LibRe: Protocole de gestion de la cohérence dans les systèmes de stockage distribués

Raja Chiky *, Sathiya Prabhu Kumar *,**
Sylvain lefebvre*, Eric Gressier-Soudan **

* 28, rue Notre Dame Des Champs
75006 Paris
{rchiky,sathiya-prabhu.kumar,sylvain.lefebvre}@isep.fr
** 292 Rue Saint-Martin
75003 Paris, France
autre-adresse@email
eric.gressier_soudan@cnam.fr

Résumé. Nous présentons dans ce papier un protocole de gestion de la cohérence appelé LibRe adapté aux systèmes de stockage orientés Cloud (telles que les bases de données NoSQL). Ce protocole garantit l'accès à la donnée la plus récente tout en ne consultant qu'une seule réplique. Cet algorithme est évalué par simulation et est également implémenté au sein du système de stockage Cassandra. Les résultats de ces expérimentations ont démontré l'efficacité de notre approche.

1 Introduction

Dans les systèmes de stockage de données distribués, il est d'usage de répliquer les données afin d'améliorer la performance et la disponibilité du système. La réplication des données est le processus de sauvegarde des mêmes données plusieurs fois, appelées répliques, dans plusieurs unités de stockage. Bien que ces données soient stockées à des emplacements physiques différents, pour une application donnée, elles jouent le même rôle et représentent la même chose. Le nombre de copies physiques et l'emplacement des copies peuvent varier en fonction des besoins de l'application. Toutefois, le fait de répliquer les données dans des emplacements physiques différents engendre un problème de cohérence auquel doit faire face le système. Lorsqu'une donnée d'une des répliques est modifiée, les autres répliques deviennent obsolètes tant que la mise à jour ne leur est pas parvenue. Par conséquent, il est important de mettre en place une stratégie de réplication qui permet de gérer en toute sécurité des opérations d'écriture/lecture de type CRUD (Create, Read, Update, Delete).

En effet, la stratégie de réplication affecte le comportement des systèmes distribués en ce qui concerne les propriétés de cohérence, disponibilité et tolérance au partitionnement. Nous savions d'après le théorème du CAP (Consistency, Availability, Partition Tolerance)(Gilbert et Lynch, 2002) qu'un système distribué ne peut respecter les trois propriétés à la fois. Ce théorème énonce que tout système distribué peut répondre à une seule contrainte parmi la cohérence (Consistency, i.e tous les noeuds du système voient exactement les mêmes données

au même moment) et la disponibilité (Availability, i.e les données sont toujours accessibles même en cas de panne). Par conséquent, les hypothèses traditionnelles, telles que la réplication complète ou le support des transactions, doivent être assouplies. Les solutions existantes diffèrent quant au degré de cohérence des données qu'elles fournissent, un compromis reste à déterminer entre la latence, la cohérence et la disponibilité des données (Abadi, 2012).

Dans ce cadre, nous proposons un protocole de gestion de la cohérence appelé LibRe (pour Library for Replication), permettant de fournir la dernière mise à jour d'une donnée à l'utilisateur tout en ne consultant qu'une seule réplique des données. Ceci se fait en stockant la liste des mises à jour estampillées dans un annuaire. Ce protocole a été évalué par simulation grâce à la plateforme Simizer (Lefebvre et al., 2014). Nous avons également évalué LibRe en l'implémentant au sein du système de stockage Cassandra (Lakshman et Malik, 2010) afin de le comparer avec les niveaux de cohérence offerts par ce système.

La suite de ce papier est organisée comme suit : la section 2 décrit l'état de l'art. Dans la section 3, nous présentons notre approche de gestion de la cohérence : LibRe. La section 4 détaille l'étude expérimentale de libRe par simulation. La section 5 décrit l'intégration de libRe dans Cassandra. Enfin, la section 6 conclut ce papier et donne un aperçu de nos perspectives de recherche.

2 Etat de l'art

Le problème de la cohérence des données se pose quand une mise à jour pour une des répliques se produit. Par conséquent, la cohérence des données est principalement influencée par la stratégie de réplication de données adoptée par le système. Daniel Abadi explique dans (Abadi, 2012) que les mises à jour se propagent de trois façons différentes selon la stratégie de réplication adoptée (une combinaison de ces modes de propagation peut être envisagée) : Synchrone, asynchrone ou hybride.

