Projet de Systèmes Concurrents

Jorge Gutierrez & Grégoire Martini

25 Janvier 2016

Table des matières

1 Etape 1

On doit implanter le service de gestion d'objets partagés répartis. Dans cette première version, les SharedObject sont utilisés explicitement par les applications.

Plusieurs applications peuvent accéder de façon concurrente au même objet, ce qui nécessite de mettre en œuvre un schéma de synchronisation globalement cohérent pour le service que l'on implante. On suppose que chaque application voulant utiliser un objet en récupère une référence (un SharedObject) en utilisant le serveur de nom. On ne gère pas le stockage de référence à des objets partagés dans des objets partagés pour l'instant.

1.1 Coté client

La classe Client

init Initialise la connexion avec le serveur.

lookup Retourne la référence à l'objet partagé, null si il n'existe pas.

register Enregistre un nouvel objet partagé dans le serveur.

create Crée un nouvel objet partagé.

lock_read Demande le verrou en lecture d'un objet partagé.

lock_write Demande le verrou en écriture d'un objet partagé.

reduce_lock Réduit le verrou d'un objet partagé.

invalidat_reader Supprime le verrou en lecture d'un objet partagé.

invalidate_writer Supprime le verrou en écriture d'un objet partagé.

La classe SharedObject contient trois attributs :
- une énumération représentant l'état de l'objet
- l'état actuel de l'objet
- l'identifiant unique de l'objet partagé

lock_read Gère la demande de verrou en lecture sur l'objet partagé en fonction de l'état actuel.

lock_write Gère la demande de verrou en écriture sur l'objet partagé en fonction de l'état actuel.

reduce_lock Gère la demande de réduction de verrou en lecture sur l'objet partagé en fonction de l'état actuel.

invalidate_reader Gère la demande d'invalidation de verrou en lecture sur l'objet partagé en fonction de l'état actuel.

invalidate_writer Gère la demande d'invalidation de verrou en écriture sur l'objet partagé en fonction de l'état actuel.

La gestion de la synchronisation est garantie par l'exclusion mutuelle des callbacks du serveur (méthodes synchronized) et l'utilisation du moniteur java sur l'objet et ses méthodes wait et notify.

1.2 Coté serveur

La classe Server contient trois attributs :
- un entier qui est l'identifiant unique retourné pour chaque SharedObject
- une HashMap qui lie les ServerObject à leur identifiant
- une HashMap qui lie les identifiants des ServerObject à leur nom

lookup Permet de récupérer l'identifiant d'un objet partagé en fonction de son nom.

register Enregistre le lien entre l'identifiant d'un objet partagé et son nom.

create Crée un nouvel objet partagé et retourne son identifiant.

lock_read Passe la demande de lock_read au ServerObject correspondant.

lock_write Passe la demande de lock_write au ServerObject correspondant.

La méthode principale initiale la connexion distante et enregistre le serveur dans le serveur de nom.

La classe ServerObject contient cinq attributs :
- une énumération représentant l'état de l'objet
- l'état actuel de l'objet
- la liste des clients en lecture
- le client en écriture
- l'identifiant unique de l'objet partagé

lock_read Permet d'ajouter un client en lecture tout en s'assurant qu'il n'ai pas de client en écriture. Appel reduce_lock sur le client en écriture pour s'en assurer.

lock_write Permet de passer un client en lecture tout en s'assurant qu'il n'y ai pas de client en lecture ou en écriture.
Appel invalidate_writer sur le client en écriture et crée un thread qui appel invalidate_reader sur chaque client dans la liste des lecteurs pour une gestion efficace de la cohérence.

Les deux méthodes sont synchronized ce qui garanti leur exécution en exclusion mutuelle et évite le croisement d'appels comme illustré dans le sujet.

L'application a été testée en modifiant Irc afin de séparer les lock et les unlocks (ajout d'un bouton) et la classe Test qui permet de stresser le serveur quand on en lance plusieurs instances.
Si l'application client imbrique les locks sur un même objet, le SharedObject lève l'exception NestedLocksException et les interblocages ne sont pas gérés.

2 Etape 2

Dans l'étape deux, on a implanté le service de transaction aux objets dupliqués de la façon suivante :

Dans la Transaction on stocke trois listes (HashMap), une liste pour les SharedObjects lus, une liste pour les SharedObjects écrits et une liste des Objects qui ont été modifiés au cas où l'application fasse abort. Il y a aussi un boolean pour indiquer si la transaction est active et un boolean pour indiquer si l'application a appelé commit ou abort. Finalement, la Transaction a un attribut static transaction qu'on va envoyer chaque fois que SharedObjet appelle getCourrentTransaction.

