Projet Données Réparties

MEGEMONT Clément, SEGHROUCHNI Anas

Département Sciences du Numérique - Deuxième année
2022-2023

Table des matières

1 Introduction **3**
1.1 Présentation des fonctionnalités 3

2 Etape 1 **3**

3 Synchronisation **4**
3.1 Test de synchronisation 5
3.2 Implémentation 5
3.2.1 Premier problème 5
3.2.2 Deuxième problème 6

4 Etape 2 **7**
4.1 Générateur de stub 7
4.2 Annotation 7

5 Conclusion **7**

1 Introduction

Le but de ce projet est d'illustrer les principes de programmation répartie vus en cours. Pour ce faire, nous allons réaliser sur Java un service de partage d'objets par duplication, reposant sur la cohérence à l'entrée (entry consistency).

En utilisant RMI, nous avons implanter un service de partage d'objets en essayant d'assurer la cohérence des accès concurrents aux ressources. Le plus de ce système est que le client garde dans son cache une version locale de l'objet et, selon les cas d'utilisation, il devra appeler le serveur ou non afin de pouvoir lire ou écrire sur l'objet.

1.1 Présentation des fonctionnalités

Chaque objet partagé est représenté par un descripteur (*SharedObject*) qui possède en attribut l'objet en question. Le but du projet est que chaque utilisateur (Client) possède une version de ce SharedObject, soit directement comme dans l'étape 1, soit indirectement comme dans l'étape 2.

Le service est architecturé comme suit :

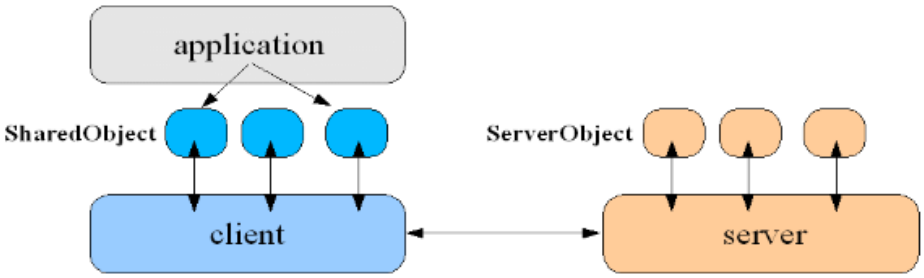


FIGURE 1 – Architecture du service

La lecture et l'écriture sur l'objet sont toujours précédées par les méthodes *lock_ read* et *lock_ write* de *SharedObject*. Ces méthodes nous permettent de gérer la cohérence des accès concurrents sur l'objet. Notons aussi que nous supposons qu'un *lock_ read* et *lock_ write* est toujours suivi par un *unlock*.

L'objectif du projet est la mise en partage d'objets entre différents utilisateurs à travers un serveur. Nous avons donc une classe *Client* qui peut soit :

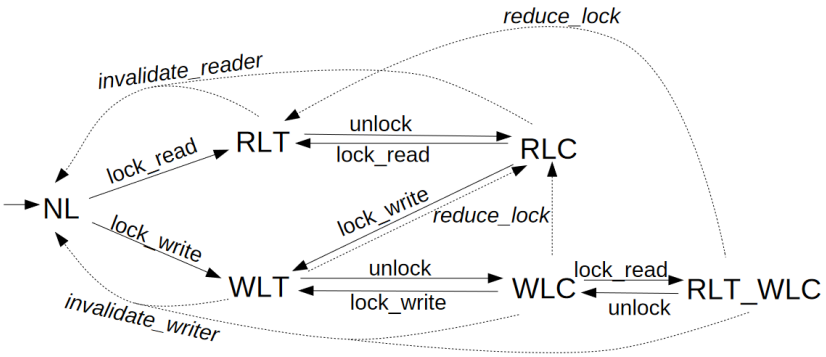
- créer un objet partagé : à partir de la méthode *create(Object o)*. On retourne le descripteur de l'objet partagé. A la création, l'objet n'est pas verrouillé.
- consulter le serveur pour savoir si l'objet partagé existe déjà : à partir de la méthode *lookup(String n)*. La méthode retourne l'objet partagé enregistré.
- enregistrer un objet partagé dans le serveur de noms à partir de *register((String name, SharedObject_ itf so)*

