Rapport PDR2 (Etape 3)

François TESTU, Nihal BELMAKHFI, Nicolas DREGE

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

Le but essentiel de cette dernière partie du projet est de définir et implémenter un protocole de cohérence robuste 1WnR (écritures séquentielles, lectures concurrentes), qui puisse fonctionner correctement, même si une partie des sites est défaillante.

Pour ce faire, il faut alors éviter que des processus fonctionnant correctement (sites corrects) ne se trouvent bloqués car en attente de réponses de sites tombés en panne (sites défaillants). C'est pourquoi, nous allons abandonné le schéma de syn- chronisation lecteurs-rédacteurs. En effet, il ne faut pas qu'un rédacteur ayant l'accès exclusif bloque tous les autres processus en tombant en panne. Autrement dit, le protocole va devoir gérer le fait que les accès en lecture et en écriture peuvent être concurrents, tout en garantissant le même niveau de cohérence que celui qui est obtenu avec le schéma lecteurs-rédacteurs. Le système des locks a donc été supprimé.

Le protocole tolérant aux pannes est ainsi radicalement différent du protocole précédent. Cependant, si le protocole est entièrement différent, l'architecture générale du service, ainsi que les structures de données nécessaires resteront sensiblement les mêmes.

2 Implémentation

2.1 Moniteur

Nous avons ajouté une interface Moniteur ainsi qu'une implémentation MoniteurImpl.

En ce qui concerne l'interface, elle permet notamment le contrôle des délais des réponses aux requêtes du protocole grâce à *feuVert*. Elle permet aussi l'affichage sur les opérations de lecture/écriture du protocole grâce à la fonction *signaler*.

Elle permet, de manière interactive à l'aide d'une invite de commandes de :

— suivre les débuts et fins d'opérations de lecture/écriture par les différents sites sur les différents objets dupliqués

— fixer le temps de réponse d'un ou de tous les sites (delai <nomSite> <nbSecondes>), ou de geler momentanément (délai <nomSite> "-") ou définitivement (délai <nomSite> "X") pour simuler la panne d'un site les réponses.

— avoir accès à la liste des différentes copies d'un objet dupliqué (cliche <nomObjet>)

— avoir la liste des délais de chaque site (cliche –d)

.

Le serveur contient une référence au moniteur qu'il fournit aux clients pendant leur initialisation statique afin de permettre sa bonne utilisation (utilisé dans la classe *Client* et dans la classe *SharedObject*, elle meme prenant la reference statique du client) .

2.2 Shell

Le Shell implante un client interactif. Celui-ci permet les opérations suivantes : Il peut créer un nouvel objet dupliqué dont on lui donnera une valeur initiale, lire un objet donné, écrire dans un objet donné. Il peut aussi afficher la valeur de la copie locale grâce à un get. Enfin, il peut lister les valeurs des copies locales. Le Shell, sera, comme Irc, lancé au démarrage.

2.3 Serveur

Le serveur est une classe remote. Celle-ci s'initialise avec le nombre de clients (passé en argument) et dispose d'un ensemble contenant ces clients une fois ajoutés. Elle contient une table de hachage associant les noms des objets partagés à leur identifiant entier, une table de hachage associant ces identifiants aux instances objets partagés sur le serveur (*serverObject*).

Pour ajouter des clients, le serveur dispose d'une méthode *addClient(client)*. Celle-ci est appelé par un client, ajoute ce client à l'ensemble, incrémente le compteur de client actif, et met le thread en pause si le nombre n'est pas atteint. Sinon, si tous les clients sont actifs, celle-ci réveillent les threads des clients en attente. Elle renvoie l'ensemble des clients à chaque client qui s'est ajouté.

Une méthode *publish(name, obj, reset)* permet de créer (ou renvoyer si celui ci existe deja) un objet partagé nommé *name* sur le serveur et sur chaque client. Le booléen *reset* permet de réinitialiser (recréer) un objet partagé. Elle est appelée par la méthode statique *publish* du client.

Une méthode *lookup(name)* permet de renvoyer l'identifiant associé à l'objet partagé nommé name, utilisé par les clients pour récuperer l'id correspondant à l'aide du serveur de noms.

