Systèmes Concurrents & Intergiciel

Rapport du projet de données réparties

MOUTAHIR Jed
ZHENG Steven

Département Sciences du Numérique - Deuxième année
2022-2023

Table des matières

Table des figures

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).

Les applications Java utilisant ce service peuvent accéder à des objets répartis et partagés de manière efficace puisque ces accès sont en majorité locaux (ils s'effectuent sur les copies (réplicas) locales des objets). 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.

Dans ce service, les objets sont représentés par des descripteurs (instances de la classe *SharedObject* d'interface $SharedObject\_itf$) qui possèdent un champs *obj* qui pointe sur l'instance (ou une grappe d'objets) Java partagée. Toute référence à une instance partagée doit passer par une telle indirection.
Dans une première étape, cette indirection est visible pour le programmeur qui doit adapter son mode de programmation. Dans une seconde étape, on implantera des stubs qui masquent cette indirection.

Le service est architecturé comme suit.

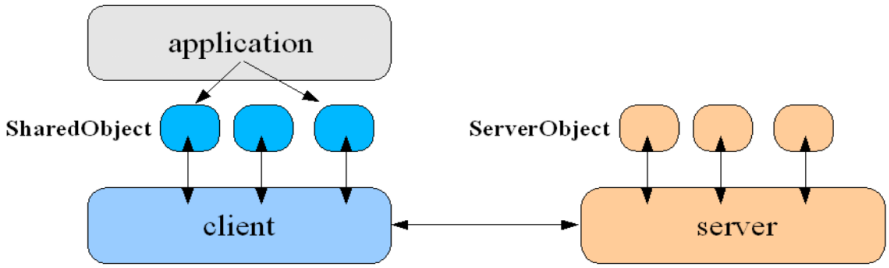

FIGURE 1 – Architecture du projet

2 Classes du projet

2.1 Classes du Projet

Les classes qui constituent le projet sont les suivantes :

— $Client\_itf$: L'interface du Client.
— *Client* : L'implementation de $Client\_itf$.
— $Server\_itf$: L'interface du Serveur.

— *Server* : L'implementation de $Server\_itf$.

— *SharedObject* $\_itf$: L'interface de l'objet partagé.

— *SharedObject* : L'implementation de $SharedObject\_itf$.

— *ServerObject* : La classe pour un objet côté Serveur.

— *StubGenerator* : La classe qui permet de générer un stub à partir d'une l'interface.

2.2 Classes de Test simple

Les classes qui contituent le test de fonctionnement simple sont les suivantes :

— *Irc* : La classe qui lance un chat sur un objet partagé IRC de type *Sentence*.

— $Sentence\_itf$: L'interface pour une phrase.

— $Sentence\_stub$: Un stub généré avec *StubGenerator* à partir de $Sentence\_itf$.

— *Sentence* : L'implémentation de $Sentence\_itf$.

2.3 Classes de surcharge simple

Le test de surcharge choisi est le suivant :

— On lance N processus sur un objet (Un entier) en parallèle.

— Chaque processus lit la valeur de l'objet et ajoute 1.

— On attends qu'ils finissent tous et on lit la valeur finale de l'objet.

— On attends de lire N.

En effet, si le système fonctionne parfaitement, on attent de trouver exactement N. Tout nombre inférieur indique des erreur lors de l'attribution des jetons.

Les classes qui contituent le test de surcharge sont les suivantes :

— $TextBruteForce$: La classe qui lance des processus en parallèle sur un même objet.

— $clientPlus1$: La classes d'un processus qui ajoute 1 à l'objet partagé.

— $IntContainer\_itf$: L'interface de l'objet partagé.

— $IntContainer\_stub$: Le stub généré à partir de $IntContainer\_itf$ avec *StubGenerator*.

— $IntContainer$: L'implémentation de $IntContainer\_itf$.

2.4 Classes de test du bon fonctionnement de la logique à implémenter

Le test de vérification du bon fonctionnement de la logique à implémenter choisi est le suivant :

— On lance N processus qui possèdent chacun une JVM sur un objet (Un entier) en parallèle.

