ENSEEIHT

Projet System Concurent
Service de partage d'objets répartis et dupliqués en Java

Vivian GUY
Jules L'ALLEMAND

25 janvier 2016

Table des matières

Présentation du projet

Présentation

Description du projet

Le but du projet est de réaliser un service d'accès transactionnel à des objets partagés par duplication en utilisant la cohérence à l'entrée. Les accès aux fichiers se faisant en majorité en local, on pourra ainsi accéder à des objets répartis et partagés efficacement. Les objets répartis utilisés communiquent en utilisant java/RMI.

Présentation de l'application

Chaque objet est représenté par un descripteur qui contient un identifiant unique distribué par un serveur centralisé ainsi qu'un pointeur sur l'instance de l'objet partagé. L'accès à une instance d'un objet doit donc passer par le descripteur. Dans un premier temps cette indirection sera visible, on la cachera dans un second temps grâce à des stubs. //

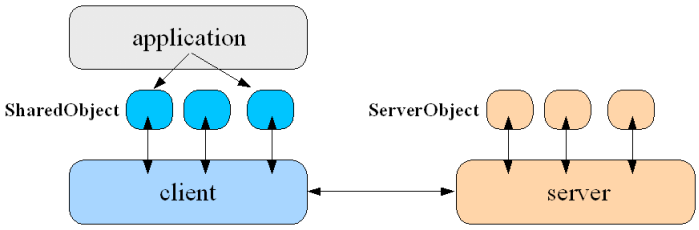

FIGURE 1 – Fonctionnement de l'application

La gestion de la concurrence entre les différents clients se fera à l'aide de locks. La classe du descripteur (SharedObject d'interface SharedObject$_itf$)$contient les mthodes qui permettent de gre rces verrous (locks) et donc de mettre en oeuvre la c$

On ne s'occupera pas de la prise de verrou multiple : une fois qu'on a pris un verrou il faut attendre de l'avoir libéré pour en reprendre un autre sur le même objet. Néanmoins, on peut avoir des verrous sur plusieurs objets différents. On ne gérera pas les interblocages dans ce cas là. //

Les SharedObject contiennent l'état du verrou sur l'objet. Cet état est représenté un par entier qui peut prendre plusieurs valeurs :

lock=0 NL(no lock) : Aucun verrou sur l'objet.

lock=1 RLC(read lock cached) : Le verrou réside sur le site et peut être pris dans l'état de lecture sans demander au serveur.

lock=2 WLC(write lock cached) : Le verrou réside sur le site et peut être pris dans l'état de d'écriture sans demander au serveur.

lock=3 RLT(read lock taken) : Le verrou est pris dans l'état de lecture.

lock=4 WLT(write lock taken) : Le verrou est pris dans l'état d'écriture.

lock=5 RLT_WLC(read lock taken and write lock cached) : Le verrou est pris dans l'état de lecture suite à un SharedObject dans l'état WLC.

Toutes les demandes de verrous de la part du client (lock_write ou lock_read) doivent passer par le serveur pour vérifier l'état du SharedObject du point de vue du serveur (et donc vis-à-vis des autres clients) et le mettre à jour le ServerObject correspondant.

Par conséquent, l'interface d'un serveur doit avoir les méthodes lock_read et lock_write pour répondre à la requête du client. Ces méthodes ont donc en paramètre la référence au client pour pouvoir lui donner la réponse.

Dans un première étape, nous allons donc mettre en place le système de verrou sur cette application et les systèmes de communication entre le serveur et les clients.

Etape 1

Mise en place des verrous et des systèmes de communication entre le serveur et les clients

1.1 Implémentation

1.1.1 Principe général

Toute application qui utilise le service doit d'abord initialiser le client. C'est ce client qui fera le lien avec le serveur. Il faut ensuite enregistrer le *SharedObject* dans le serveur de noms (à travers le client). Si l'application n'y est pas encore, on crée le *SharedObject* et on l'enregistre auprès du serveur. Ensuite, comme dit dans le sujet, tout appel à une méthode se fera d'abord en bloquant le verrou attaché au *SharedObject* et en finissant par le débloquer. Par exemple si on veux lire l'objet o représenté par le *SharedObject* s, on fera d'abord s.lock_read(), puis s.obj.read() et enfin s.unlock().

1.1.2 Le client

Tout d'abord il faut initialiser le client. Cette initialisation consiste à créer la HashMap représentant la correspondance entre les *SharedObject* et leur identifiant unique. Il reste plus qu'à créer le client et a localiser le serveur en utilisant la class *Naming* et la méthode *lookup* .

On définit naturellement une méthode *create* qui permet de créer un *SharedObject* a partir d'un object. L'ID de ce *SharedObject* est fourni par le serveur qui assure l'unicité de celui-ci.Il ne faut pas oublier de rajouter le *SharedObject* dans la HashMap des correspondances du client.

