Rapport de Projet Données Réparties
Service de partage d'objets répartis et dupliqués en Java

Julie PAUL et Héloïse LAFARGUE

Département Sciences du Numérique - 2A SN Systèmes logiciels
Systèmes Concurrents et Intergiciels
2022-2023

Table des matières

Table des figures

1 Introduction

Ce projet consiste à réaliser un service de partage d'objets par duplication sur Java, reposant sur la cohérence à l'entrée.

Les applications Java utilisant ce service peuvent accéder à des objets répartis et partagés de manière efficace puisque ces accès sont en majorité locaux. 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.

2 Architecture et conception

Dans ce service, l'application manipule des objets partagés et le lien avec le serveur est fait via la couche client. Nous allons décrire les principales interactions.

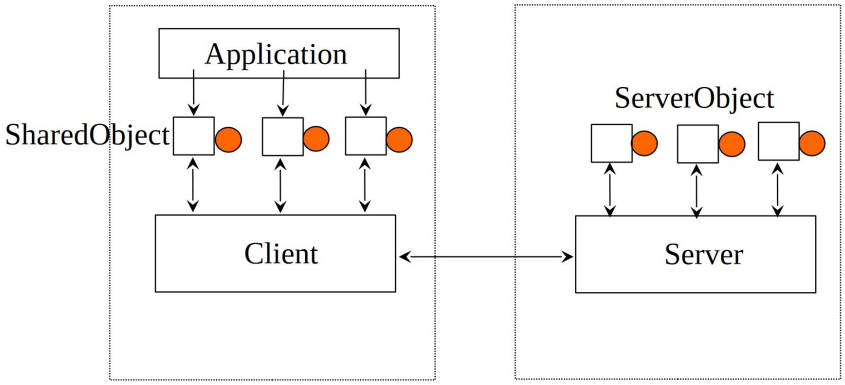

FIGURE 1 – Architecture du service de partage d'objets

2.1 Service de nommage

Ce service est le service fournit par la couche Client, il permet de créer ou retrouver des objets dans un serveur de noms. Il doit également être implanté coté serveur pour être partagé.

2.1.1 Initialisation de la couche client

Pour initialiser la couche client, il faut d'abord récupérer le serveur associé à son URL avec la méthode *Naming.lookup*. Ainsi, on obtient les informations pour communiquer avec le serveur.
Ensuite, on initialise le client et la HashMap des objets locaux *sharedObj* contenant l'identifiant unique de l'objet et l'objet partagé associé. Cela servira lorsque le seveur appellera les méthodes de nommage et déverrouillage.

L'initialisation du client est réalisée par la méthode *init*.

2.1.2 Création d'un objet partagé dans le serveur de noms

Pour créer un *SharedObject*, une *Application* demande la création d'un objet partagé au *Client*. Celui-ci transmet la demande au *Serveur* qui crée un *ServerObject* en lui attribuant un identifiant unique, stockés dans la HashMap *serverObject*. Cet identifiant est ensuite retourné au *Client* qui peut alors instancier le nouveau *SharedObject* et l'enregistrer dans les objets partagés locaux.

La création du *SharedObject* est réalisée par la méthode *create*.

2.1.3 Enregistrement de l'identifiant d'un objet partagé dans le serveur de noms

Une fois que l'objet partagé est créé, il est enregistré dans le serveur de noms : la HashMap *carnet* avec son nom et son identifiant. L'enregistrement est similaire à la création : une *Application* demande l'enregistrement d'un objet partagé associé à son nom au *Client*, qui transmet la demande d'enregistrement de l'identifiant associé au nom au *Server*. Le *Server* inscrit ensuite l'identifiant

associé au nom dans le serveur de noms.

L'enregistrement d'un objet partagé dans le serveur de noms est réalisé par la méthode *register*.