— Chaque processus lit la valeur de l'objet et ajoute 1.

— On attends qu'ils finissent tous et on lit la valeur finale de l'objet.

— On attends de lire N.

En effet, si le système fonctionne parfaitement, on attent de trouver exactement N. Tout nombre inférieur indique des erreur lors de l'attribution des jetons.

Les classes qui contituent le test de surcharge sont les suivantes :

— $TestPlus1$: La classe qui ajoute 1 à l'objet partagé.

— $Validation$: La classe qui valide les nombre final attendu.

— $IntContainer\_itf$: L'interface de l'objet partagé.

— $IntContainer\_stub$: Le stub généré à partir de $IntContainer\_itf$ avec $StubGenerator$.

— $IntContainer$: L'implémentation de $IntContainer\_itf$.

— $test.c$: Lance N $TestPlus1$ avec chacun une JVM

2.5 Classes de test de l'étape 3

Le test de vérification du bon fonctionnement de l'étape 3 choisi est le suivant :

— On lance N processus qui possèdent chacun une JVM sur un objet (Un Compteur qui référence un Historique) en parallèle.

— Chaque processus lit la valeur de l'objet et ajoute 1.

— On attends qu'ils finissent tous et on lit la valeur finale de l'objet.

— On attends de lire N.

En effet, si le système fonctionne parfaitement, on attent de trouver exactement N. Tout nombre inférieur indique des erreur lors de la résolution des références aux objets.

Les classes qui contituent le test de surcharge sont les suivantes :

— $CompteurTest$: La classe qui lance le test.

— $CompteurValidation$: La classe qui valide les nombre final attendu.

— $Compteur\_itf$: L'interface de l'objet compteur partagé.

— $Compteur\_stub$: Le stub généré à partir de $Compteur\_itf$ avec $StubGenerator$.

— $Compteur$: L'implémentation de $Compteur\_itf$.

— $Historique\_itf$: L'interface de l'objet historique partagé.

— $Historique\_stub$: Le stub généré à partir de $Historique\_itf$ avec $StubGenerator$.

— $Historique$: L'implémentation de $Historique\_itf$.

— $testCompteur.c$: Lance N $CompteurTest$ avec chacun une JVM

3 Algorithmes des opérations essentielles

3.1 *ServerObject*

voici le graphe qui décrit le fonctionnement voulu pour un object côté serveur :

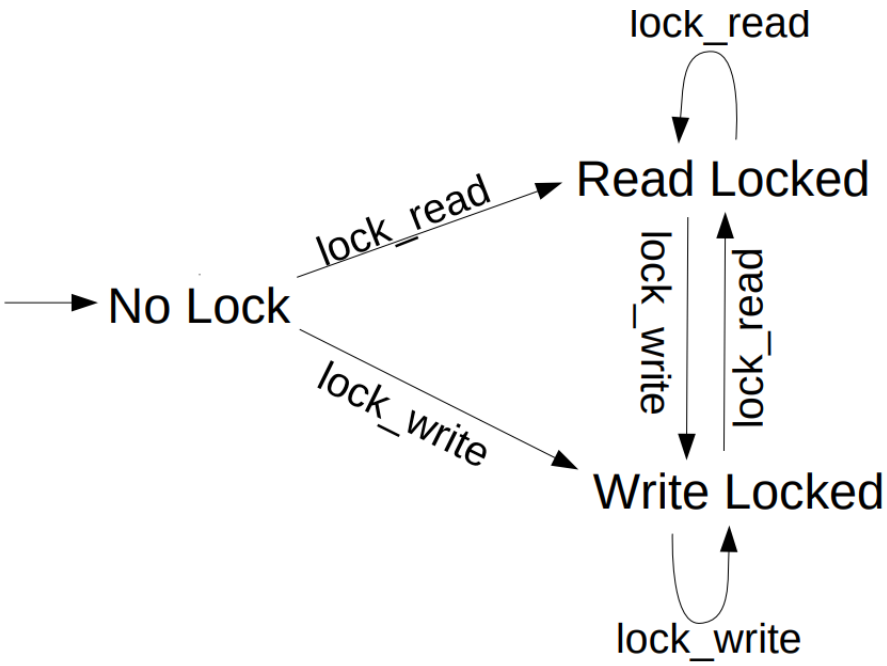

FIGURE 2 – Graphe *ServerObject*

3.1.1 *lock_read* et *lock_write*

On utilise un *Lock* de la bibliothèque *java.util.concurrent.locks* qui nous permet de spécifier l'accès avec exclusion mutuelle à cette partie du code.
La logique implémentée est conforme à celle proposée par le graphe précedent.