Synchrone Si les mises à jour sont appliquées de manière synchrone, la lecture des données de n'importe quelle réplique est correcte (cohérence forte). Cependant, le temps de propagation peut être affecté par la lenteur d'un noeud ou son emplacement physique dans le système. Il est d'usage de verrouiller l'accès à une donnée le temps que les mises à jour se propagent afin d'assurer une cohérence forte (Saito et Shapiro, 2005). Dans ce cas, la demande de lecture doit attendre ou doit être abandonnée jusqu'à ce que les données soient disponibles après mise à jour. Par conséquent, cela affecte la disponibilité du système.

Asynchrone Dans l'approche asynchrone, le noeud qui reçoit la première mise à jour l'applique localement et renvoie un message de succès au client. La mise à jour est ensuite propagée à ses répliques de manière asynchrone (en arrière-plan). Dans ce cas, il existe un compromis entre cohérence et latence en fonction de la façon dont les demandes de lecture sont traitées par le système.

— Cas 1 : Les requêtes de lecture sont servies à partir d'un noeud particulier en tant que point d'entrée central. Il n'y aura donc pas de sacrifice de la cohérence. Cependant, puisque les demandes sont servies à partir d'un noeud central, il existe un risque de surcharge de ce noeud ce qui peut affecter le temps de latence.

— Cas 2 : Si il n'y a pas de point d'entrée central pour les requêtes de lecture, il y a un risque de lecture d'une réplique où la mise à jour récente n'est pas encore appliquée. Dans ce cas, la latence pour la lecture serait minimisée mais en sacrifiant la cohérence des données.

Hybride La combinaison des deux modes synchrone et asynchrone dans l'objectif d'atteindre un meilleur compromis entre la latence, cohérence et disponibilité est également possible. Ce mode est connu sous le nom de cohérence au vote majoritaire (Quorum consistency). La majorité des répliques forme un quorum. La taille du quorum peut être estimée par la formule $N/2 + 1$, où N est le nombre de répliques. Le système va propager la mise à jour de façon synchrone au quorum et de manière asynchrone pour le reste des répliques. Dans ce cas, la cohérence du système peut être assurée par la formule $W + R > N$ où N est le nombre de répliques, W et R sont respectivement le nombre de répliques contactées pour l'écriture et la lecture.

Par ailleurs, il existe plusieurs niveaux de cohérence qui peuvent être classés de la façon suivante :

Cohérence forte Les systèmes de cohérence forte garantissent que toute lecture sur un élément puisse accéder à la dernière écriture faite sur cet élément. Les systèmes assurant une cohérence forte suivent les principes de sérialisabilité et linéarisabilité afin d'éviter la divergence des répliques (Herlihy et Wing, 1990).

Cohérence à terme Les systèmes qui respectent ce type de cohérence, garantissent qu'une mise à jour se propage vers toutes les répliques pour former un état cohérent si aucune autre mise à jour n'a été opérée entre temps. La lecture/écriture est considérée comme réussie si au moins une des répliques répond correctement à l'émetteur de la requête (valide la mise à jour). Les mises à jour sont ensuite propagées à travers le réseau et en arrière-plan de manière asynchrone. Dans ce type de cohérence, il n' y a aucun ordre imposé aux opérations, ainsi il peut y avoir des conflits de mises à jour. Ces conflits sont généralement résolus au niveau du client ou par des règles applicatives (Vogels, 2009).

Cohérence causale Ce type de cohérence permet de limiter les conflits qui peut subvenir dans le cas d'une cohérence à terme (comme vu précédemment). Il est également conçu pour des systèmes concurrents multi-lectures/écritures donnant la possibilité à toute réplique de répondre aux requêtes utilisateur. Toutefois, le système qui vérifie la cohérence causale impose qu'une mise à jour ne s'opère que si d'autres mises à jour ont été exécutées avant, ainsi les conflits peuvent être évités. En d'autres termes, la pré-condition définit un ordre partiel des opérations qui doivent être exécutées sur la base de leur causalité. Il peut y avoir tout de même des conflits si deux mises à jour sur la même donnée arrivent en même temps. La phase de résolution de conflit est toujours nécessaire. Même si la cohérence causale est plus "forte" que la cohérence à terme, il ne faut pas négliger sa consommation en bande passante et en nombre de messages transmis à travers le réseau.

