NoSQL avec KVStore

Administration des bases de données réparties

NI401 - ABDR

Hubert Naacke

13 janvier 2014

Robin Keunen
3303515
Clément Barbier
3061254

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 $P1$.

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 A en annexe pour un exemple d'exécution.

M3 Pour que les résultats soient cohérents, il faut que les opérations de mises à jours soient atomiques. Les transactions atomiques sont implémentées dans la classe `Transaction`. Les opérations ne peuvent être exécutées de façon atomiques que si les clés ciblées partagent la même clé majeur. C'est le cas ici, nous ne manipulons que

des éléments de la même catégorie. La clé des produits est composée d'une partie majeure correspondant à la catégorie et d'une clé mineure correspondant au produit.

2 Equilibrage de charge sur plusieurs KVStores

2.1 Structure de notre solution

Nous avons essayé de rajouter une couche d'abstraction au dessus des KVStores : les applications clientes s'adressent à un élément unique (*Singleton*), le maitre : le `StoreMaster`. Le maitre redistribue la charge aux différents stores. Puisque l'objet est unique et centralisé, il est possible que les opérations subissent un effet de goulot d'étranglement. Cependant, un maitre décentralisé aurait été trop long à implémenter pour ce projet. Les effets de goulot sont limités car les accès au maitre peuvent être concurrents.

Cette section détaille les éléments logiciels de notre solution. La figure 1 illustre la structure de l'application.

project

Item
`Item` modélise l'objet à ajouter à la base. Il contient cinq champs numériques et cinq champs `String`.

ServerParameters
`ServerParameters` encapsule les paramètres nécessaires pour accéder à un KVStore (nom, IP et port).

ConfigsServer
`ConfigsServer` mémorise les paramètres correspondant aux serveurs lancés par le script fourni avec le projet.

project.master

StoreMaster
`StoreMaster` est l'interface fournie aux applications clientes. Il maintient une liste de `StoreController`s (un par KVStore concret). Ces `StoreController`s fournissent une interface aux KVStore. `StoreMaster` délègue les opérations du client à un des controleurs. La délégation est faite par un `StoreDispatcher` au moyen du numéro du profil manipulé.

MissingConfigurationException
Avant d'être utilisé, le `StoreMaster` doit être configuré. La configuration consiste à lui passer une liste d'instances de `oracle.kv.KVStore`. Si cette configuration n'a pas été faite avant l'instantiation, cette exception est levée.

StoreSupervisor
`StoreSupervisor` est un Runnable lancé au moment de l'instantiation du maitre. Cet objet consulte les statistiques de chaque contrôleur et déclenche un déplacement de profil pour équilibrer la charge.

project.master.dispatchers

StoreDispatcher
Les `StoreDispatcher`s servent à attribuer un contrôleur à chaque profil. La fonction *getStoreIndexForKey* retourne un entier qui servira d'index dans la liste de controleurs du `StoreMaster`.