Les méthodes *setMonitor* et *getMonitor* permettent l'acquisition et l'envoi de l'instance du moniteur (appelé à l'initilisation du moniteur pour *setMonitor*, et à l'initialisation des clients pour *getMonitor*). Si le moniteur est nul, *getMonitor* endort le thread, et *setMonitor* réveille les threads endormis. Ainsi les clients sont mis en attente tant que le moniteur n'est pas actif.

La méthode *broadcast(idObj, value, ver)* (synchronized) permet d'envoyer l'objet value de version ver pour l'objet partagé d'identifiant idObj à tous les sites pour que ceux-ci se mettent à jour. Elle crée pour cela un objet de rappel *WriteCallback* wcb (voir section correspondante), crée des threads pour chaque *client.update(idObjet, v, valeur, wcb)* à effectuer, les exécute, appelle la méthode *wcb.waitResponses()* qui endort le thread en attendant d'avoir suffisament de réponses pour considérer l'écriture terminée. Celle-ci est appelée par la méthode *write* du serveur et par la méthode *read* du client (pour le registre atomique, voir explication du protocole).

La méthode *write(idObj, value)* appelée par la méthode statique *write* du client permet de récupérer la version en cours (dans le *ServerObject* correspondant qui lui est toujours mis à la dernière version en début d'écriture), incrémente cette version, met à jour l'objet du *ServerObject* et diffuse cette version à tous les clients avec la méthode *broadcast*. Cette méthode est synchronized afin d'avoir des écritures séquentielles.

2.4 WriteCallback

Il s'agit d'un objet de rappel remote. Celui-ci s'initialise avec un nombre de réponses minimum attendus et dispose de 2 méthodes.

response() est appelé par un objet partagé (dans la méthode *update*) d'un client pour annoncer qu'il s'est mis à jour. La méthode *waitReponses* endort le thread en attendant d'atteindre le seuil. La classe compte le nombre de mises à jour et une fois le seuil atteint réveille le thread qui s'est endormi dans la méthode *waitReponses*, signalant ainsi la fin de l'écriture.

2.5 ReadCallback

Il s'agit d'un objet de rappel remote. Celui-ci s'initialise avec un nombre de réponses minimum attendus et dispose de 2 méthodes.

response(version, value) est appelé par un objet partagé (dans la méthode *reportValue*) d'un client pour donner sa valeur et son numéro de version. La classe sauvegarde l'objet de plus grande parmi les réponses recues. La méthode *getValue* endort le thread en attendant d'atteindre le seuil puis renvoie l'objet sauvegardé. La classe compte le nombre de mises à jour et une fois le seuil atteint réveille le thread qui s'est endormi dans la méthode *getValue*, signalant ainsi la fin de l'acquisition des objets des clients.

2.6 ServerObject

Cette classe correspond à un objet partagé sauvegardé sur le serveur et ne sert qu'à des fins de maintenance/correction, en se mettant à jour directement avec la dernière version lorsqu'une écriture est commencée.

2.7 SharedObject

Cette classe correspond à un objet partagé. Elle contient un identifiant entier, un objet et la version de cet objet.

Une méthode *update(version, obj, wcb)* utilise le moniteur pour simuler le bon délai du site, met à jour l'objet si la nouvelle version est supérieure à celle courante et finit par appeller la méthode *wcb.response()* pour signaler sa réception de la requête de modification.

La méthode *reportValue(rcb)* utilise le moniteur pour simuler le bon délai du site, puis renvoie sa valeur et version à l'objet de rappel en lecture rcb à l'aide de la méthode *rcb.response(version, val)*

Les méthodes *write* et *read* signalent au moniteur les requetes, puis appellent les méthodes statiques correspondantes du client.

2.8 IRC

Cette classe est similaire aux protocoles précédents du projet.

2.9 Client

Il s'agit d'une classe remote représentant un client. Elle s'initialise de manière statique avec un nom, récupère le serveur à travers le RMI, s'ajoute aux clients actifs , se met en pause en attendant la connection de tous les clients et recupère l'ensemble des clients avec la méthode du serveur *addClient* puis attend la connexion du moniteur et le récupère avec la méthode du serveur *getMonitor*.
La classe contient ainsi les objets partagés associés à leur identifiant, son nom, le moniteur, le serveur et la liste des clients.

Une méthode *initSO* permet d'initialiser un objet partagé, appelée par le serveur lors d'un *publish*.