On définit aussi une méthode *lookup* qui permet de chercher un nom dans l'annuaire du serveur et de retourner le *SharedObject* correspondant.S'il n'existe pas, on le crée et on l'ajoute à la HashMap des correspondances. Cette méthode appelle la méthode *lookup* du serveur.

Il faut aussi définir une méthode qui permet de rajouter un *SharedObject* à l'annuaire du serveur. C'est la méthode *register*. Cette méthode consiste tout simplement à appeler la méthode *register* du serveur, c'est elle qui va se charger d'ajouter le *SharedObject* à l'annuaire.

1.1.3 Le serveur

Un serveur possède deux HashMaps qui permettent de garder en mémoire les correspondances entre entre les noms et les IDs des *SharedObject* et aussi entre les IDs et les *ServerObject* qui représentent les *SharedObject* de même ID mais du point de vue du serveur.

On a vu précédemment que lorsque le client appelle la méthode *lookup*, cela revenait à appeler celle du serveur. Il faut donc la définir dans le serveur. Cette méthode consiste tout simplement à regarder dans la HashMap des correspondances entre les noms et les IDs des *SharedObject* si le nom passé en paramètre (demandé donc par le client) existe ou pas. Si il existe, on renvoi l'ID correspondant, -1 sinon.

De même on a vu que la méthode *register* du client appelait celle du serveur. Cette méthode consiste simplement à ajouter le *SharedObject* dans la liste des correspondances entre les noms et les IDs (qui sont passeés en paramètre afin de renseigner le *SharedObject*).

Le serveur doit aussi s'occuper de fournir un ID unique lors de la création d'un *SharedObject*. Le serveur possède donc un compteur d'ID que l'on incrémente à chaque création d'objet afin d'assurer son unicité. Il faut ensuite créer le *ServerObject* associé à cet objet qui s'occupera de gérer le lien entre les différentes applications (plusieurs applications peuvent accéder à un même objet partagé) et l'objet.

1.1.4 Les *ServerObject*

Un *ServerObject* gère donc un objet du point de vue du serveur. Il possède donc la référence à l'ID de l'objet et directement à l'objet représenté par des *SharedObject* côté client. Il doit ainsi gérer la liste des lecteurs sur cet objet et le rédacteur. C'est principalement lui qui gère la synchronisation entre les clients sur les objets.

1.1.5 La gestion des concurrents

La gestion des clients concurrents se fait à l'aide de verrous. Lorsqu'on veut lire un objet lié à un objet partagé, on effectue d'abord un *lock_ read* sur l'objet partagé. La suite dépend de l'état précédent du verrou sur cet objet :

1. Si le verrou était en NL, alors on le passe en RLT et on fait un *lock_ read* sur le client pour l'objet en question (que l'on représente avec son unique ID), ce qui aura pour effet de passer par le serveur pour décider de la suite.
2. Si le verrou était en RLC, on le passe en RLT et on n'a pas besoin de passer par le serveur.
3. Si le verrou est dans l'état WLC, on le passe en RLT_WLC.

Dans le cas de l'écriture, les changements de verrou se font selon ce modèle :

1. Si le verrou était en NL, on le passe en WLT et de même que dans le cas de la lecture on fait un *lock_ write* sur le client.
2. Si le verrou était en RLC, on le passe WLT et on en informe le serveur en faisant un *lock_ write* sur le client.
3. Si le verrou était dans l'état WLC, on le passe en WLT dans passer par le serveur (= sans faire de *lock_ write* sur le client).

Une fois la la l'opération effectuée, on déverrouille le verrou avec la méthode *unlock* qui a pour effet de passer le verrou en *cached* (RLC si on était en RLT et WLC si on était en WLT ou RLT_WLC). Une fois le verrou déverrouillé, on notifie tous les clients que la lecture est fini pour qu'ils puissent à leur tour accéder à l'objet (en lecture ou en écriture).

L'appel à la méthode *lock_ read* ou *lock_ write* sur le client envoie la requête au serveur avec l'ID de l'objet et la référence du client afin que le serveur puisse le recontacter ultérieurement. Lorsque le serveur reçoit la requête du client, il regarde dans sa HashMap des correspondances entre IDs et *ServerObject* quel est le *ServerObject* correspondant et effectue un *lock_ read* (ou *lock_ write*) sur celui-ci. C'est donc ce *ServerObject* qui va gérer la concurrence entre les applications, c'est-à-dire entre les lecteurs et les rédacteurs.