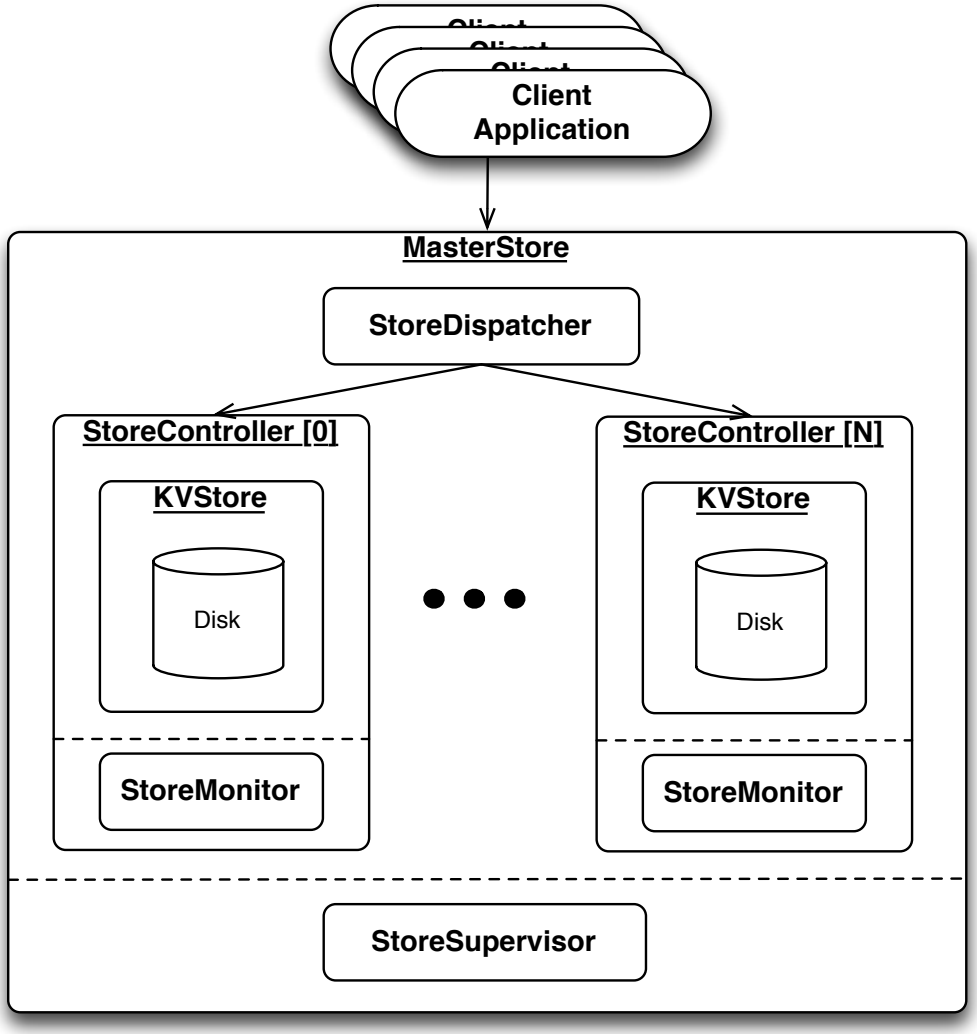

FIGURE 1 – Structure générale de l'application

SingleStoreDispatcher
Utilisé dans l'étape 1 de la résolution. Retourne l'index 0 quelque soit le profil en entrée.

TwoStoreDispatcher
Utilisé dans l'étape 2 de la résolution. Retourne 0 pour les profils pairs et 1 pour les profils impairs.

MultipleStoreDispatcher
Dispatcher final. Il retourne $index = (profil \mod n)$ où n est le nombre de stores. `MultipleStoreDispatcher` supporte en outre la fonction *manualMap*. Cette fonction permet d'assigner manuellement un store à un profil. Lorsque le `StoreMaster` demande un index pour un profil,
`MultipleStoreDispatcher` fait un lookup dans sa table de profil. Si le profil est dans la table, il retourne l'index associé, sinon il retourne l'index selon la fonction indiquée plus haut. Ce *dispatcher* est le dispatcher

par défaut.

project.store

StoreController
`StoreController` est un *wrapper* pour les KVStore concrets (`oracle.kv.KVStore`). Il implémente les opérations du KVStore dont nous avons besoin.

ProfileTransaction
`ProfileTransaction` est créé par le controleur pour créer une transaction atomique. Cette classe est instanciée pour un profil donné, le controleur lui ajoute des opérations et lance l'exécution.

StoreMonitor
`StoreMonitor` est un `Runnable` permettant de récolter des statistiques sur les opérations. Cette classe gère l'attribution des indexes aux `Item`s insérés.

TransactionMetrics
Implémente `oracle.kv.stats.OperationMetrics`. Cette classe sert à collecter les statistiques sur les transactions effectuées. Les statistiques sont mises à jour pour chaque transaction. Nous avons ajouté le champ *filteredLatency* dont nous parlerons plus tard.

project.application

ClientApplication
`ClientApplication` est un `Callable`. Les résultats sont retournés grâce à `ClientApplicationResult`. Les applications reçoivent une liste de profils cibles à l'instanciation. Cette liste permet de contrôler la pression exercée sur chaque store.

ClientApplicationResult
Cette classe encapsule les valeurs de retour : des statistiques sur les transactions effectuées par l'application.

tests

Tests
`Tests` lance les opérations nécessaires pour les étapes 1 et 2 du projet. Il initialise les dispatcher correspondant au scénario : `SingleStoreDispatcher` pour l'étape 1 et `TwoStoreDispatcher` pour l'étape 2.

LoadBalancingTest
Cette classe teste le `StoreMaster`pour déclencher une répartition de la charge. Le `StoreMaster` contrôle 5 KVStores (indexes de 0 à 4). Elle lance 10 applications ciblant [1] les stores 0 à 3 et 10 applications ciblant le store 0. On remarque que le store 4 n'est pas du tout utilisé.

2.2 Etape 1a

L'étape 1a permet d'étudier la réponse d'un store unique quand il est au service de 1 à 10 clients. Les résultats sont illustrés à la figure 2. On remarque que la latence augmente linéairement en fonction de la charge.

