RAPPORT
Projet Données Réparties

Gabriel ROCHAIX–YAMAMOTO
Jonas LAVAUR

Département Sciences du Numérique - Deuxième année
2022-2023

Sommaire

Table des figures

1 Introduction

Le but de ce projet était de développer en Java un service de partage d'objets dupliqués, afin de mettre en pratique les notions de programmation répartie vues dans l'UE Systèmes Concurrents et Communicants.

De plus, le service s'exécute par des objets Java répartis qui communiquent entre eux à l'aide de Java/RMI. La gestion de la cohérence lors des accès aux objets se fait à l'aide de verrous.

2 Explications Etape 1

Cette 1ère étape avait pour but d'implanter le service à objets dupliqués avec une utilisation explicite des SharedObject. Le plus compliqué à cette étape est d'avoir une synchronisation fonctionnelle.

2.1 Implantation du service et Synchronisation

Nous avons choisi d'implémenter des SharedObject et des ServerObject. Au niveau du client l'objet partagé est géré par le SharedObject alors qu'au niveau du serveur, l'objet est géré par le ServerObject. Pour reconnaître quel SharedObject/ServerObject est associé à un objet, nous attribuons un entier au nom de l'objet qui sert d'identité. Cet entier est ainsi gardé en mémoire par le SharedObject/ServerObject.

Pour représenter les états de l'objet nous avons implémenté 2 énumérations. EtatLockClient au niveau du Client (NL, RLC, WLC, RLT, WLT, RLT_WLC) et EtatLockServer au niveau du Serveur (NL, RL, WL).

Pour la synchronisation au niveau du SharedObject nous avons utilisé des wait() et notify(). Dans les fonctions représentant des requêtes venues du serveur (reduce_lock() ;invalidate_reader() ;invalidate_writer()), nous utilisons des wait() si l'état de l'objet est en "taken". Et dans la fonction unlock() nous avons utilisé un notify() en fin de fonction.

Pour la synchronisation au niveau du ServerObject nous avons choisi de lock() unlock(), en début et en fin de toutes les fonctions (sauf le constructeur), afin de permettre à un seul Client de lock_write() ou lock_read(). Cela permet de résoudre le cas de figure 11 des slides de présentation du projet.

2.2 Tests réalisés

Nous avons d'abord testé les cas de figure 1 à 11 des slides de présentation du projet, avec l'exemple fourni Irc, à la main avec seulement 2 clients. D'ailleurs, nous avons dû ajouter des sleep dans la classe Irc.java (que nous avons par la suite supprimés) entre les lock et les unlock afin de maintenir certains états comme RLT ou WLT.

De plus, afin de visualiser les différents états, nous avons ajouté des print dans les classes : Client.java ; Server.java ; SharedObject.java et ServerObject.java. Nous avons mis ces print dans des "if (affiche)" où affiche est un booléen à mettre à Vrai si l'on veut avoir les print (le booléen affiche est un attribut présent dans les 4 classes citées au-dessus).

Afin de tester avec plus de clients nous avons implémenté une nouvelle classe MyInteger.java qui représente un entier qu'on peut lire et incrémenter de 1. Puis nous avons implémenté une nouvelle classe Synchro.java qui écrit ou lit aléatoirement dans un MyInteger partagé et qui compte localement le nombre de fois qu'il a écrit. Ensuite à l'aide d'un script bash (qui s'utilise ainsi dans le terminal : ./TestSynchro.sh) nous lançons une dizaine de clients (Synchro.java) et nous sommons tous les compteurs locaux des clients pour savoir en réalité combien de fois nous avons incrémenté l'entier. Nous pouvons ensuite comparer ce résultat à l'entier lu sur l'objet partagé et s'il s'agit du même nombre alors la synchronisation a marché. En effet, le script bash génère le fichier "out.txt", dans lequel nous avons tous les résultats obtenus, dont voici un exemple :

```
1
2 5
3 4
4 6
5 4
6 4
7 5
8 4
9 4
10 6
11 7
12 3
13
14 Compteur théorique (somme des valeurs précédentes) :
15 52
16
17 Compteur réel :
18 52
```

FIGURE 1 – Exemple de fichier "out.txt"

Nous avons cependant rencontré un problème : 2

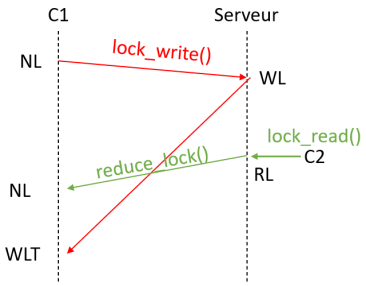

FIGURE 2 – Un problème que nous avons rencontré

Dans ce cas, le client pense qu'il a les droits d'écriture mais le serveur pense que ce client n'a que les droits de lecture. Nous avons palié à ce problème en rajoutant des sleep dans les fonctions lock_write() et lock_read() de ServerObject mais la solution n'est pas idéale et ne marche plus à partir d'un certain seuil de clients.