Cohérence faible Ce type de cohérence est généralement utilisé dans des systèmes hors-ligne. Les répliques ne sont pas tout le temps connectées entre elles, elles peuvent l'être à de

courtes périodes qui ne sont a priori pas connues. Quand deux noeuds se connectent, elles partagent leurs mises à jour pour converger vers un état cohérent. Ainsi, si une donnée est modifiée, cette modification sera propagée aux répliques au bout d'un certain temps(Saito et Shapiro, 2005). A noter que d'autres définitions existent dans la littérature (Mosberger, 1993).

3 Notre approche : LibRe

A partir de l'étude des types de cohérence existants, nous avons décidé de proposer un nouveau protocole permettant de faire un compromis entre la cohérence, la disponibilité et la latence, nous l'avons appelé LibRe (Librairie pour la Réplication). Notre objectif étant d'atteindre une bonne cohérence (proche de la cohérence forte) tout en n'accédant qu'à UNE (ONE) seule réplique au lieu de TOUTES (ALL) les répliques ou à un QUORUM, et ceci peu importe le nombre de répliques dans le système. En effet, dans un système à cohérence à terme, nous savons qu'au moins une réplique est mise à jour et ceci à n'importe quel moment. Ainsi, en transmettant les demandes de lecture au niveau du noeud qui a reçu la version la plus récente de la donnée désirée et en évitant les noeuds erronés (ceux qui n'ont pas encore reçu les modifications), nous pourrions préserver la cohérence du système.

LibRe a été proposé afin de garantir :

— l'accès à la version la plus récente des données ;
— la disponibilité du système ;
— un minimum de latence.

La principale composante de LibRe est un annuaire qui répertorie toutes les écritures et mises à jour opérées dans le système. Ainsi, un noeud peut être cohérent pour quelques données mais erroné pour d'autres. A titre d'exemple, supposons une topologie où les serveurs sont connectés à un réseau commun. Si un noeud particulier dans la topologie est en panne ou séparé du réseau pendant une période de temps, le noeud sera incohérent pour les opérations qui ont eu lieu pendant cette période. Toutefois, le noeud sera cohérent pour les données non affectées par ces opérations. En d'autres termes, nous souhaitons identifier les mises à jour qui ne se sont pas encore exécutées sur les noeuds. Ceci permet au système d'arrêter de transmettre les demandes au noeud erroné jusqu'à ce qu'il soit à nouveau cohérent. Un noeud est considéré comme erroné si il contient des données périmées pour la requête entrante. Cette restriction peut être libérée une fois que les mises à jour sont opérées sur le noeud.

Pour cela, les noeuds annoncent leurs mises à jour à l'annuaire. L'annonce contient l'identifiant de la ressource modifiée ainsi que l'identifiant du noeud qui a reçu la mise à jour. L'annuaire a été implémenté de façon décentralisée selon une table de hachage distribuée (DHT) (Zhang et al., 2013) afin d'éviter un unique point d'entrée SPOF (Single Point of Failure) si celui-ci tombe en panne. A noter également que la fiabilité de LibRe, son coût d'exécution ainsi qu'une étude formelle de son comportement ont été effectués. Cette étude ne sera pas présentée dans ce papier, mais pour plus d'information, le lecteur peut se référer à (Kumar et al., 2015).

3.1 Description de LibRe

La figure 1 montre où se positionne LibRe dans le système complet.

Nous appelons le "Frontend" le noeud par lequel un client se connecte pour envoyer ses requêtes. Nous considérons une architecture multi-lecture/écriture où chaque noeud peut jouer le rôle du frontend. Le frontend interroge le registre LibRe afin de trouver le noeud cible pouvant répondre à la requête.

Tel que le montre la figure 1, le protocole LibRe est constitué de 3 composants à savoir : l'annuaire également appelé registre (*Registry*), le gestionnaire de disponibilité (*Availability Manager*), le gestionnaire de notification (*Advertisement Manager*). L'annuaire est une structure de stockage en mémoire sous format clé-valeur : la clé étant l'identifiant d'une ressource (donnée) et la valeur est une liste des répliques contenant les données les plus récentes. Si la taille de la liste des noeuds mis à jour atteint le nombre de répliques dans le système, cela signifie que cette donnée est cohérente. L'enregistrement clé-valeur pour cette donnée peut donc être supprimé de l'annuaire en toute sécurité. Le gestionnaire de disponibilité est en charge de transmettre la requête de lecture vers le noeud contenant la version récente des données. Le gestionnaire de notification quant à lui se charge d'enregistrer dans l'annuaire les notifications des noeuds ayant reçu des mises à jour pendant les opérations d'écritures. Ces composants sont stockés dans les noeuds répliques suivant une architecture de DHT (Zhang et al., 2013).