2 Etape 1

Nous avons commencé par créer 4 classes :

- *Serveur.java* qui permet de créer un serveur qui va gérer les accès concurrents aux objets partagés.
- *ServerObject.java* qui implante les objets gérés par le serveur
- *Client.java* qui définit les fonctions utilisateurs à appliquer aux objets partagés.
- *SharedObject.java* qui implante les objets partagés du point de vue client.

Nous avons ensuite défini un nouveau type *Lock* dans la classe *SharedObject*, qui est une énumération de NL, WLT, WLC, RLT, RLC, RLT_WLC qui sert à caractériser l'état d'un *SharedObject* (écriture, lecture, verrou actif ou pas...), puis nous avons écrit 3 fonctions lock_read, lock_write et unlock qui servent respectivement à prendre le verrou en lecture, en écriture ou à rendre le verrou. Ces fonctions changent l'état du verrou du *SharedObject* en suivant le schéma ci-dessous. De plus, lock_read et lock_write renvoie soit l'objet gardé en cache, soit appellent le serveur pour récupérer l'objet du *ServerObject* associé, suivant l'état dans lequel est le *SharedObject*.



Après cela, nous avons défini les fonctions create, lookup et register dans la classe *Client*, qui appellent le serveur pour permettre respectivement de créer un objet partagé, d'accéder à un objet du serveur par son identifiant, et d'enregistrer un objet dans le serveur.
Nous avons choisi, lors de l'appel à create d'initailiser l'objet à *null* car le client est obligé de faire appel soir à *lock_ read* soit à *lock_ write* pour manipuler l'objet ce qui mettra à jour l'objet. Nous aurions pu faire appel à *lock_ read* de client pour récupérer l'objet mais cela entrainait l'ajout d'un "faux" lecteur aux yeux du serveur. Ensuite, pour gérer les accès concurrents aux objets, nous avons implémenté 3 fonctions supplémentaires : invalidate_writer, invalidate_reader et reduce_lock. Ces fonctions permettent à un client, lorsque l'accès à un objet n'est possible instantanément, d'attendre que l'objet soit libéré pour pouvoir prendre le verrou sur celui-ci. Ces fonctions mettent donc à jour l'état des *SharedObject* qui n'ont plus l'accès à l'objet. Ils devront donc par la suite repasser par le serveur pour récupérer un nouvel accès sur l'objet.
Nous avons pu tester nos fonctionnalités à l'aide de l'application fourni *Irc.java* qui crée 2 clients ayant un accès sur le même objet. Pour cela, nous avons simulé chaque cas possible pour voir si le résultat était celui attendu. Après une phase de debuggage, nous avons réussi à obtenir un service répondant à ce que l'on souhaitait.

3 Synchronisation

La synchronisation représente une majeure partie du projet. En effet, la possibilité d'avoir un grand nombre de lecteurs et de rédacteurs sur un même objet partagé peut conduire à des problèmes de cohérence et donc entraîner de plus gros problèmes. Nous avons donc décidé dans un premier temps de créer un test de synchronisation afin de pouvoir valider par la suite notre code.

3.1 Test de synchronisation

Pour tester notre synchronisation nous avons décidé de créer une classe *IntShared* qui représentera notre entier partagé. La classe *IntShared* possède un unique attribut entier *data* initialisé à 0. La classe *IntShared* possède les méthodes :

— *read* : qui retourne l'entier
— *write(n)* : qui ajoute n à la *data*

Ensuite nous avons une class *TestNRV* qui, lorsque l'on appelle *TestNRV(t)* va créer un tableau de *SharedObect* de taille t et va les enregistrer au niveau du serveur.