3 Explications Etape 2

Dans cette 2ème étape, nous avons développé un générateur de stubs pour rendre l'accès aux objets transparent, afin de soulager le programmeur de l'utilisation des SharedObject. De plus, nous avons décidé de ne pas utiliser de système d'annotations.

Voici les différents ajouts/modifications que nous avons effectué :

- On a rajouté le constructeur Sentence_stub à la classe Sentence_stub.java fournie. Ce constructeur utilise celui de SharedObject (classe dont Sentence_stub hérite).

- On a créé le générateur de stubs (classe Generator_stub.java), il s'utilise ainsi dans le terminal :

java Generator_stub <nom_objet_dont_on_veut_creer_le_stub>

Par exemple, "java Generator_stub Sentence" génère la classe Sentence_stub.java. De même avec l'objet MyInteger.
Pour créer ce générateur, on s'est servis du mécanisme d'introspection de Java vu en métaprogrammation (utilisation des méthodes : getDeclaredMethods(), getReturnType(), getParameters(), ...).

De plus, pour générer un fichier et écrire dedans, nous nous sommes servis d'un FileWriter sur un File.

- Dans la classe Client.java, nous avons ajouté la méthode create_stub qui retourne une instance du stub généré. Et donc, dans les méthodes lookup et create de Client, on a remplacé la ligne "SharedObject so = new SharedObject(client, id, o) ;" par la ligne "SharedObject so = create_stub(client, id, o) ;".

- On a également modifié le type du 2ème paramètre de la méthode register dans Client.java, on a remplacé SharedObject par SharedObject_itf.

- Enfin, on a modifié les classes Synchro.java et SynchroRead.java de la même façon que Irc.java a été modifiée (3).

```
Sentence_Itf s = (Sentence_Itf)Client.lookup ("MySentence");
s.lock_read() ;
s.meth();
s.unlock() ;
```

FIGURE 3 – Modifications à apporter à Irc lors de l'étape 2

Pour tester, nous avons réalisé les mêmes tests que lors de l'Etape 1 mais avec les changements effectués lors de l'Etape 2.

4 Explications Etape 3

Le but de cette 3ème étape était de prendre en compte le stockage de référence à des objets partagés dans des objets partagés. Lors de la sérialisation d'un stub, ne pas copier l'objet référencé et lors de la désérialisation, installer le stub de façon cohérente sur la machine (sans installer de doublons).

Pour ce faire, nous avons spécialisé la méthode de sérialisation Object readResolve() dans les classes SharedObject.java et ServerObject.java :

- On regarde si le client (resp. serveur) possède le SharedObject (resp. ServerObject).
- Si oui, on retourne cet objet.
- Sinon, on retourne "this" : l'objet associé à la classe SharedObject (resp. ServerObject).

De plus, nous avons ajouté le mot-clé "transient" à l'attribut de type Object, dans les classes SharedObject.java et ServerObject.java. Ce mot-clé est lié au concept de sérialisation et permet de ne pas enregistrer la valeur d'un attribut lors de la sauvegarde de l'état d'un objet.

Enfin, nous n'avons pas su comment réaliser des tests pour cette dernière étape. En effet, nous ne voyons pas trop comment vérifier à la fin si l'on a installé ou non plusieurs stubs pour un même objet sur la même machine.

5 Conclusion

Pour conclure, les plus grandes difficultés que nous avons rencontré résident dans la compréhension de ce qui était attendu lors des différentes étapes et dans la réalisation d'une synchronisation fonctionnelle (Etape 1).

Enfin, au niveau des améliorations éventuelles, pour résoudre le problème de synchronisation nous avons essayé de mettre en attente les requêtes du serveur lorsqu'un lock_write ou un lock_read est en cours d'exécution mais nos solutions pouvaient amener à des situations d'interblocage. Nous pensons qu'il existe une solution que nous n'avons pas trouvé.