A noter que tous les noeuds contiennent un annuaire. Toutefois, celui-ci n'est pas répliqué, il est colocalisé avec les données dont il est en charge (supervision de leurs versions). En cas de défaillance d'un noeud, son annuaire est reconstruit dynamiquement avec le flot des requêtes transmis aux noeuds responsables de la réplication. LibRe est destiné à être adopté dans des système éventuellement cohérents. Par conséquent la mise à jour de l'annuaire peut ne pas être exactement la même que celle des noeuds de réplication sans nuire de manière importante à l'utilisateur.

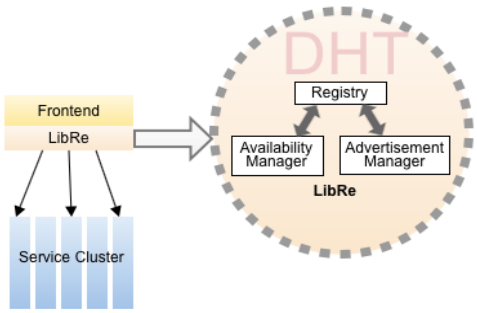
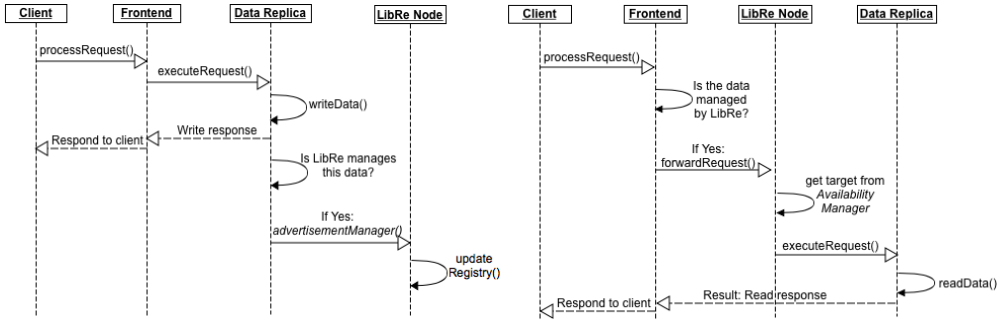


FIG. 1: Architecture globale de LibRe

Les figures 2a et 2b montrent le diagramme de séquence du comportement de LibRe pendant les opérations d'écriture et de lecture respectivement.

3.1.1 Opération d'écriture

Dans tout système distribué, quand un frontend reçoit une requête d'écriture, il la transfère à toutes les répliques. Si tous ces derniers lui accusent la bonne réception de la modification, le frontend émet une réponse de succès au client. Si le nombre suffisant de réponses des répliques n'a pas été reçu dans les délais, le frontend émet une réponse d'échec. Le protocole LibRe suit



(a) LibRe Write/Update Operation (b) LibRe Read Operation

FIG. 2: Diagramme de séquence LibRe

ce même comportement usuel, en l'étendant avec un message de notification que la réplique envoie de manière asynchrone à l'annuaire. Cette notification est constituée de la clé de la donnée, sa version *version-id* et le noeud ayant reçu la mise à jour.

L'algorithme 1 décrit la contribution du gestionnaire de notifications lors d'une opération d'écriture. On distingue deux cas :

— Insertion : Lorsqu'une donnée est écrite dans le système de stockage pour la première fois ;

— Mise à jour : Quand la donnée à mettre à jour existe déjà dans le système de stockage. La mise à jour peut provenir du frontend ou transférée par un noeud réplique.