2.1.4 Récupération d'un objet partagé dans le serveur de noms

Pour récupérer un objet, l'application demande de rechercher l'objet dont le nom est donné, le *Client* transmet alors la demande de recherche de l'identifiant, associé au nom, au *Server*, qui recherche l'identifiant associé au nom dans le serveur de noms. L'identifiant est alors retourné au *Client*. Ensuite, le *Client* demande le verrou de lecture et l'objet partagé est récupéré. Ce *SharedObject* est ensuite inscrit dans la HashMap des objets locaux sharedObj. Puis le verrou en lecture est débloqué et le *SharedObject* est retourné à l'application.

La récupération d'un objet partagé est réalisée par la méthode *lookup*.

2.2 Primitives de verrouillage

2.2.1 Verrouillage

Lors d'une demande de verrouillage, l'*Application* effectue une demande de verrou en lecture ou en écriture auprès du *SharedObject*.
Selon l'état de son verrou, lorsque l'objet n'est pas en *cached* par exemple, le *SharedObject* demande au *Client* de propager la demande de verrouillage au *Server*.
Le *Server* transfère alors la demande vers le *ServerObject* associé à l'identifiant du *SharedObject*.

Le verrouillage d'un objet partagé est réalisée par les méthodes *lock_read* et *lock_write*.

2.2.2 Réduction et invalidation

Après une demande de verrouillage, le *ServerObject* effectue, selon l'état de son verrou, des demandes de réduction ou d'invalidations auprès des différents *Clients*. Ensuite, les clients propagent les demandes aux objets partagés correspondants et la liste de lecteurs *readers* et l'écrivain *writer* est mis à jour.

La réduction et l'invalidation d'un verrou d'un objet partagé est réalisée par les méthodes *reduce_lock*, *invalidate_reader* et *invalidate_writer*.

2.2.3 Déverrouillage

La méthode déverrouillage permet de libérer un verrou. Elle est utilisée dans la couche Client lors de la récupération d'un objet partagé.

Le déverrouillage d'un objet partagé est réalisée par la méthode *unlock*.

3 Implantation des opérations essentielles

Dans cette partie, nous détaillerons les méthodes mises en œuvre pour faire communiquer client et serveur.

3.1 Mise en place du verrouilage

Le changement de l'état d'un verrou d'un objet partagé s'effectue, tout d'abord, dans la classe *SharedObject* via les méthodes *lock_read* et *lock_write* qui gèrent la politique de verrouillage d'un objet partagé.

Le passage d'un état à un autre dépend des autres clients. En effet, le verrou ne peut être pris que par un unique écrivain ou par plusieurs lecteurs. De plus, seul le client peut décider de relacher son verrou. Pour implanter cela, nous avons utilisé les figures suivantes.

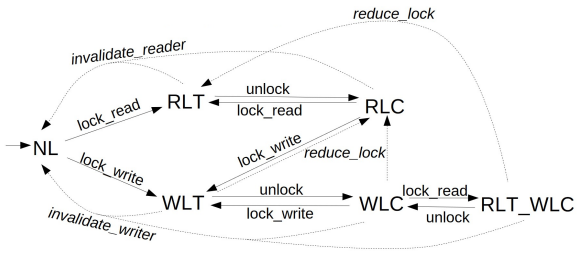

FIGURE 2 – Représentation interne des verrous sur l'objet partagé du point de vue du client

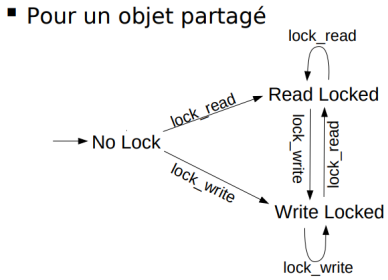

FIGURE 3 – Représentation interne des verrous sur l'objet partagé du point de vue du serveur

Selon l'état du verrou, le changement d'état peut s'effectuer en local sans informer le serveur sinon l'appel est redirigé. La demande est alors propagée au client puis au serveur puis au serverobject correspondant. Cette situation correspond aux cas suivants :

— Pour *lock_read*, lorsque le verrou est dans l'état No Lock
— Pour *lock_write*, lorsque le verrou est dans l'état No Lock ou ReadLockCashed

Dans ces cas là, le serveurObject peut demander au client des invalidations en lecture, en écriture, passage à l'etat NoLock, mais également des réductions de verrou, passage de l'état écriture à lecture. Nous avons donc implémenté les methodes *reduce_lock*, *invalidate_reader* et *invalidate_writer*. Ces demandes sont redirigées vers le sharedObject correspondant. Pour implanter ces méthodes dans la classe *SharedObject* nous utilisons différentes méthodes qui portent sur le principe de moniteur et gèrent les différents états internes d'un *SharedObject*.