Les méthodes *reportValue(idObj, rcb)*, *update(idObj, val, wcb)* récupère l'objet partagé associé à l'id et exécute la méthode correspondante de l'objet partagé. Elles servent d'intermédiaires entre le serveur/client et les objets partagés. Les méthodes statiques *lookup(name)* et *publish(name, obj, reset* exécutent les methode correspondante du serveur et renvoient l'objet partagé associé.

La méthode *write* appelle la méthode correspondante du serveur.

2 méthodes *read* : une pour le registre régulier *readRegular* et une pour le registre atomique *read*. Elles permettent de récuperer la dernière valeur écrite ou une valeur en cours d'écriture de manière à respecter le protocole correspondant (régulier/atomique).

readRegular crée un objet de rappel ReadCallback rcb (voir section correspondante), crée des threads pour chaque *client.reportValue(id, rcb)* à effectuer, les exécute, appelle la méthode *rcb.getValue()* qui endort le thread en attendant d'avoir suffisament de réponses pour considérer l'acquisition des valeurs terminée, et qui renvoie la valeur "lue". La méthode *read* effectue les memes etapes et une fois la valeur finale recue par l'objet de rappel, diffuse cette valeur aux clients avec la méthode *broadcast* du serveur, assurant ainsi qu'une lecture est terminée lorsque l'écriture de la valeur lue l'est.

3 Le registre régulier

3.1 Définition

Un registre régulier gère les accès concurrents entre un unique écrivain et un ensemble de lecteurs. Dans ce cadre, le résultat d'une lecture est

— la dernière valeur écrite ou la valeur de l'une des écritures en cours, si la lecture est concurrente avec une ou plusieurs écritures.

— sinon, la dernière valeur écrite.

3.2 Exemple

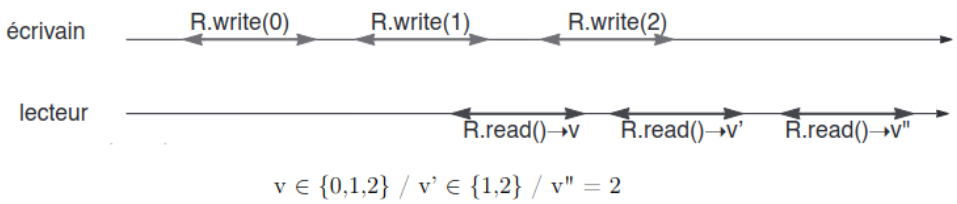

FIGURE 1 – Exemple pour le registre régulier

3.3 Le protocole : terminaison et correction

L'implémentation est décrite précedemment (lire les fonctions *readRegular* du client et *write* du serveur). Prouvons son bon fonctionnement. Pour la terminaison, on suppose que la majorité des clients ne tombe pas en panne (on prouvera que le maximum de pannes est la partie entiere de n/2). Ainsi les écritures et lectures terminent (puisqu'elles attendent une majorité de réponse)

Supposons une écriture (ver, obj) terminée suivie par une lecture non concurrente avec une prochaine écriture. Puisque l'écriture ne se termine que lorsque la majorité des clients ont répondu, plus de la moitié des clients ont la version ver de l'objet à la fin de l'écriture. Lorsque la lecture se terminera, la majorité des clients auront répondu, et nécessairement au moins un de ces clients a la version ver (il n'y a pas eu d'autres écritures entre temps). Puisque les versions sont croissantes, que la lecture prend l'objet de version la plus haute, que ver est la plus grande version (la plus récente) et qu'elle est rapportée par un des clients ayant répondu, la lecture renverra forcement l'objet de cette version. Ainsi la lecture renvoie la derniere valeure écrite dans ce cas.

Supposons maintenant un lecture en concurrence avec une écriture. Cela signifie que l'écriture n'est pas terminée au début de la lecture. Ainsi la majorité n'a pas encore reçu la dernière valeur à jour. 2 possibilités :

— soit un des sites qui répond à la demande de lecture avait déjà eu le temps de finir sa modification disposant ainsi de la dernière version (en cours d'écriture). La lecture renverra donc forcément cette version (étant la plus grande)

— soit aucun des sites des clients ayant répondu n'a eu le temps de faire la modification (ce qui est possible puisque la majorité ne l'a pas encore obtenu au début de la lecture), un de ces clients disposent forcément de la valeur de la dernière écriture terminée (puisqu'au moins la majorité dispose de celle-ci ou d'une version plus récente en cours d'écriture, mais que dans notre cas aucun n'a celle en cours) et la lecture renverra ainsi cette valeur.