Dans le cas d'un *lock_ read*, il faut d'abord regarder l'état du verrou que possède le *ServerObject* sur l'objet. Celui ci n'a que trois valeurs NoLock(NL),ReadLock(RL) et WriteLock(WL) puisqu'on a vu que dans les autres cas on ne passait pas par le serveur :

1. Si le verrou était en WL, on fait un *reduce_ lock* sur le rédacteur qui a pour effet de le passer en lecteur. On ajoute ensuite le rédacteur dans les lecteurs et on réinitialise le rédacteur à *null*. Enfin on passe le verrou du *ServerObject* à RL et on ajoute le client à la liste des lecteurs.
2. Si le verrou était en RL, on ajoute juste le client à la liste des lecteurs.

Le *reduce_ lock* permet de passer le verrou du *SharedObject* à RLC une fois l'écriture terminée.

Dans le cas d'un lock_write, on suit le scénario suivant :

1. Si le verrou était en RL, on invalide tous les lecteurs et on réinitialise la liste des lecteurs. On passe ensuite le verrou à l'état WL et on change le client en rédacteur.
2. Si le verrou était en WL, on invalide le rédacteur et on réinitialise le rédacteur à *null*. On passe ensuite le verrou à l'état WL et on change le client en rédacteur.

Les méthodes *invalidate_reader* et *invalidate_writer* permettent d'attendre la fin de la lecture ou de l'écriture avant de passer le verrou à NL.

1.2 Test ajouté

Une classe de test *Irc* nous était fournie mais il nous était demandé de proposer d'autres tests pour la compléter, nous avons donc créer la classe *TestEtape1*. La classe *Irc* ne faisait les tests que sur un seul objet partagé, la classe *TestEtape1* reprend le même schéma de test avec la même interface graphique mais avec plusieurs objets partagés de type *Entier* (en l'occurrence 5) et la possibilité de poser des verrous sur 2 en même temps. On peut lire la valeur d'un *Entier* en indiquant son numéro et on peut également bloquer 2 *Entier* (toujours en indiquant leurs numéros) pour écrire la somme des deux dans le premier et la différence entre les deux dans le deuxième. Ces calculs basiques sont juste là pour tester les locks sur deux objets en même temps.

1.3 Difficultés rencontrées

La principale difficulté rencontrée fut la synchronisation d'un même *SharedObject* entre les différents clients. Nous n'arrivions pas à relier les *SharedObject* entre eux à cause d'une mauvaise utilisation de la méthode *lookup*. Cette difficulté nous a pris beaucoup de temps mais une fois que nous avions notre objet partagé entre deux clients, le débogage fut beaucoup plus simple à l'aide de *print* que nous avons d'ailleurs laissés car ils aident à la compréhension des enchaînements des appels de méthodes entre *Client*, *Server* et *SharedObjet*.

Etape 2

Ajout d'un service d'accès transactionnel aux objets dupliqués

Dans cette étape on va implanter un service d'accès transactionnel aux objets dupliqués. Ce service consiste à garantir l'atomicité et l'isolation des accès aux objets. On peut choisir si on passe en gestion transactionnelle ou non.

2.1 Implémentation

Les transactions sont implantées dans la classe *Transaction*. Une transaction possède une HashMap qui correspond aux objets accédé par la transaction (la correspondance entre l'ID et le *SharedObject*). Une transaction possède aussi un booléen qui indique si elle est active ou non. Plusieurs méthodes peuvent être appelées sur une transaction :

1. *start* qui permet de rendre la transaction active
2. *commit* qui permet de valider les écritures réalisées en mode transactionnel et de finir le mode transactionnel.
3. *abort* qui annule les effets de la transaction pour les autres applications en mode transactionnel. Il faut donc remettre les objets dans l'état dans lequel ils étaient avant la transaction.

Maintenant que l'on a notre transaction, on va l'utiliser lorsque notre client effectue un *lock_ write*. Si la transaction est active, on la démarre et on ajoute aux objets accédés le *SharedObject* sur lequel on veut écrire et une fois que l'écriture est fini (*invalidate_ writer*, il faut valider les écritures réalisées par un *commit*.

2.2 Difficultés rencontrées

Nous n'avons clairement pas bien compris la façon d'utiliser les transactions. La classe *Transaction* en elle-même n'avait rien de compliqué mais nous avons l'impression que notre façon d'intégrer les transactions au code est inutile. À cause du manque de confiance sur le résultat obtenu, nous n'utiliserons pas les transactions pour les étapes suivantes et nous n'avons pas fait de tests supplémentaires.

Etape 3

Masquage de l'indirection vers les SharedObject grâce à des stubs

Dans cette étape, pour soulager le programmeur de l'utilisation explicite des *SharedObject*, on va rendre invisible pour le programmeur l'indirection que l'on réaliser jusqu'ici vers les ces derniers grâce à des *stubs*. On supposera que les *SharedObject* sont sérialisables et sont utilisables à partir de variables de type une interface. Les *stubs* générés vont hériter de *SharedObject* afin de récupérer les méthodes en rapport avec les verrous et implémenter l'interface définit précédemment (l'interface lié au *SharedObject*, par exemple Sentence_itf).

