NoSQL avec KVStore

Administration des bases de données réparties

NI401 - ABDR

Hubert Naacke

5 janvier 2014

Robin Keunen
3303515
Clément Barbier
3061254

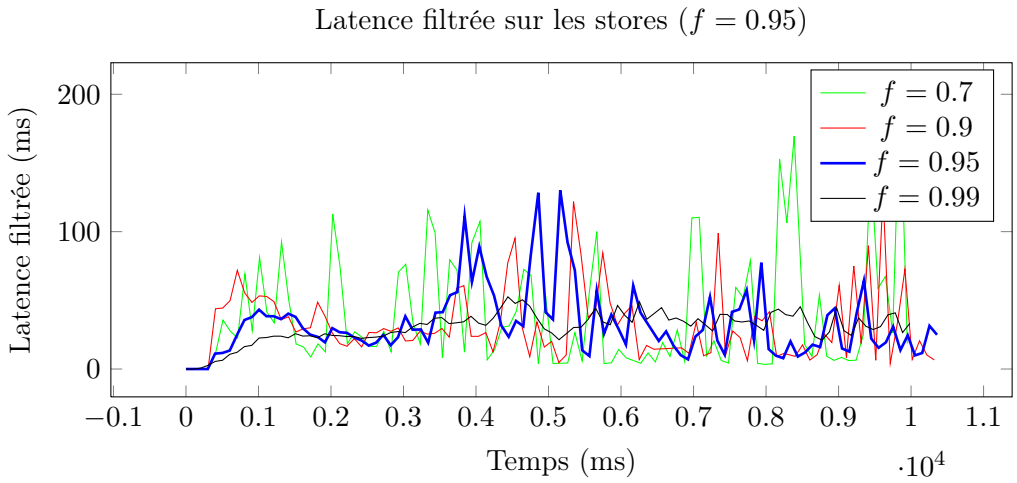

FIGURE 1 – Latence filtrée sur les stores 1, 3 et 5. Les stores 1 à 4 subissent une charge de 15 applications. Le store 1 subit la charge additionnelle de 5 applications. Pas de balance de la charge.

1 Transaction et concurrence

Cette section présente la solution du tme KVStore. Le code se trouve dans le projet kvstore dans les packages `tme1.ex1` et `tme1.ex2`.

Exercice 1

La classe `init` permet d'initialiser le store afin d'obtenir des résultats reproductibles.

A1 Dans cet exercice, nous ne devions pas tenir compte des accès concurrents au données. Le programme lit simplement la valeur stockée à la clé $P1$, incrémente ce qu'il a lu et l'écrit à la clé $P1$. Cette solution n'est plus viable dès que deux programmes manipulent la même donnée simultanément. En effet, si les programmes a et b lisent n simultanément, ils écriront chacun $n + 1$ en base alors que la valeur aurait du être incrémentée deux fois. La valeur finale devrait être $n + 2$.

Ce résultat est montré par l'expérience : en lançant deux programmes `A1` qui lisent et incrémentent 1000 fois la valeur stockée à P1, on obtient une valeur finale en $P1$ de 1261, 1269 et 1289 (3 exécutions consécutives) au lieu de 2000.

A2 Dans `A2`, le programme vérifie si la valeur stockée en base est fraiche avant d'écrire. On vérifie la fraicheur grace à la fonction `putIfVersion`. Cette fonction n'écrit en base que si la version de la donnée en base correspond à la version de la donnée que nous y avions lue. Si la version est périmée, le programme relit la valeur et tente à nouveau de l'incrémenter et de la réécrire.

Cette version retourne les résultats attendus : deux programmes `A2` exécutés simultanément incrémentent effectivement 2000 fois la valeur de $P1$.