Des inversions de valeurs sont possibles dans l'exemple du schéma précédent pour notre implémentation : si la premiere lecture (concurrente aux écritures) recoit un client qui a déjà écrit le 2 (qui est en cours d'ecriture, la majorité ne l'a pas encore recu) alors celle-ci renverra 2. La seconde lecture est concurrente à la 3e écriture. Comme cette écriture n'est pas terminée, la majorité n'a pas encore reçu l'ecriture du 2. Ainsi cette lecture peut recevoir la partie des clients qui ont encore un 1 et terminer. Elle renverra alors un 1. Il y a une inversion de valeurs, le 1 étant écrit avant le 2 Ainsi notre implémentation aec la méthode *readRegular* permet bien un registre régulier

4 Le registre atomique

4.1 Définition

Un registre atomique gère les accès concurrents entre un ensemble d'écrivains et un en- semble de lecteurs. (Tout site peut être alternativement lecteur et écrivain). Dans ce cadre, le protocole assure que les accès sont linéarisables, c'est-à-dire que pour tout ensemble d'accès concurrents, il existe un entrelacement S de ces accès (i) qui donne le même résultat final, pour ce qui est de la valeur du registre (ii) qui respecte la chronologie des opérations non concurrentes, c'est-à-dire que si un accès a1 se termine avant le début d'un accès a2, alors a1 précède a2 dans S. (iii) tel que tout lecture de S renvoie la dernière valeur écrite dans S (légalité).

4.2 Exemple

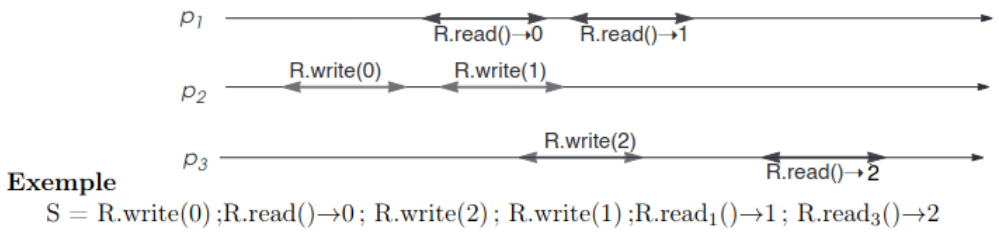

FIGURE 2 – Exemple pour le registre atomique

4.3 Le protocole : terminaison et correction

Le *write* ne change pas entre l'implémentation des 2 registres. Nous nous intéressons ici à la primitive de lecture *read*. Seule une diffusion de la valeur retenue à une majorité de clients vient s'ajouter au read. Les preuves précédentes s'appliquent au nouveau protocole (en particulier la terminaison). Pour montrer le reste de la bonne correction de l'algorithme mis en jeu on prouve le résultat suivant (non interversion des valeurs) : Si une lecture L2 suit une lecture L1 et si L1 retourne une valeur v1 écrite pendant une opération O1 et L2 retourne une valeur v2 écrite pendant une opération O2 alors O2 est après O1.

Preuve : L1 a retourné une valeur v1 de version i1, diffusé (=écriture) à la majorité des clients avant le début de L2, puisque L2 suit L1. Nécessairement, la version de la valeur retenue par L2 est plus grande que celle de v1 : pour les memes raisons que le registre régulier lecture apres écriture (les versions sont croissantes, les objets sauvegardés correspondent à la version la plus grande, L2 est après l'écriture imposée par L1 et L1 et L2 ont une intersection non vide de clients ayant répondu puisque les 2 considerent la majorité des clients pour terminer). Ainsi, comme on attribue des versions croissantes pour chaque nouvelle écriture, nécessairement O2 est après O1.

5 Choix de la majorité

On cherche à faire en sorte que nos algorithmes fonctionnent sous le maximum de pannes possibles et soient pertinents. Si on note n_p le nombre de pannes possibles, on veut que la lecture et l'écriture attendent la réponse du plus de sites possibles soit $n - n_p$ sites (avec n le nombre de sites total). Pour qu'une lecture renvoie la bonne valeur après une écriture, il faut qu'un des n-np sites ayant répondu à l'écriture réponde aussi à la lecture. On ne peut imposer le choix des clients qui vont répondre (puisqu'on ne sait pas qui va tomber en panne). Il faut donc qu'il y ait assez de clients qui répondent aux deux (plus de la moitié pour avoir une intersection non vide). Ainsi il faut que $n - n_p > n/2$ d'ou $n_p < n/2$. Le maximum de pannes possibles dans ce cas est la partie entiere de $n/2$ et le nombre pertinent d'attentes de requetes correspond à la majorité.

