Rapport de Projet Données Réparties PODR

Mathieu Campan, Besson Germain, Simon Hautesserres
Groupe L1

Département Sciences du Numérique
2022-2023

Table des matières

Introduction

Ce rapport décrit nos choix de conception et les algorithmes que nous avons implémenté au cours de l'étape PODR.

1 Accès concurrents et cohérents à un objet partagé :

Nous avons repris les bases de nos projets précédents et enlevé l'intégralité du système de verrous ce qui implique qu'il n'y a plus de blocages en écriture ou en lecture : elles peuvent se faire de manière asynchrone.

1.1 Server.java

Contrairement aux premières parties du projet (PODP 1 et 2), les clients ont des copies locales qui ne sont pas forcément à jour, même pour le client ayant écrit sur l'objet. Le *Server* s'occupe de propager les écritures, tandis qu'un client obtient l'objet à jour de la part des autres clients.
Le serveur gère également les numéros de version et de nommage afin d'avoir le numéro de version de l'objet le plus récent.
Par soucis de clarté, nous avons gardé une map associant l'id de l'objet et un *ServerObject*, même si nous ne l'utilisons principalement que pour son numéro de version.

Pour ce qui est de l'initialisation des clients, le **lookup** n'est plus d'actualité et nous avons également remplacé le **create** et **register** par un **publish**. Nous partons de l'hypothèse que nous connaissons le nombre de clients qui devra se connecter au serveur : nous le stockons comme une variable statique **NB_CLIENTS**. Nous récupérons ainsi les clients dans un **Set** à chacune de leur connexion jusqu'à atteindre le nombre requis grâce à la méthode **addClient**. Suite à cela, chaque client récupère le set des clients pour pouvoir chercher une version plus récente de l'objet à lire. De plus, nous utilisons la méthode **publish** du serveur pour propager l'objet partagé d'un client au set obtenu.

1.2 Client.java :

Le *Client* se connecte au serveur grâce à la méthode **init**. Il attend alors que le nombre de clients nécessaire soit atteint. Ensuite, il récupère le set des clients, qu'il utilisera pour mettre à jour son objet lors d'un **read** grâce à une **enquête** (2 Registre régulier).
Après son écriture, un client va récupérer son nouveau numéro de version auprès du serveur, qui va propager la modification aux clients de manière asynchrone grâce à un nouveau Slave : *Client_maj_Slave* dans la méthode **mise_a_jour** pour chaque client.
Ce slave va s'occuper de faire la mise à jour asynchrone : **majAsynchrone**.

Afin de prendre en compte le pire cas possible et de montrer la robustesse de notre code, nous avons rajouté avant chaque mise à jour (donc propagation de l'écriture) une attente aléatoire afin de simuler des délais de propagation asynchrone impactants.

2 Registre Régulier :

Pour effectuer un **read**, nous lançons tout d'abord une **enquête** de *Client* qui va lancer pour chaque autre client un *Client_ enquete_ Slave* puis attendre les réponses avec un wait.
Ce slave va demander au client recherché son objet et sa version. Il rajoute ensuite la réponse pour le client d'origine grâce à la méthode **addReponse**.
Le client ayant demandé l'enquête va stocker cette réponse grâce au *Rappel_ lec* et se réveiller avec un notify après avoir reçu une réponse de 50% des clients ayant répondu. Cette approche est expliquée en détails dans la section 4.

3 Registre Atomique :

Pour gérer les accès concurrents entre un ensemble d'écrivains et un ensemble de lecteurs, nous nous sommes assurés que les accès soient linéarisables.
Pour ce faire, nous avons ajouté une classe *Rappel_ ecr* qui propage grâce à *Client_ maj_ Slave* de manière asynchrone lors d'une lecture la modification à tous les clients lui ayant répondu. Assurant ainsi que tous les clients ayant répondus (et donc étant fonctionnels à cet instant t) auront la version mis à jour de l'objet. D'autre part, nous avons mis la méthode **write** de *Server* en synchronized afin d'empêcher les écritures concurrentes, et les versions croissantes permettront d'ordonner les modifications. Ainsi nous vérifions que la mise à jour de l'objet se fasse uniquement s'il s'agit bien d'une version à jour.

4 Tests et tolérance aux pannes

Hormis les tests manuels qui fonctionnent pour quelques clients sans problèmes, nous avons choisi d'écrire un script sh (**sh lancementTest.sh**) qui lance une quantité de clients souhaitée qui vont alterner entre écriture et lecture.
Ceux-ci sont soit des *ClientNormal* qui font un certain nombre d'écriture/lecture choisis et ont une probabilité choisie de mourir également, soit des *ClientLazy* qui ont le même comportement que des normaux mais en plus attendent 5 secondes avant d'effectuer une nouvelle opération. Les deux écrivent les résultats dans un fichier **test.txt** que nous analyserons dans le fichier **demonstration.pdf**.

Nous allons dans cette section, grâce notre script de test, expliquer notre méthode pour savoir si le protocole tolère des pannes :

Nous avons décidé de réveiller le lecteur après qu'il ait reçu une réponse de 50% des clients lui ayant répondu la dernière fois (soit 50% du nombre de départ pour la première fois) afin de gérer la tolérance aux pannes. Si tous les clients lui répondent, la condition ne changera pas (bon fonctionnement dans le cas usuel) et si certains ne répondent pas pour cause de panne, la condition évolue pour ne plus les prendre en compte (en cherchant toujours au moins une réponse).
Cela nous a rajouté un nouveau problème : après avoir reçu la réponse de la moitié des clients, notre client fait une autre demande de lecture, mais reçoit des réponses de sa demande précédente. Nous avons donc rajouté un système de numéro de demandes similaire à celui des ACK pour distinguer les différentes demandes de lecture d'un même client.
Le seul défaut de cette approche est lors d'une panne générale et massive des clients dépassant les 50%. Cependant, nous supposons qu'avec une panne de cette ampleur, le fonctionnement de l'application n'est plus une priorité.
Il reste cependant une réflexion sur ce pourcentage : en effet, le diminuer à 30% augmenterait la résistance aux pannes (car moins de chance que 70% des clients tombent en panne), mais cela augmenterait le risque d'obtenir une mauvaise version. Inversement, augmenter ce chiffre nous donnerait une meilleure chance d'avoir une bonne version, mais augmenterait notre vulnérabilité aux pannes. C'est pour cela que nous avons décidé de garder 50%.

Conclusion

Les problèmes que nous avons rencontrés sont les délais de propagation des objets suite aux premiers publish : en effet, un appel trop rapide au read/write peut parfois rendre un objet null si les objets n'ont pas été propagés assez rapidement.
Nous n'avons pas voulu rajouter une attente forcée qui ne serait pas suffisante lors d'un agrandissement de l'application. Cela pourrait donc être une perspective d'amélioration dans le cas où nous avons beaucoup de clients qui écrivent simultanément dès leur création.