Par conséquent on est obligé d'ajouter les fonctions addObjectRead(), addObjectWrite(), addMemoire() qui ajoutent les SharedObjects et les Objects avec leur ID dans leur liste respective. Le hasbeenModified() a été ajouté afin de savoir si c'est la première fois que l'objet est modifié et le garder dans la mémoire. Le commitDone() a pour but de réinitialiser la variable commitvalide une fois que l'abort est exécuté.

En sachant que l'application gérera la Transaction seulement avec les méthodes start(), commit() et abort(), on a modifié la classe SharedObject en ajoutant la méthode static getCurrentTransaction() pour récupérer la même transaction dans autre interface, ainsi le traitement de lockwrite(), lockread et unlock() pourront dans cette classe vérifier si la transaction est active.

À l'intérieur du lock() on vérifie si l'application est en mode transactionnel avec isActive() et on appelle addObjectWrite() ou addObjectRead() selon la méthode. Si l'application modifie l'objet on garde sa valeur en faisant un clonage Byte à Byte de celui-ci grâce à la méthode deepClone qui utilise la serialisation.

Puis pour la fonction unlock() on regarde si la transaction n'est pas active pour libérer le verrou ou si l'objet n'a pas été modifié, sinon on ne le libère pas. Dans le cas où l'application appelle abort on retourne l'objet à l'état avant modification.

Pour tester la fonctionnalité de transaction on a mis des boutons dans Irc pour appeler start, commit et abort, en assurant qu'on ne verrouille pas l'objet tant que la transaction est active.
On a aussi crée une classe Tests qui permet de stresser la transaction. Elle crée le nombre de ClientsTest passé en paramètre et ces derniers exécutent en mode transactionnel le nombre d'actions sur le nombre d'objets définis dans Tests.

3 Etape 3

On doit implanter un générateur de stub, destiné à soulager le programmeur de l'utilisation explicite des SharedObject.

Le générateur de stub est la classe StubGenerator.
La classe métier et l'interface de l'objet à partagé doivent être fournis par le programmeur.

Le stub étends la classe SharedObject et implémente l'interface de l'objet à partager ainsi que la classe Serializable. Il contient un constructeur qui est celui de SharedObject et l'ensemble des méthodes publiques de la classe métier. Il gère aussi les annotations @Read and @Write qui permettent de soulager le programmeur de la gestion explicite des verrous sur l'objet partagé dans son application. Il lui suffit d'annoter les méthodes de la classe métier selon qu'elles modifient (@Write) ou utilisent simplement l'état (@Read) de l'objet.

On utilise la classe File qui permet de créer un nouveau fichier et la classe FileWriter qui permet d'écrire dans un fichier. Le code générer est stocké dans un StringBuffer avant d'être écrit dans le fichier.
Toute l'architecture du stub est écrite dans des chaines de caractères statique et seules les parties dépendantes de la classe dont on génère le stub telles que le nom des méthodes ou des paramètres est récupérés au moment opportun.
On utilise l'introspection de Java pour accéder à la classe métier. Attention, l'utilisation de la méthode getParameter pour récupérer les paramètres d'une méthode nécessite l'utilisation de Java 8.

Lorsque le client à besoin du stub d'un objet, il utilise la méthode getStub en passant la classe de l'objet dont il veut le stub en paramètre et la méthode lui renvoi la classe du stub. Le code du stub est ainsi généré et compilé à la volée et instancié par le client de façon transparente pour le programmeur. Le stub n'est généré que si une classe nomdeclasse_stub n'existe pas déjà pour éviter de créer un source et le compiler à chaque fois que l'on demande un stub. Si l'on modifie l'objet, il faut donc supprimer son stub pour qu'un nouveau soit généré au prochain appel.

Une méthode a été ajoutée à la classe Server pour récupérer la classe de l'objet partagé afin de créer le bon stub lors de l'appel de la méthode lookup dans la classe Client.

Si la compilation échoue, le StubGenerator lève l'exception CompilationFailedException.

4 Etape 4

On désire prendre en compte le stockage de référence à des objets partagés dans des objets partagés.

Il s'agit de spécialiser la méthode readResolve dans la classe SharedObject pour que lorsqu'un SharedObject est désérialisé dans le client (ce qui arrive uniquement lorsqu'il est lui même contenu dans un objet partagé) il soit remplacé par le stub local correspondant.

La méthode lookup a été surchargée dans la classe Client afin de rendre le stub de l'objet partagé non pas en fonction de son nom mais de son identifiant.

L'étape a été testée en modifiant Irc pour utilisée un Sentence_cx qui contient un Sentence et vérifier l'installation du stub en local lorsque l'on récupère l'objet partagé.