3.1 Implémentation

On va créer les *stubs* grâce à la classe *StubGenerator*. Cette classe va créer un fichier X_stub.java par classe sérialisable dans notre dossier courant (il faut que les classes représentants nos *SharedObject* soient dans ce dossier).

Il faut d'abord pouvoir récupérer toutes les classes sérialisables de notre dossier. On définit donc une méthode *getClassesSerializables* qui permet de le faire. Cette méthode parcours tous les fichiers .java de notre dossier et récupère la liste des classes qui implémentent *java.io.Serializable*. Comme *SharedObject* implémente *Serializable* et qu'on ne veut pas générer de stub pour celle-ci, on la retire de la liste.

Une fois cette liste obtenue, on va créer les fichiers stubs correspondant puis les compiler avec le compilateur java du système.

Ainsi, pour chaque classe de la liste, il faut récupérer son code. On écrit donc la première ligne de spécification de la classe : c'est forcément une classe qui hérite de *SharedObject* et qui implémente son interface correspondant. Enfin pour chaque méthode de cette classe on va récupérer les *modifiers*, les paramètres et écrire le code.

3.2 Test

Pour pouvoir tester plus efficacement notre *StubGenerator*, nous avons créer la classe de test *TestEtape3* qui utilise 2 types d'objets partagés (*Entier* défini à l'étape 1 et *Sentence*). Ainsi nous voyons si les deux stubs différents sont bien générés et si l'utilisation est bien la même pour pour le client quelque soit le type de l'objet.

Pour pouvoir utiliser ce test, il faut d'abord exécuter le *StubGenerator* dans un terminal. Ainsi, on crée les fichiers Entier_stub.java et Sentence_stub.java qui sont exécutés automatiquement. Ensuite comme d'habitude on lance le serveur dans un terminal puis deux applications TestEtape3 dans d'autres terminaux. On peut ensuite écrire soit un entier soit une phrase dans la zone de saisie et ça écrira dans l'objet correspondant la saisie. Pour afficher les deux objets, il suffit de cliquer sur *read*.

3.3 Difficultés rencontrées

Beaucoup de programmation à tâtons pour créer le code des stubs mais ça restait assez amusant. Le plus gros soucis résidait dans la gestion des types d'objet côté client. Lors d'un *lookup* où l'objet existe déjà, seul le serveur connaît le type de l'objet. Nous avons donc créer une méthode qui nous le renvoie et nous nous sommes pris la tête avec des erreurs de compilation avec des mauvais cast, des mauvaises implémentations... Nous avons

pris la décision finale de rajouter cette méthode dans *Server_itf*, cela réglait tout nos soucis mais nous pensons qu'il existe un moyen plus propre de le faire.

Etape 4

Stockage de référence à des objets partagés dans des objets partagés

Désormais, on va prendre en compte le stockage de référence à des objets partagés dans des objets partagés. Pour cela on va se servir de la méthode *readResolve*.

4.1 Implémentation

Après des recherches sur internet, nous avons trouvé un template d'utilisation de *readResolve* que nous avons essayé d'adapter à notre service. A priori, *readResolve* doit juste récupérer la valeur de l'objet et le renvoyer.

4.2 Test

Pour tester l'étape 4, on a crée une classe Auteur qui est un *SharedObject* qui possède comme attribut une sentence (qui aussi un *SharedObject*.Le *SharedObject* auteur possède donc une référence à un autre *SharedObject*. Lorsqu'on exécute le test, on peut soit modifier l'auteur avec la zone de saisie de gauche, soit modifier la sentence avec la zone de saisie de droite. Sans le *readResolve*, la modification de l'auteur dans une application est répercutée dans l'autre application mais toute modifications sur la sentence n'est pas répercutée. On voit bien ici que l'on n'a pas de stockage de référence.

4.3 Difficultés rencontrées

A l'heure du rendu, nous n'avons toujours pas réussi à faire marcher notre application. Du moins, nous ne sommes pas sûrs du résultat qu'on doit obtenir. L'utilisation de *readResolve* ne change rien sur notre test. En fait, nous ne savons pas si c'est notre *readResolve* qui est faux ou s'il s'agit de notre test qui ne met pas en évidence le problème.

Conclusion

Ce projet nous a permis d'utiliser les principes de programmation concurrente et répartie vus en cours, plus spécialement l'utilisation des verrous et des transactions. On a ainsi pu utiliser des notions utilisées en Intergiciel (serveur RMI) et en Systèmes Concurrents afin de développer ce service d'accès transactionnel à des objets partagés.

Le sujet nous a paru globalement difficile, mais nous pensons être arrivé à un résultat satisfaisant.