3.1.2 Point Délicat

Une difficulté que nous n'avions pas remarquée lors de la première version de cette classe est la suivante :
Si on *unlock* avant le *return this.obj* dans ces méthodes, un problème peut arriver : Un processus en attente peut se reveiller avant le *return*, demander *lock_write* et se retrouver avec un objet vide. Ceci crée un erreur. Afin d'éviter ce sénario, on utilise une méthode appelée *clientValidation* qui est appélée après que le client est mis à jour le verrou. *clientValidation* se charge d'*unlock*.

3.2 *SharedObject*

voici le graphe qui décrit le fonctionnement voulu pour un object côté serveur :

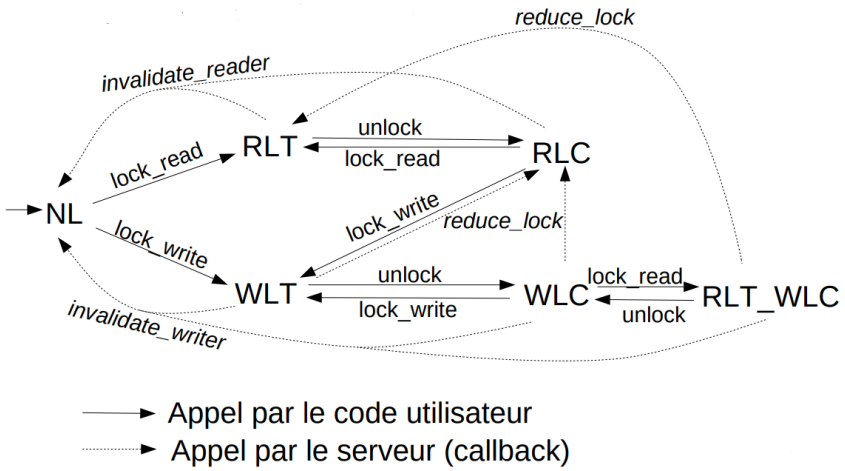

FIGURE 3 – Graphe *SharedObject*

3.2.1 *lock_read* et *lock_write*

La logique implémentée est conforme à celle proposée par le graphe précedent. On remarque l'utilisation de *clientValidation* mentionné à la partie précedente.

3.2.2 *unlock*

Cette methode permet de notifier que l'utilisation de l'objet est terminée.

3.2.3 *reduce_lock*, *invalidate_reader* et *invalidate_writer*

Ces methode permettent d'implémenter la logique donnée par le graphe précedent. Elles ont la particularité d'être *synchronized* ce qui permet de faire en sorte que les appels à ces méthodes soient en exclusion mutuelle.

3.2.4 *readResolve* (Etape 3)

Cette methode permet d'instancier la façon avec laquelle l'objet doit être sérialisé. En effet, afin d'éviter de copier un objet partagé qui inclut la référence à un autre objet partagé, on désire inclure la référence au stub de cet objet.

3.2.5 Point Délicat

L'implémentation de l'étapes 3 a été compliquée car il fallait prendre en compte le cas ou la *HashMap mapObjects* était vide. De plus, afin de pouvoir retrouver la référence à l'objet dans readResolve, nous avons ajouté une méthode dans *Client* : *getObjectById* qui nous permet de retrouver une référence à un objet à partr de son identifiant.

4 Test de surcharge du système.

Un des tests que nous avons choisi est de surcharger le système pour voir si un grand nombre de processus sur un objet partagé crée des erreur d'allocation de l'objet. Ce test ne comporte qu'une JVM et test seuelement la concurrence dessus. Il se décrit comme ceci :

- On lance N processus sur un objet (Un entier) en parallèle.
- Chaque processus lit la valeur de l'objet et ajoute 1.
- On attends qu'ils finissent tous et on lit la valeur finale de l'objet.
- On attends de lire N.