Ainsi, on a une classe *Creator* qui lance un tableau de taille 10 de *SharedObject*.

Enfin, la classe *Accessor* : elle va récupérer le tableau de *SharedObject*. Ensuite elle va prendre 2 nombres au hasard, le premier entre 1 et 9 (r) et l'autre entre 0 et 1000 (n), puis va modifier la case r du tableau de SharedObject en lui ajoutant n (va appeler tableau[r].write(n) avec les dispositions préalables). De plus, *Accessor* va aussi ajouter n au premier élément du tableau (va appeler tableau[0].write(n) avec les dispositions préalables). Ensuite elle va faire des *lock_read* de tout les éléments de notre tableau afin d'essayer de capter l'ensemble des cas d'interblocage possibles.

Nous lançons 100 *Accessor* simultanément via ce script bash :

```
nb=100
while [ $nb -ne 0 ]; do
    java Accessor &
    nb=`expr $nb - 1`
done
sleep 60
java Accessor affiche
```

L'objectif du test est, qu'à la fin, la première valeur soit égale à la somme des autres. Le sleep 60 permet d'attendre que finissent tous les processus.

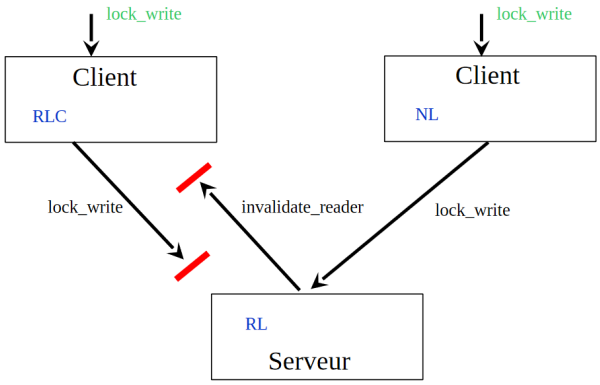
NB : Il y a sûrement une erreur dans notre code ou test car nous avons une variation légère entre la première case du tableau et la somme des autres. Sûrement un écriture qui n'a pas dû se faire au début.

3.2 Implémentation

Dans un premier temps, la synchronisation implique de gérer les demandes d'invalidation (*invalidate_reader*, *invalidate_write*, *reduce_lock*) qui sont envoyées à un objet SharedObject qui est en taken (soit entrain de lire, soit entrain d'écrire). Pour cela, nous avons utilisé la primitive *wait* pour mettre un thread en attente. Lorsque la méthode unlock est invoquée par l'utilisateur, un appel à *notify* est utilisé pour réveiller le thread endormi.

3.2.1 Premier problème

La mot clé "synchronized" permet à une partie du code d'être en exclusion mutuelle par rapport à une instance de sa classe. Il peut être tentant de déclarer toutes les méthodes d'un objet Server, ServerObject, Client ou SharedObject comme synchronisées, mais cela entraîne des cas de d'interblocage. En particulier ce dernier :



Ainsi, au vu de ce problème, il était important de ne pas déclarer la méthode *lock_ write* synchronized. Afin de le régler ce problème de synchronisation nous avons modifié plusieurs méthodes.

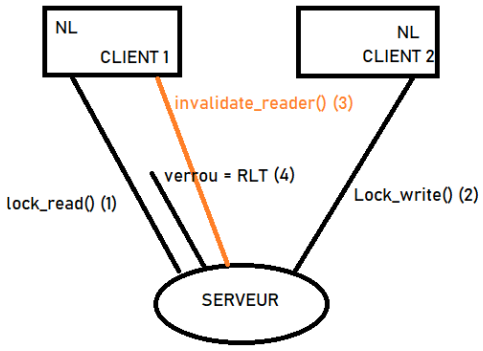
Premièrement nous avons décidé que lorsque l'on se fait invalider, l'objet pointé par le *SharedObject* passe à *null*. C'est un moyen pour le *SharedObject* de savoir si il s'est fait invalider. Ensuite, pour la méthode *lock_ write* nous avons donc décidé de sortir les appels au serveur du bloc synchronisé. A la suite du bloc synchronisé, nous appelons la méthode *lock_ write* du serveur qui nous renvoie la version mise à jour de l'objet. Puis nous testons **dans un bloc synchronisé** si l'objet est *null* (donc nous nous sommes fais invalidé entre temps) et donc notre verrou ne passe pas à WLT. Nous bouclons ainsi jusqu'à que nous réussissons à avoir le verrou en écriture.

