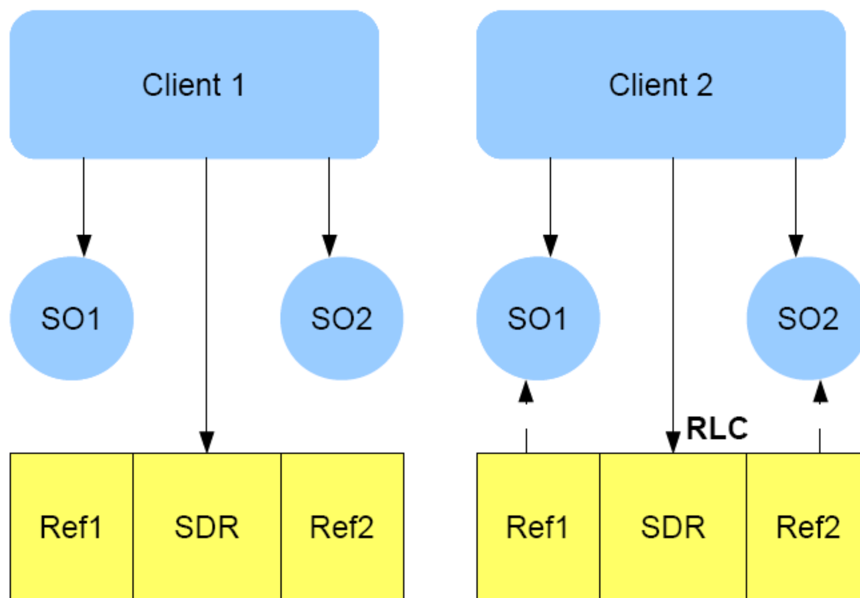

Fig. 6 – Étape 2 : Déserialisation côté Client2

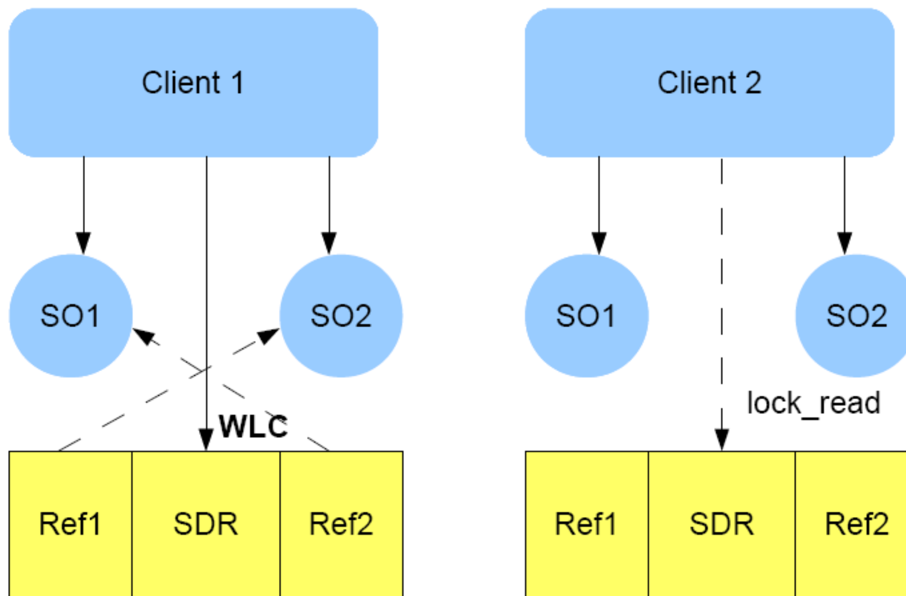

Fig. 7 – Étape 3 : Inversion des références côté Client1...

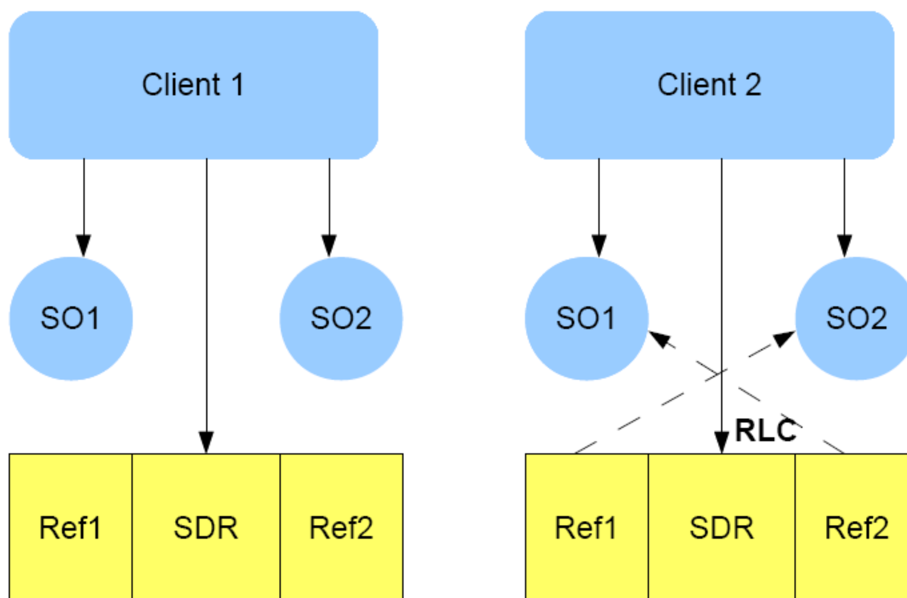

Fig. 8 – Étape 4 : ...bien répercutées côté Client 2.

5 Conclusion

Au final, ce projet nous a paru extrêmement intéressant. Il mélange en effet diverses notions que nous avons étudiées pendant l'année pour proposer la réalisation d'un service concret.

Mais cela a été aussi l'occasion de découvrir d'autres possibilités de *Java*, telles l'introspection, puissant outil d'analyse de classe lors de l'exécution, ou de nous amuser avec la librairie JAVASSIST.