1. Les profils sont choisis de telle sorte que la fonction de répartition du dispatcher cible les stores 0 à 3

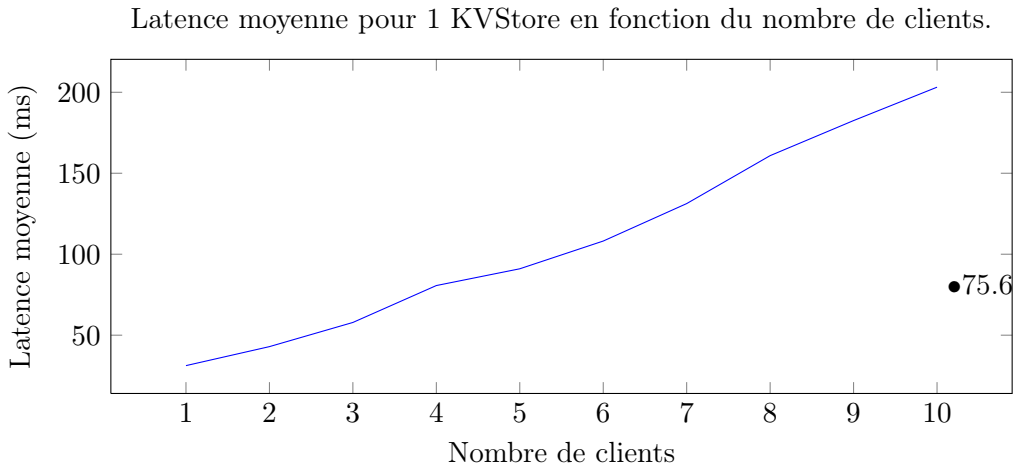

FIGURE 2 – Latence moyennes obtenues sur 10 tests consécutifs. Le point représente la latence obtenue en répartissant la charge sur deux KVStores.

2.3 Etape 1b

A l'étape 1b, la charge de 10 clients est répartie sur deux stores. On obtient une latence de $75,69\ ms$ (moyenne sur 10 tests). La réduction de la latence est évidente : plus de deux fois moins longue ! On obtient un temps de latence équivalent à celui que nous avions obtenu à l'étape 1a pour 5 clients. Les deux stores ont des latences équivalentes.

2.4 Etape 2

Pour déplacer une donnée d'un store à un autre, nous avons besoin du numéro du profil en question, du store source et du store destination. Il faut aussi mettre au point une politique de transfert : quand bouger quel profil ? C'est `StoreSupervisor` qui déclenche le transfert et c'est `StoreMaster` qui effectue le transfert.

Le déplacement d'un profil Px d'un store Si à un store Sj n'est pas une opération atomique. Imaginons un fonction *moveProfil* prenant en paramètre un store source (Si), un store cible (Sj) et l'identifiant du profil à déplacer (Px). Cette fonction est une transaction qui :

- lit le profil Px sur Si,
- copie le profil Px sur Sj,
- supprime le profil Px sur Si.

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

Cela 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 occasionner des problèmes de consistence des données. Ces problèmes de consistence n'ont pas été résolus.

Il faut à tout moment (même pendant le déplacement) être capable d'assurer les opérations suivantes :

- accès à la dernière version du profil Px et ses objets,
- modification des profils sans que ces modifications soient perdues durant le déplacement.

– insertion de nouveaux objets dans le profil Px tout en assurant la validité et l'unicité des clés crées.

2.5 Etape 3 : politique d'équilibrage

Notre approche consiste à repérer quels sont les stores surchargés et ceux qui sont sous utilisés. Pour cela, le `StoreSupervisor` parcourt les controleurs et décide si ils sont dans l'état UNDERLOADED, LOADED ou OVERLOADED. Cette décision se base sur la moyenne m et la déviation standard σ de la latence lissée (cf section 2.5.1).

L'idée de base consistait à considérer tous les stores dont la latence est supérieure $m + 2\sigma$ comme surchargés et ceux dont la latence est inférieure à $m - 1\sigma$ comme sous-chargé. Mais le petit nombre de store n'est pas suffisant pour obtenir des σ pertinents. Nous nous sommes alors contentés de mettre des bornes empiriques sur les latences pour déterminer les états : surchargé si latence $> m + 40\ ms$ et sous-chargé si latence $< m - 40\ ms$. Quand un store est surchargé, on déplace le profil le plus volumineux vers le store (sous-chargé) le moins chargé.

2.5.1 Filtre exponentiel de la latence

Nous avions commencé par mesurer la charge de chaque store grâce à la latence moyenne. Ce paramètre n'est pas satisfaisant car il n'est pas assez réactif : il prend en compte toutes les latences passées. Si dans le passé, le store a été soumis à une charge raisonnable, il faudra un certain temps avant de détecter l'augmentation de la charge. De même, la latence moyenne ne permet pas de détecter les pics de charge.

