Rapport PDR2 (Etape 1)

François TESTU, Nicolas DREGE, Nihal BELMAKHFI

Département Sciences du Numérique - Filière HPC et Big data - 2A
UE Applications concurrentes et communicantes - Bases de données
2022-2023

Table des matières

1 Introduction

Ce projet se concentre sur l'étude de protocoles de cohérence pour des objets partagés implémentés par des copies réparties sur plusieurs machines. L'objectif est de maintenir la cohérence entre les différentes copies malgré les mises à jour asynchrones, sans avoir à diffuser et synchroniser chaque modification pour garantir une égalité stricte. Le projet consiste à compléter un protocole existant appelé PODP pour permettre une mise à jour synchronisée des copies, et à concevoir un nouveau protocole robuste capable de gérer les pannes potentielles des copies réparties.

2 Échauffement : de l'IRC à la cohérence continue

2.1 Etape 1 : rendre l'IRC actif

2.1.1 Les changements effectués

Nous avons ajouté des méthodes à l'étape 1 pour respecter le nouveau cahier des charges.

Sur le serveur

Sur le serveur les modifications suivantes ont été effectuées : Le serveur maintient une table de hachage associant un entier (id de l'objet partagé) à un ensemble de clients, les abonnés à l'objet partagé.

Une méthode void subscribe(int id, Client_itf client) ajoute le client à l'ensemble des abonnés à l'objet partagé d'identifiant id (ou créer cet ensemble lorsqu'il n'y a aucun abonné sur cet objet) Une méthode void unsubscribe(int id, Client_itf client) enleve le client de l'ensemble des abonnés à l'identifiant correspondant.

Une méthode void publish(int id) appelle la méthode de rappel *update* de chaque client abonné à id. Pour éviter un bloquage dans la boucle d'appel lorsqu'une des fonctions *update* d'un client est bloquante et doit attendre (empêchant les autres clients d'être appelés immédiatement), ces méthodes sont appelées à l'aide d'un *parallelStream*.

Sur le client

Une nouvelle interface a été ajoutée : l'interface Updater_itf qui spécifie une unique méthode update. Cette méthode sera utilisée par le client.

Sur le client : Les méthodes statiques void subscribe(int id), unsubscribe(int it), publish(int id) ont été ajoutés et appelle les méthodes du serveur correspondantes en ajoutant le client initialisé avec la méthode statique init.

Une méthode statique setUpdater(Updater_itf updater) permet d'attribuer *updater* à un attribut statique qui servira à la méthode *update* du client.

La méthode (non statique) void update() du client appelle la méthode *update* de l'*updater*. Ainsi on peut définir et changer la méthode d'*update* du client à l'aide de setUpdater, avec une classe (anonyme) implémentant Updater_itf.

Sur les objets partagés

La méthode *unlock* a été modifiée : lorsqu'un *unlock* est effectué sur un verrou en écriture, cette méthode appelle la méthode statique publish du client. On considère ici que chaque fin d'écriture correspond à une publication, même si l'objet n'a pas réellement été modifié. Il revient alors à la charge du client de poser un verrou en écriture uniquement pour réellement modifier l'objet. Il serait possible de changer ce comportement, en ajoutant des méthodes de haut-niveau dans la classe des objets partagés pour changer l'objet sous-jacent (et vérifier qu'il ait été effectivement modifié), et en retirant la possibilité d'accéder à l'objet d'une quelconque autre manière (attribut privé).

La méthode unlock est modifiée pour ne plus être complétement bloquante : le mot-clé synchronized restreignant l'exécution à un seul thread n'est plus sur la méthode mais uniquement sur la partie critique du code (le changement de verrou). Ainsi l'appel Client.publish n'est pas bloquant dans cette méthode. Cela évite le deadlock suivant : L'application unlock le verrou sur un objet partagé après une écriture et la méthode *unlock* appelle Client.publish après le changement de verrou. Un des clients abonnés demande un verrou en lecture dans sa méthode de rappel *update*. Le serveur cherche à invalider l'écrivain en cours pour donner un verrou en lecture mais l'écrivain n'a pas terminé sa méthode *unlock* puisque l'abonné n'a pas recu son verrou en lecture (*Client.publish* et *server.publish* n'ont pas terminé tant que les abonnés n'ont pas terminé les *updates* donc *unlock* n'est pas non plus terminée). Si *unlock* était complétement bloquante, on se retrouve avec un "deadlock".

Comme ici seule la partie critique du code est bloquante, on peut invalider un écrivain une fois que ces verrous ont été changés par *unlock*, même si les *updates* de clients ne sont pas finis et *unlock* non plus.

2.1.2 2 exemples d'IRC

2 IRC différents sont implémentés pour mettre en évidence la cohérence des objets partagés et la possibilité de changer le traitement par les clients abonnés. Leur utilisation et démonstration de leur fonctionnement à travers des scénarios sont présentes dans le pdf de démonstration du projet.

Pour le 1er exemple, on met à jour une led qui notifie les abonnés après une publication (qui correspond à une écriture avec le bouton write). Cette led est verte après une lecture et rouge lorsqu'une nouvelle écriture (par un client quelconque) sur l'objet partagé a été effectuée si le client était abonné et tant qu'il n'a pas effectué une nouvelle lecture. Cet IRC correspond à la classe IRC.

Pour le 2e exemple, les abonnés lisent directement la valeur après une publication. Cet IRC est implémentée à travers la classe IRC_synchrone.

3 Conclusion

Pour conclure, nous avons avons mis en place un système publier/s'abonner dans le système d'objets partagés, maintenant une cohérence entre les clients. Cependant, une panne d'un client ou du serveur entraine encore la panne du système entier. Nous chercherons à traiter ce problème dans une seconde partie.