3.2 La synchronisation

Plusieurs applications peuvent accéder au service de façon concurrente, il faut alors mettre en place un systeme d'attente et de synchronisation. Pour cela nous avons utilisé le principe du moniteur avec les méthodes *synchronized*, *wait* et $notify$.

Tout d'abord, la méthode *synchronized* est utilisée pour créer des verrous et garantir que les threads qui tentent d'accéder à un bloc de code spécifique ne se chevauchent pas. Nous avons utilisé cette méthode dans les classe *SharedObject* et *ServerObject* pour garantir l'exclusion d'accès au SharedObject et au ServerObject.

Les méthodes *wait* et $notify$ sont ensuite applicables à tout objet, pour lequel l'activité a obtenu l'accès exclusif. La méthode *wait* libère l'accès exclusif à l'objet et bloque l'activité appelante en attente d'un réveil via une opération $notify$. La méthode $notify$ réveille une unique activité bloquée sur l'objet, et la met en attente de l'obtention de l'accès exclusif, si aucune activité n'est bloquée, l'appel ne fait rien.

3.3 Génération des stubs

L'intérêt de l'étape 2 est de soulager le programmeur de l'utilisation des SharedObject en implantant un générateur de stubs : le *GeneratorStub*.
Cette classe permet d'obtenir un nouveau stub à partir de la classe donnée en paramètre, en faisant l'appel suivant : *java GeneratorStub Sentence* et cela crée *Sentence_stub.java* la classe qui implémente l'interface $Sentence\_itf$. Le stub est donc généré à partir d'une instrospection Java et il hérite de *SharedObject*.

Ainsi, nous allons pouvoir utiliser les stubs directement dans la classe Client pour fournir la transparence d'accès aux objets.
L'interface de Client est inchangée, mais maintenant on peut utiliser le stub à la place du *SharedObject* dans les méthodes *create* et *lookup*. Cette fois-ci on crée donc un stub à partir de l'objet et de l'identifiant récupéré et non un SharedObject. Et ensuite, c'est identique à l'étape 1, on ajoute toujours la clé id et l'objet à la HashMap des objets locaux.
Pour créer ces stubs, nous avons ajouté une méthode *create_stub* qui permet de récupérer la nouvelle classe stub à partir de la classe de l'objet et ainsi de créer le stub avec l'objet et son id que l'on utilise dans *create* et *lookup*.

Enfin, nous avons modifié la classe de tests *Irc* pour s'assurer que le programmeur manipule bien des objets tels que des *Sentence* et plus des SharedObject.

3.4 Stockage de référence dans des objets partagés

L'objectif de l'étape 3 est de prendre en compte le stockage de référence à des objets partagés dans d'autres objets partagés. Pour cela la désérialisation d'un stub doit se faire sans copié l'objet référencé. En effet l'objet référencé doit être dans un état cohérent. Lors de la déserialisation la machine doit prendre en compte le dernier état cohérent de l'objet si le stub est déjà créé ou alors en créer un. Cela permet d'éviter d'installer plusieurs stubs pour un même objet sur la même machine.

Cette étape n'a pas été implanté, nous avons eu des difficultés à visualiser l'architecture. L'idée serait d'ajouter une méthode Object readResolve() dans la classe *SharedObject* qui permettrait de récupérer l'objet référencé, créer un nouveau stub si l'objet référencé n'existe pas. Si le stub existe il faut vérifier son état de cohérence.

4 Exemples de tests et difficultés

Premièrement, nous avons effectué de nombreux test à la main à l'aide de la console fournit par la classe Irc. Nous avons pu tester les différents cas de verrou présents sur les slides de présentation du projet.