Pour remédier à ce problème, nous avons décidé d'implémenter un filtre exponentiel. Ce filtre permet de tenir compte de la latence courante et des latences passées. L'influence des latences passées est contrôlée grâce au facteur d'oubli f.

Modèle d'état

La latence mesurée en t se note $l[t]$, la latence lissée calculée en t se note $y[t]$. Afin de ne pas devoir stocker les latences passées et de permettre un calcul efficace de la latence, nous maintenons une variable d'état $x[t]$. L'équation de sortie et de mise à jour de l'état sont :

$$\begin{cases} y[t] = f\ x[t-1] + (1-f)\ l[n] \\ x[t+1] = y[t] \end{cases}$$

ou plus simplement :

$$y[t] = f\ y[t-1] + (1-f)\ l[n]$$

Facteur d'oubli

Le choix du facteur d'oubli est important, il détermine les influences relatives de la mesure courante et des mesures passées. Comme on le constate à la figure 3, pour un petit facteur d'oubli ($f = 0,7$), les oscillations sont grandes. Pour un grand facteur d'oubli ($f = 0,99$), le signal est lissé, tous les pics sont effacés.

On désire déplacer les profils si la charge augment significativement et durablement. Nous avons choisi $f = 0.95$ afin d'éviter le déplacement des profils à chaque pic tout en restant réactif.

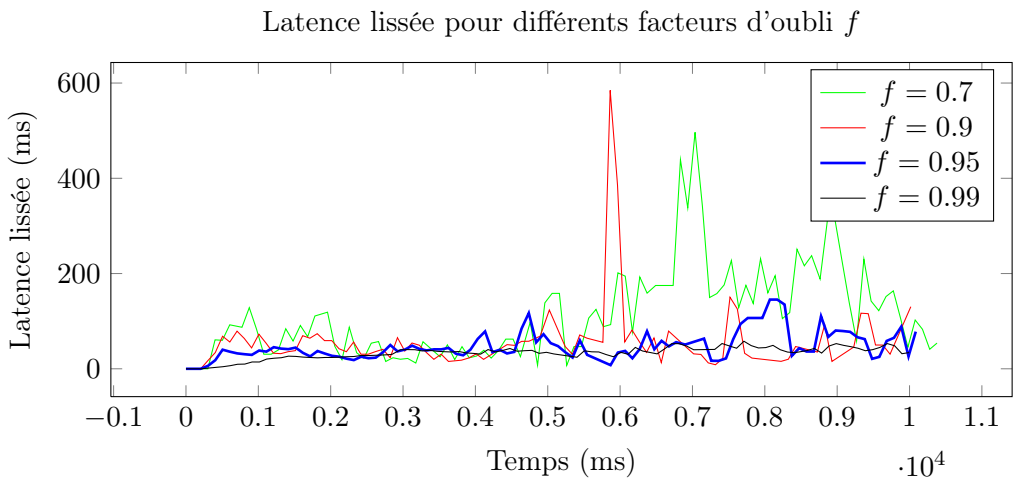

FIGURE 3 – Latence lissée pour différents facteurs d'oubli f. Testé sur le store 1.

2.6 Etape 3 : tests

A la figure 4, on peut voir les latences pour les stores 1, 3 et 5 sans balance de la charge. On constate que le store 1 résiste bien à la charge au début mais est de plus en plus lent. Plusieurs pics de latence sont significativement plus grand que pour les autres stores. Des statistiques plus détaillées [2] se trouvent à la section B en annexe. Le store 1 a traité 2696 transactions et 270000 opérations tandis que le store 3 a traité 515 transactions et 51600 opérations.

La figure 5 illustre les résultats lorsque la balance de charge est activée. Le transfert d'un profil du store 1 au store 5 commence à partir de 5 secondes d'exécution. On remarque que le store subit la charge du transfert et commence à recevoir des transactions. Au final, il semble que les pics sont atténués et que la charge est mieux répartie.

Les résultats sont beaucoup plus concluants sur des périodes d'exécution de 30 secondes comme le montrent les figures 6 et 7. Sans balance de charge, les latences mesurées sont extrêmement variables. Le store 1 a une latence beaucoup plus élevée que les autres stores et le store 5 reste inutilisé.

En activant la balance de la charge par contre, les résultats sont meilleurs. Le store 1 est d'abord presque 2 fois plus lent que les autres stores. Ensuite, le store 5 prend le relais et la latence du store 1 descend graduellement. A la fin de l'exécution, les latences semblent converger !

Par contre, les latences du store 5 sont énorme, cela est causé par le haut taux d'abandon de la transaction de déplacement.