Algorithm 1 Opération d'écriture avec LibRe

```
1:  function LOG(dataKey, versionId, nodeIP)
2:      if Reg.exists(dataKey) then
3:          RegVersionId ← Reg.getVersionId(dataKey)
4:          if versionId = RegVersionId then
5:              replicas ← Reg.getReplicas (dataKey)
6:              replicas ← appendEP(replicas, nodeIP)
7:              Reg.updateReplicas(dataKey, replicas)
8:          else if versionId > RegVersionId then
9:              replicas ← reinitialize(nodeIP)
10:             Reg.updateReplicas(dataKey, replicas)
11:             Reg.updateVersionId(versionId)
12:         end if
13:     else
14:         replicas ← nodeIP
15:         Reg.createEntry(dataKey, replicas)
16:         Reg.updateVersionId(versionId)
17:     end if
18: end function
```

Quand un noeud réplique envoie un message de notification concernant une mise à jour, le *Gestionnaire de disponibilité* suit les actions suivantes. D'abord il vérifie si la clé de la donnée *data-key* existe déjà dans l'annuaire : la ligne 2. Si oui (opération de mise à jour), ligne 3 : la *version-id* pour cette ressource est récupérée. *Version-id* est un nombre (qui croit de façon monotone) représentant le caractère récent de la mise à jour, cela peut être par exemple le timestamp de l'opération.

Ligne 4 : Si la *version-id* enregistrée dans l'annuaire correspond à la version-id de l'opération (la mise à jour), alors la ligne 7 : l'adresse IP du noeud sera ajoutée avec la liste des répliquas existant. Ligne 8 : Si les deux versions-ID ne correspondent pas et si la version-id de l'opération est supérieure à la *version-id* existant dans l'annuaire (ce qui signifie, la mise à jour est nouvelle), la ligne 9 -10 : la liste des répliques pour la *data-key* sera réinitialisée à l'adresse IP du noeud. La *Version-id* sera également mise à jour (Ligne 11).

Si *data-key* n'existe pas dans l'annuaire (ligne 13), ce qui signifie que c'est une insertion, alors une nouvelle entrée sera créée avec la nouvelle *data-key*, l'adresse IP du noeud et la *version-id* de l'opération (lignes 14 à 16). Cette approche correspond à la politique LWW (Last Writer Wins : dernière mise à jour gagnante) (Saito et Shapiro, 2005).

3.1.2 Opération de lecture

Quand le frontend reçoit une requête de lecture, il envoie un message de demande de disponibilité (avec la requête et la *data-key*) au noeud hébergeant l'annuaire. Celui-ci envoie la requête aux répliques se trouvant dans l'annuaire. Si aucune entrée n'est trouvée, alors la requête sera envoyée à n'importe quel noeud réplique (cf. figure 2b). L'algorithme 2 décrit ce comportement.

Algorithm 2 Opération de lecture avec LibRe

```
1: function GETTARGETNODE(dataKey)
2:     replicas ← Reg.getReplicas(dataKey)
3:     targetNode ← getTargetNode(replicas)
4:     if targetNode is NULL then
5:         targetNode ← useDefaultMethod(dataKey)
6:     end if
7:     forwardRequestTo(targetNode)
8: end function
```

4 Evaluation de LibRe avec Simizer

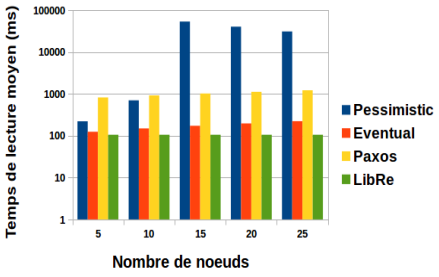
Nous avons tout d'abord évalué la politique de LibRe par simulation en étendant la plateforme Simizer [1] (Lefebvre et al., 2014). Nous avons ainsi pu comparer Libre avec des protocoles de synchronisation de données connues. Les outils et librairies existantes tels que OptorSim (Bell et al., 2003), CloudSim (Calheiros et al., 2011) et SimGrid (Casanova et al., 2008) ne fournissent pas de moyens simples pour simuler des problèmes de cohérence des données dans les systèmes à large échelle.