Le dernier cas de figure (cas 11) montrant un croisement de requêtes a été plus difficle à tester. En effet nous avons créé un nouvel Irc *IrcTestThread*, utilisant des thread pour effectuer des tâches parallèles dans les actions réalisées par les boutons de la console. L'objectif est de tester la synchronisation lors de l'envoi de 100 messages à la suite, en activant la lecture en même temps.

Pour tester nous lançons 2 consoles les deux en lecture et en ecriture. Voici des résultats obtenus.

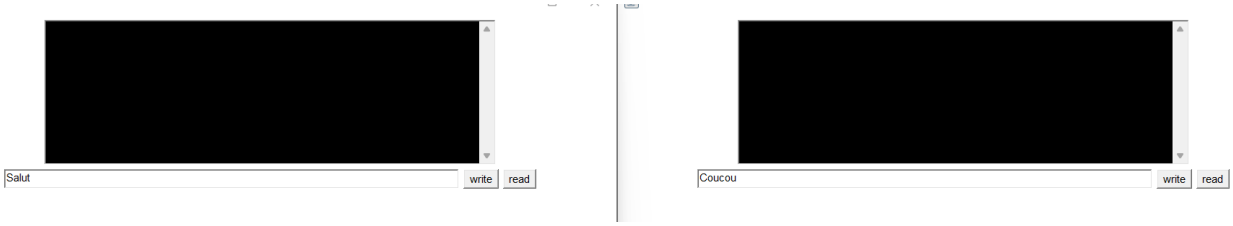

FIGURE 4 – Les deux consoles avant le début du test

Lancement du premier write, puis du deuxième, puis du premier read, puis du deuxième :

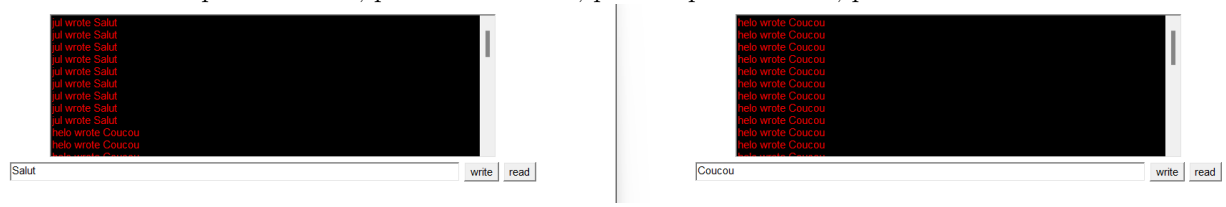

FIGURE 5 – Les deux consoles pendant le test

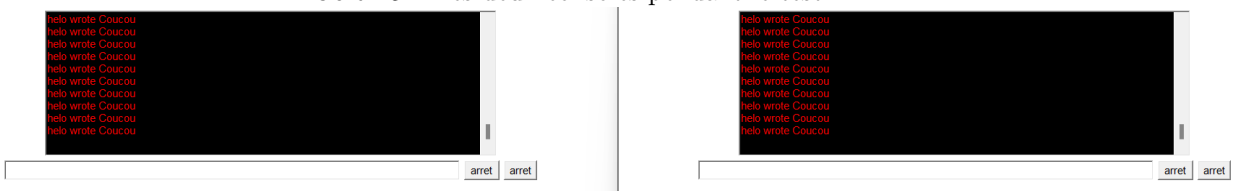

FIGURE 6 – Les deux consoles à la fin du test

La synchronisation est donc bien mise en place.

Pour l'étape 2, nous avons tout d'abord testé la création d'un stub en regénérant la classe $Sentence_stub$. Puis nous avons modifié la classe Irc pour tester les stubs à la place des $SharedObject$. Nous avons ensuite utilisé la même méthode de test qu'à l'étape 1.

5 Conclusion

Pour conclure, ce projet nous a permis de s'approprier entre autres les notions de RMI, concurrence, stubs... vues en cours au travers de la création d'un service d'écriture-lecture.
Il était intéressant de travailler en binôme sur ce projet pour confronter nos idées de conception et de programmation et se partager le travail.