3 Conclusion

Les transferts ne sont pas assez rapide pour atténuer les pics mais la charge est bien répartie et le store 1 est soulagé. Voici quelques points qui pourraient être améliorés :

– Le calcul des états des stores pourrait être affiné, nous aurions voulu décider si

2. Nombre de requêtes (transactions), nombre d'opérations, latence minimale, latence moyenne et latence maximale.

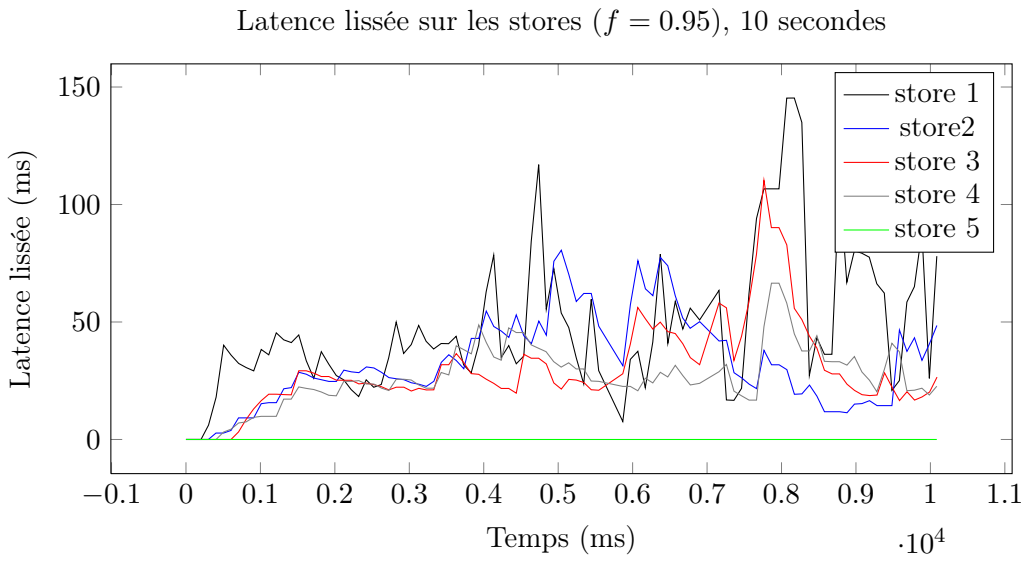

FIGURE 4 – Latence lissée sur les stores 1, 3 et 5. Les stores 1 à 4 subissent une charge de 10 applications. Le store 1 subit la charge additionnelle de 10 applications. Pas de balance de la charge.

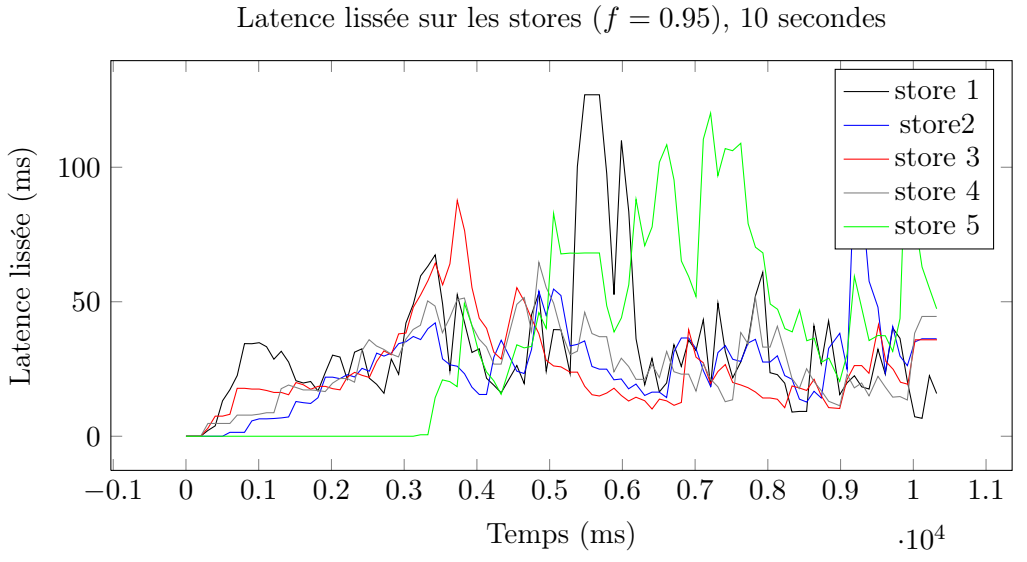

FIGURE 5 – Latence lissée sur les stores 1, 3 et 5. Les stores 1 à 4 subissent une charge de 10 applications. Le store 1 subit la charge additionnelle de 10 applications. Balance de la charge.