Voilà, en pseudo code, la portion de code qui n'est pas synchronisé dans le *lock_ write* :

while verrou != WLT **do**
  objet = appel au serveur
  if objet != null **then**
    synchronisé(modifier l'objet du SharedObject et le verrou)
  end if
end while

3.2.2 Deuxième problème

De plus nous avons rencontré un autre problème de synchronisation. Le cas d'un lecteur/redacteur qui se fait invalider alors qu'il est toujours en NL :

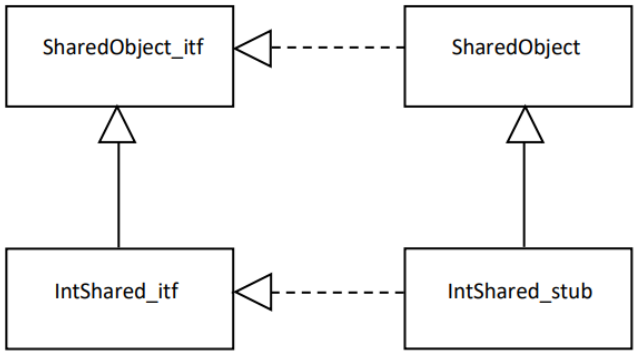


Sur le schéma, le client 1 demande un *lock_read* au serveur mais un autre client demande une écriture. Le serveur doit donc invalider l'ensemble des lecteurs. En particulier le serveur doit invalider le client 1 qui est toujours en NL. Il faut donc modifier la classe *Invalidate_reader* et *invalidate_writer* afin, dans le cas NL, d'attendre (via la primitive wait()) que son verrou passe en RLC ou WLC et que la lecture/ecriture soit faîte.

4 Etape 2

4.1 Générateur de stub

Pour éviter que l'utilisateur manipule directement les *SharedObject*, nous créons une classe *Generateur2stub* qui à partir d'une interface héritant de *SharedObject_itf*, crée un stub qui implémente cette interface. Ainsi on ne manipule plus les *SharedObject* qui ont une référence sur un objet mais sur un stub qui hérite de *SharedObject*. Nous avons donc dû modifier les fonctions create et lookup dans la classe *Client.java* pour ne plus créer de *SharedObject* mais un stub. Une autre différence est, qu'ici, *create* a besoin de l'objet afin de récuperer sa classe "_stub" et donc l'objet ne peut plus être initialisé à *null*. Donc nous avons rajouter une méthode *get_obj* dans *Serveur* et *Server_Object* qui renvoie juste l'objet sans modifier les lecteurs.



4.2 Annotation

Pour que les appels à lock_read et lcok_write ne se fassent plus directement par l'utilisateur, nous avons créé deux annotations @read et @write que nous mettons dans les interfaces héritant de *SharedObject_itf*, de telle sorte que chaque appel à une fonction annotée fait d'abord un appel à lock_read ou lock_write, éxécute le code de la fonction, puis appelle unlock. Ainsi, les accès concurrents aux objets sont gérés automatiquement, le client peut simplement appeler les méthodes sur le stub.

5 Conclusion

La résolution des problèmes de ce projet nous a pris beaucoup de temps vu que chaque classe appel une autre classe. De plus la synchronisation a été particulièrement difficile et nous ne savons pas vraiment si notre code est robuste. L'étape 3 n'a pas était traité nous avons préféré essayer de régler les problème de synchronisation.