4.1 Résultats

Voici les résultats obtenus avec la configuration suvante :

- Processeur : AMD Ryzen 5 3600 6-Core Processor 3.60 GHz

— RAM : 16 Go (3200MHz)

N	valeure lue	valeure attendue	nombre d'erreur	erreur relative (en %)
1	1	1	0	0
100	100	100	0	0
500	500	500	0	0
1 000	1 000	1 000	0	0
5 000	4 996	5 000	4	0.08
10 000	9 996	10 000	4	0.04
50 000	49 989	50 000	11	0.022
100 000	99 994	100 000	6	0.006

‘

4.2 Conclusion sur ce test

On remarque que le taux d'erreur diminue après 5 000 processus. Ceci est du au fait que le nombre de processus réellement en parallèle est différent de N. En effet, des processus peuvent avoir fini avant que les suivant aient été lancés. De ce fait, La valeur du taux d'erreur à prendre en compte est 0.08%. De plus, après analyse de ces erreurs, on se rend compte qu'elles sont duent à une surcharge du service *Naming*.

5 Test de surcharge de JVM

Un des tests que nous avons choisi est de surcharger le système de processus avec chacun une JVM sur un même objet partagé. Il se décrit comme ceci :

— On lance N processus qui possèdent chacun une JVM sur un objet (Un entier) en parallèle.

— Chaque processus lit la valeur de l'objet et ajoute 1.

— On attends qu'ils finissent tous et on lit la valeur finale de l'objet.

— On attends de lire N.

5.1 Résultats

Voici les résultats obtenus avec la configuration suvante :

— Processeur : AMD Ryzen 5 3600 6-Core Processor 3.60 GHz

— RAM : 16 Go (3200MHz)

N	valeure lue	valeure attendue	nombre d'erreur	erreur relative (en %)
1	1	1	0	0
5	5	5	0	0
10	10	10	0	0
50	50	50	0	0
100	100	100	0	0
250	250	250	0	0

‘

5.2 Conclusion sur ce test

On remarque que le taux d'erreur est toujours nul. Ceci nous indique que la logique de l'objet partagé implémentée fonctionne correctement.
Avec la configuration donée précedemment, il n'est pas possible de faire tourner plus de 250 JVM en même temps. d'où l'arrêt des test à $N = 250$.

6 Test de surcharge de l'étape 3

Un des tests que nous avons choisi est de surcharger le système de processus avec chacun une JVM sur un même objet partagé. Il se décrit comme ceci :

— On lance N processus qui possèdent chacun une JVM sur un objet (Un Compteur qui référence un Historique) en parallèle.

— Chaque processus lit la valeur de l'objet et ajoute 1.

— On attends qu'ils finissent tous et on lit la valeur finale de l'objet.

— On attends de lire N.

6.1 Résultats

Voici les résultats obtenus avec la configuration suvante :

— Processeur : AMD Ryzen 5 3600 6-Core Processor 3.60 GHz

— RAM : 16 Go (3200MHz)

N	valeure lue	valeure attendue	nombre d'erreur	erreur relative (en %)
1	1	1	0	0
5	5	5	0	0
10	10	10	0	0
50	50	50	0	0
100	100	100	0	0
250	250	250	0	0

‘

6.2 Conclusion sur ce test

On remarque que le taux d'erreur est toujours nul. Ceci nous indique que la logique de l'objet partagé implémentée fonctionne correctement et que les références à des objets sont bien résolues.
Avec la configuration donnée précedemment, il n'est pas possible de faire tourner plus de 250 JVM en même temps. d'où l'arrêt des test à $N = 250$.

7 Conclusion

Ce projet nous a permis d'approfondir nos connaissances en Systèmes Concurrent et Intergiciel. En effet, nous avons du faire appel à nos connaissances vuent

lors des TPs mais égallement à de la réflexion sur le fonctionnement d'un système concurrent et partagé complexe.