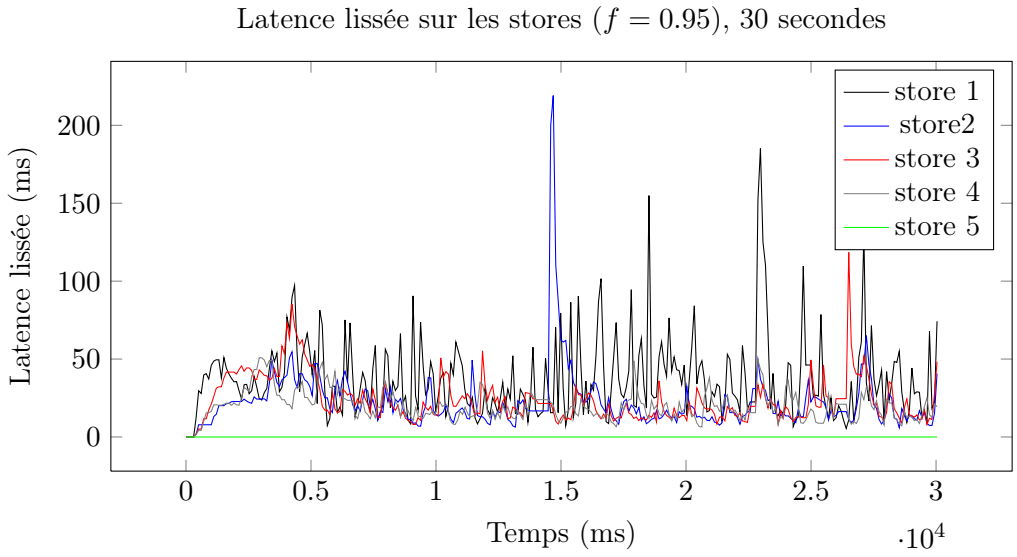

FIGURE 6 – Latence lissée sur les stores 1, 3 et 5. Les stores 1 à 4 subissent une charge de 10 applications. Le store 1 subit la charge additionnelle de 10 applications. Pas de balance de la charge.

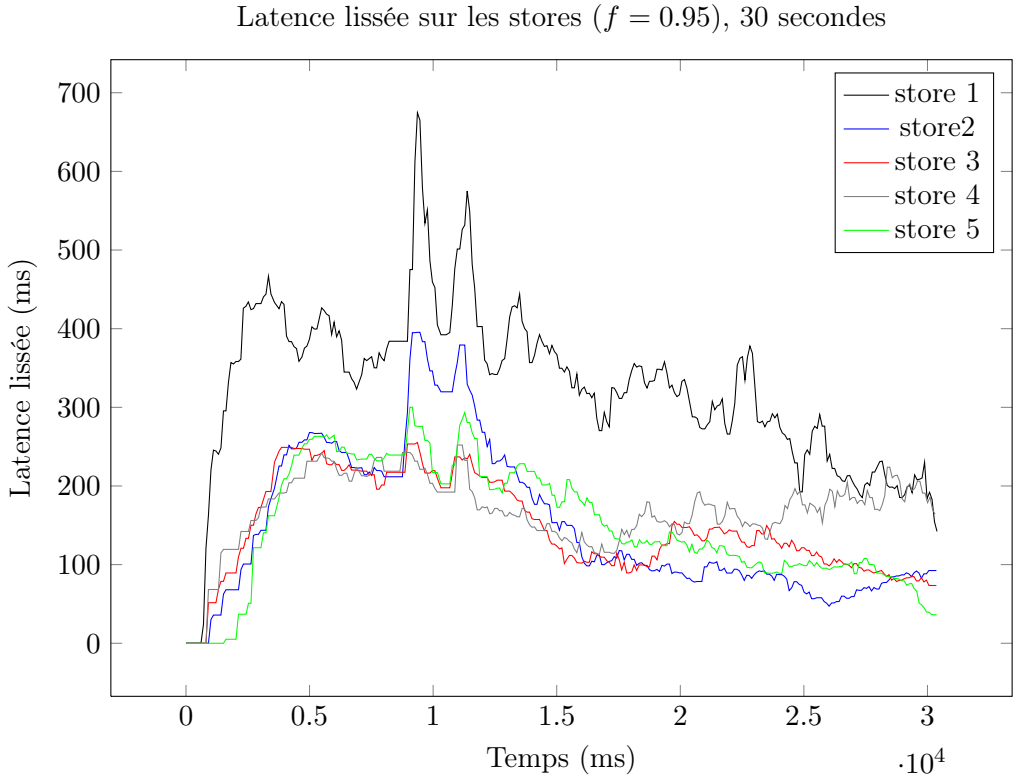

FIGURE 7 – Latence lissée sur les stores 1, 3 et 5. Les stores 1 à 4 subissent une charge de 10 applications. Le store 1 subit la charge additionnelle de 10 applications. Balance de la charge.

un store est surchargé en fonction de l'éloignement d'une latence par rapport à la latence moyenne.
- Il serait aussi possible d'investiguer un modèle plus prédictif (en prenant par exemple en compte la dérivée de la latence).
- Enfin, la consistence des opérations pendant un transfert n'est pas assurée.