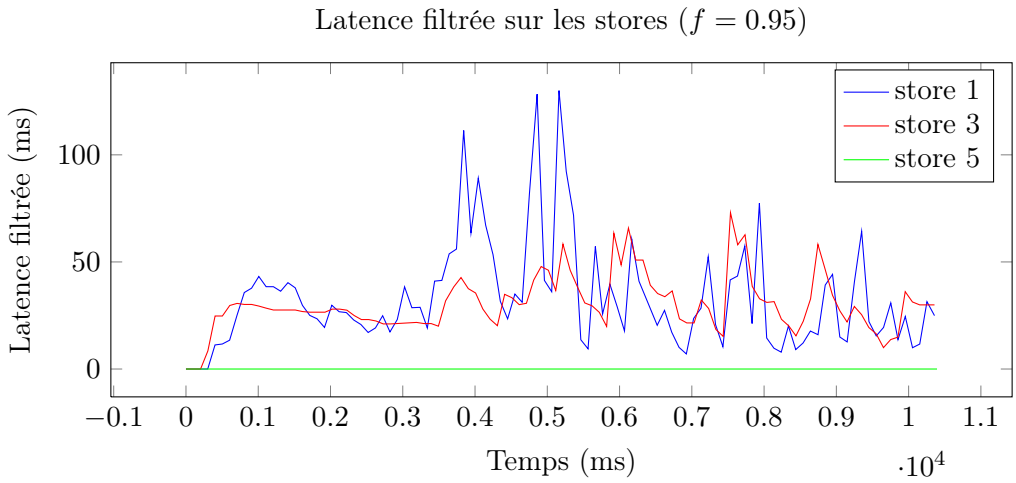

FIGURE 2 – Latence filtrée sur les stores 1, 3 et 5. Les stores 1 à 4 subissent une charge de 15 applications. Le store 1 subit la charge additionnelle de 5 applications. Pas de balance de la charge.

Exercice 2

M1 Ce programme est identique à `A2`, il lit les valeurs de $P0$ à $P4$ en base, les incrémente et les écrit en base. Si on vérifie la fraicheur des données avant de réécrire, il n'y aura pas de problème de cohérence de données. En effet, vu que les données écrites (de $P0$ à $P4$) sont indépendantes les unes des autres, l'ordre d'écriture de deux programmes n'affecte pas la valeur des données finales.

M2 En exécutant deux programmes `M2` en série, on obtient une valeur finale de 2000. En exécutant deux programmes `M2` en parallèle, les résultats divergent, on obtient par exemple lors de 3 expériences consécutives : 2054, 2056 et 2044.

Cette erreur est provoquée par la non-atomicité des écritures en base. L'écriture de $max+1$ sur P0..5 peut être interrompue par l'écriture d'un programme concurrent. Quand l'écriture est interrompue en $P3$ par exemple, la valeur $max + 1$ a déjà été écrite en $P0$, $P1$ et $P2$. Le programme recommence cette itération (lecture et écriture des 5 produits). Au final, les valeurs auront été incrémentées deux fois lors de cette itération. Voir la table 1 en annexe pour un exemple d'exécution.

Transactions

Transactions et équilibrage de charge

étape 1 a

1 clients : 3120080 2 clients : 4293081 3 clients : 5785398 4 clients : 8064602 5 clients : 9099661 6 clients : 10813041 7 clients : 13126199 8 clients : 16086278 9 clients : 18252757 10 clients : 20314303

étape 1 b

75695802

TABLE 1 – Exemple d'exécution de M2 sans le mécanisme des transactions. Le programme *a* est interrompu avant sa dernière écriture. *a* recommence l'itération lecture/écriture. Au final, les valeurs sont incrémentées deux fois lors d'une seule itération.

programme a	programme b
read(P4) = 73	
write(P0, 74)	
write(P1, 74)	
write(P2, 74)	
write(P3, 74)	
	read(P0) = 74
	read(P1) = 74
	read(P2) = 74
	read(P3) = 74
	read(P4) = 73
	write(P0, 75)
	write(P1, 75)
	write(P2, 75)
	write(P3, 75)
	write(P4, 75)
write(P4, 74) ! !	
read(P0..4) = 75	
write(P0..4, 76)	

Etape 2

Le déplacement d'un profil Px d'un store Si à un store Sj n'est pas une opération atomique par définition.

Imaginons un fonction moveProfil prenant en paramêtre un store source (Si), un store cible (Sj) et l'identifiant du profil à déplacer (Px). Cette fonction est une transaction qui fait éxécute une suite d'opérations : - Lire le profil Px sur Si. - Copier le profil Px sur Sj. - Supprimer le profil Px sur Si.

On ne peut pas se permettre de vérouiller l'accès à un profil durant le déplacement (généralement d'une longue durée) car celà altérerait trop les performances.

Celà implique que le profil Px, en cours de déplacement, peut être sollicité par des applications clientes pour des lectures, modifications, ajouts d'objets au profil ce qui peut occassioner des problêmes de consistence des données.

Il faut à tout moment (même pendant le déplacement) être capable de : - Accéder à la dernière version du Profil Px et ses objets. - Modifier les profils sans que ces modifications soient perdues durant le déplacement. - Insérer de nouveaux objets dans le profil Px tout en assurant la validité et l'unicité des clés crées.