6 Scénario de démonstration

Sans panne, fonctionnement séquentiel, registre atomique :

— `java Server 3 &`
Le serveur se lance.

— `java Irc test1 &`
Le client attend.

— `java Irc test2 &`
Le client attend.

— `java Irc test3 &`
Les clients sont prêts et attendent le moniteur.

— `java MoniteurImpl`
Les clients sont lancés avec l'interface textuelle et le moniteur aussi.

— `delai * 1 (sur la commande d'invite du moniteur)`
tous les sites ont un délai d'une seconde

— `trace IRC (sur la commande d'invite du moniteur)`
le moniteur est pret à afficher les changements d'etats des objets partagés.

— `écrire dans un des IRC et appuyer sur write`
le moniteur affiche d'abord les textes de vide des clients avec la version 0. Le serveur lance le write sur l'objet d'id 0 avec la version 1. Attente de 2 réponses pour terminer l'écriture. Réponse recue x3 au bout d'1 sec. Le minimum est atteint, l'écriture est terminée (éventuellement réception des sites restants). Le moniteur affiche les objets de tous les sites. Tous les sites ont la derniere valeur

— `appuyer sur le bouton lire apres fin de l'ecriture`
Le moniteur affiche les états courants des objets des clients. Attente du minimum (2) de réponses de la lecture. La majorité a répondu, diffusion de la valeur choisie (la plus grande) à tous les sites. Attente du minimum de réponse pour l'ecriture. Affichage par le moniteur des états courants. Le texte affichée sur l'IRC correspond à la derniere écriture

Sans panne, fonctionnement séquentiel, registre régulier (pour ca changer Client.read dans SharedObject par Client.readRegular) : pareil que dans le cas précédent en enlevant la diffusion de l'écriture apres la lecture

Sans panne, fonctionnement concurrent, registre regulier/atomique : effectuer les commandes en mettant différentes valeurs de délais entre les sites. Effectuer le cas du schema du registre régulier avec une écriture 0 terminée puis des écritures/lectures 1, 2 concurrentes

Sans panne, fonctionnement concurrent, registre regulier, interversion de valeur : effectuer le meme principe que le paragraphe juste au dessus en changeant les délais des sites pour mettre en avant le cas exposé dans le dernier paragraphe de la section registre régulier.

Avec panne, registre atomique : faire les commandes précédentes puis changer le délai en delai X (= panne simulée) d'un site ou plusieurs (au max n/2 - 1, sinon le protocole tombe en panne d'apres la section sur le registre atomique). Constater que le protocole renvoie les bonnes valeurs

avec des écritures/lectures séquentielles et concurrentes.

7 Conclusions

7.1 Conclusion sur le projet

En conclusion, le projet de données réparties sur les objets partagés SharedObject et ServerObject axé sur les pannes et la cohérence présente des solutions efficaces pour gérer les problèmes liés à la répartition des données et à la maintien de leur cohérence dans un environnement distribué.

Grâce à l'utilisation de SharedObject, qui permet la synchronisation des objets partagés entre différents processus ou machines, le projet offre une solution robuste pour assurer la cohérence des données. L'implémentation des 2 nouvelles solutions avec les registres (régulier ou atomique) garantit l'intégrité des données partagées, même en cas de pannes ou de conflits d'accès concurrents.

En intégrant des mécanismes de détection et de récupération des pannes, le projet aborde également les défis liés à la fiabilité et à la disponibilité des données réparties. Les mécanismes de redondance et de réplication garantissent que les données restent accessibles même en présence de pannes matérielles ou logicielles.

7.2 Conclusion personnelle

Ce projet était difficile à mettre en place. La principale difficulté étant la compréhension du sujet. Il était compliqué de visualiser le role de chaque classe et ainsi pouvoir l'implémenter correctement. La discussion et la réflexion nous ont permis de venir à bout de ce projet et nous en sommes très satisfaits !