Malgré ces remarques, notre approche semble concluante, la charge est effectivement répartie et la latence des stores surchargés est diminuée.

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)	

A Exemple d'exécution.

B Résultats d'exécution

B.1 Sans balance de charge - 10s

```
store 0 - doProfileTransaction - 2696 requests
store 0 - doProfileTransaction - 270000 operations
store 0 - doProfileTransaction - min latency 2 ms
store 0 - doProfileTransaction - avg latency 50.497562 ms
store 0 - doProfileTransaction - max latency 725 ms
store 0 - doProfileTransaction - filtered latency 77.63356 ms

store 1 - doProfileTransaction - 514 requests
store 1 - doProfileTransaction - 51500 operations
store 1 - doProfileTransaction - min latency 2 ms
store 1 - doProfileTransaction - avg latency 27.470825 ms
store 1 - doProfileTransaction - max latency 533 ms
store 1 - doProfileTransaction - filtered latency 13.157293 ms

store 2 - doProfileTransaction - 515 requests
store 2 - doProfileTransaction - 51600 operations
store 2 - doProfileTransaction - min latency 3 ms
```

```
 store 2 - doProfileTransaction - avg latency 24.852741 ms
 store 2 - doProfileTransaction - max latency 540 ms
 store 2 - doProfileTransaction - filtered latency 15.248835 ms

 store 3 - doProfileTransaction - 496 requests
 store 3 - doProfileTransaction - 49600 operations
 store 3 - doProfileTransaction - min latency 3 ms
 store 3 - doProfileTransaction - avg latency 25.786291 ms
 store 3 - doProfileTransaction - max latency 401 ms
 store 3 - doProfileTransaction - filtered latency 14.4859085 ms

 store 4 - doProfileTransaction - 0 requests
 store 4 - doProfileTransaction - 0 operations
 store 4 - doProfileTransaction - min latency -1 ms
 store 4 - doProfileTransaction - avg latency -1.0 ms
 store 4 - doProfileTransaction - max latency -1 ms
 store 4 - doProfileTransaction - filtered latency 0.0 ms
```

B.2 Avec balance de charge - 10s

```
store 0 - doProfileTransaction - 1655 requests
 store 0 - doProfileTransaction - 165800 operations
 store 0 - doProfileTransaction - min latency 2 ms
 store 0 - doProfileTransaction - avg latency 34.26758 ms
 store 0 - doProfileTransaction - max latency 691 ms
 store 0 - doProfileTransaction - filtered latency 12.916809 ms

 store 1 - doProfileTransaction - 453 requests
 store 1 - doProfileTransaction - 45300 operations
 store 1 - doProfileTransaction - min latency 3 ms
 store 1 - doProfileTransaction - avg latency 29.86534 ms
 store 1 - doProfileTransaction - max latency 631 ms
 store 1 - doProfileTransaction - filtered latency 36.235928 ms

 store 2 - doProfileTransaction - 442 requests
 store 2 - doProfileTransaction - 44200 operations
 store 2 - doProfileTransaction - min latency 3 ms
 store 2 - doProfileTransaction - avg latency 27.81221 ms
 store 2 - doProfileTransaction - max latency 397 ms
 store 2 - doProfileTransaction - filtered latency 35.85543 ms

 store 3 - doProfileTransaction - 444 requests
 store 3 - doProfileTransaction - 44400 operations
 store 3 - doProfileTransaction - min latency 3 ms
 store 3 - doProfileTransaction - avg latency 30.701439 ms
 store 3 - doProfileTransaction - max latency 639 ms
 store 3 - doProfileTransaction - filtered latency 44.531307 ms

 store 4 - doProfileTransaction - 515 requests
 store 4 - doProfileTransaction - 51500 operations
```

```
  store 4 - doProfileTransaction - min latency 4 ms
  store 4 - doProfileTransaction - avg latency 60.155346 ms
  store 4 - doProfileTransaction - max latency 505 ms
  store 4 - doProfileTransaction - filtered latency 34.479477 ms
```

C Sans balance de la charge - 30s

```
Load Balancing Test go ...
  20 clients
Average transaction execution time:
    182 ms
  store 0 - doProfileTransaction - 1843 requests
  store 0 - doProfileTransaction - 183900 operations
  store 0 - doProfileTransaction - min latency 11 ms
  store 0 - doProfileTransaction - avg latency 220.94473 ms
  store 0 - doProfileTransaction - max latency 865 ms
  store 0 - doProfileTransaction - filtered latency 121.46709 ms

  store 1 - doProfileTransaction - 545 requests
  store 1 - doProfileTransaction - 54500 operations
  store 1 - doProfileTransaction - min latency 7 ms
  store 1 - doProfileTransaction - avg latency 93.36698 ms
  store 1 - doProfileTransaction - max latency 610 ms
  store 1 - doProfileTransaction - filtered latency 117.047066 ms

  store 2 - doProfileTransaction - 546 requests
  store 2 - doProfileTransaction - 54500 operations
  store 2 - doProfileTransaction - min latency 12 ms
  store 2 - doProfileTransaction - avg latency 115.490425 ms
  store 2 - doProfileTransaction - max latency 624 ms
  store 2 - doProfileTransaction - filtered latency 96.89213 ms

  store 3 - doProfileTransaction - 570 requests
  store 3 - doProfileTransaction - 57000 operations
  store 3 - doProfileTransaction - min latency 12 ms
  store 3 - doProfileTransaction - avg latency 111.19829 ms
  store 3 - doProfileTransaction - max latency 740 ms
  store 3 - doProfileTransaction - filtered latency 75.099556 ms

  store 4 - doProfileTransaction - 0 requests
  store 4 - doProfileTransaction - 0 operations
  store 4 - doProfileTransaction - min latency -1 ms
  store 4 - doProfileTransaction - avg latency -1.0 ms
  store 4 - doProfileTransaction - max latency -1 ms
  store 4 - doProfileTransaction - filtered latency 0.0 ms
```

D Avec balance de la charge - 30s

```
Load Balancing Test go ...
```

```
StoreSupervisor - Start
MEAN = 0.0 | STD_DEV = 0.0
StoreSupervisor - Sleep
StoreSupervisor - Start
MEAN = 73.50361053525704 | STD_DEV = 82.17954054868014
StoreSupervisor - Profil to move : P0
from store 1 to store 5                          <-----------------
StoreSupervisor - Profil moved
StoreSupervisor - Sleep
StoreSupervisor - Start
MEAN = 74.86311075264305 | STD_DEV = 83.69950147093013
StoreSupervisor - Profil to move : P10
from store 1 to store 4                          <-----------------
  20 clients
Average transaction execution time:
    162 ms
  store 0 - doProfileTransaction - 1237 requests
  store 0 - doProfileTransaction - 123600 operations
  store 0 - doProfileTransaction - min latency 6 ms
  store 0 - doProfileTransaction - avg latency 200.69193 ms
  store 0 - doProfileTransaction - max latency 1263 ms
  store 0 - doProfileTransaction - filtered latency 43.225647 ms

  store 1 - doProfileTransaction - 556 requests
  store 1 - doProfileTransaction - 55500 operations
  store 1 - doProfileTransaction - min latency 7 ms
  store 1 - doProfileTransaction - avg latency 113.31667 ms
  store 1 - doProfileTransaction - max latency 810 ms
  store 1 - doProfileTransaction - filtered latency 131.88275 ms

  store 2 - doProfileTransaction - 525 requests
  store 2 - doProfileTransaction - 52500 operations
  store 2 - doProfileTransaction - min latency 12 ms
  store 2 - doProfileTransaction - avg latency 139.81223 ms
  store 2 - doProfileTransaction - max latency 620 ms
  store 2 - doProfileTransaction - filtered latency 174.26118 ms

  store 3 - doProfileTransaction - 872 requests
  store 3 - doProfileTransaction - 87200 operations
  store 3 - doProfileTransaction - min latency 13 ms
  store 3 - doProfileTransaction - avg latency 148.06372 ms
  store 3 - doProfileTransaction - max latency 589 ms
  store 3 - doProfileTransaction - filtered latency 137.78885 ms

  store 4 - doProfileTransaction - 549 requests
  store 4 - doProfileTransaction - 56000 operations
  store 4 - doProfileTransaction - min latency 7 ms
  store 4 - doProfileTransaction - avg latency 137.44632 ms
  store 4 - doProfileTransaction - max latency 2776 ms
```

```
  store 4 - doProfileTransaction - filtered latency 92.65897 ms

Load Balancing Test ... Done
StoreSupervisor - Profil moved
StoreSupervisor - Sleep
```