Nous avons testé le protocole Libre en le comparant à d'autres politiques connues : cohérence synchrone (Saito et Shapiro, 2005), cohérence à terme (cohérence asynchrone) (Saito et Shapiro, 2005) et Paxos (Chandra et al., 2007). Pour cela nous avons considéré 1200 requêtes, dont 400 sont des opérations d'écriture et 600 de lecture, ces requêtes ont été mélangées dans un ordre aléatoire. Pendant les opérations d'écriture, les données sont écrites soit en cache mémoire ou en disque. Pendant les opérations de lecture, le temps de latence d'une requête est calculé en fonction du nombre d'instructions nécessaires pour la traiter additionné au temps d'accès aux données. Si les données nécessaires sont disponibles dans le cache du noeud le

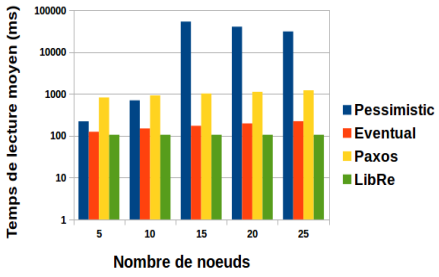
1. https ://github.com/isep-rdi/simizer

temps d'accès est négligé (considéré à zéro) sinon le temps d'accès aux ressources sera calculé en fonction de la taille des données. Si certaines des données nécessaires n'existent pas, la requête sera considérée comme échouée. Nous avons utilisé les mêmes requêtes pour l'ensemble des politiques de cohérence testées en faisant varier le nombre de noeuds : 5, 10, 15, 20, et 25.

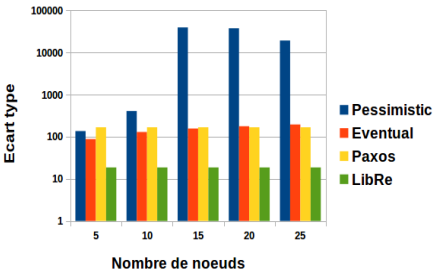
La figure 3 montre les résultats obtenus. Nous pouvons observer dans la figure 3d que seule la cohénrece à terme présente des lectures erronées, dont le nombre avec le nombre de noeuds. Mais quand nous examinons la latence, nous pouvons voir qu'elle est élevée pour Paxos et la cohérence pessimiste, ce qui peut affecter la disponibilité du système. Nous pouvons également voir que la latence dans LibRe que ce soit la moyenne (figure 3b) ou l'écart-type (figure 3c) est moins élevée que les autres politiques et n'est pas affectée par l'augmentation du nombre de noeuds.



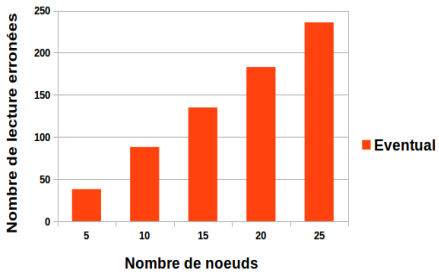
(a) Latence moyenne en lecture



(b) Latence moyenne en écriture



(c) Ecart type



(d) Nombre de lectures erronées

FIG. 3: Résultats expérimentaux

5 CaLibRe : Evaluation de LibRe avec Cassandra

5.1 Dispositif

Cassandra (Lakshman et Malik, 2010) est l'un des systèmes open-source NoSQL les plus populaires qui suit une architecture de type DHT. Par conséquent, nous avons décidé de mettre en oeuvre le protocole LibRe dans le workflow de Cassandra et évaluer sa performance contre

les niveaux de cohérence offerts par ce système : ONE, QUORUM et ALL. La version utilisée de Cassandra est la version 2.0.0.

Les expérimentations ont été menées avec un cluster de 19 instances CaLibre et Cassandra constitué de 4 medium, 4 small et 11 micro instances du service Cloud Amazon EC2 (Amazon, 2012). Une grande (large) instance a été également utilisée pour envoyer les requêtes du banc d'essai (benchmark) YCSB (Cooper et al., 2010). YCSB : Yahoo Cloud Services Benchmark est l'un des projets open source écrit en Java pour l'évaluation des systèmes de stockage dans le Cloud. L'objectif principal de YCSB est de mesurer la performance du système à travers la latence des requêtes, le passage à l'échelle, etc.

Le système d'exploitation utilisé pour toutes les instances est Ubuntu Server 14.04 LTS - 64 bit. La configuration du banc d'essai utilisé est "Update-Heavy" workload (*workload-a*), avec un nombre d'enregistrements de 100 000, nombre d'opérations de 100 000, nombre de threads de 10 et un facteur de réplication de 3.

Par défaut YCSB stocke 10 colonnes par clé (Row-key). Le row-key est utilisé comme data-key dans l'annuaire de LibRe. L'utilisation du row-key comme clé de la ressource peut conduire au problème suivant : Si une ou un sous-ensemble de colonnes est mis à jour dans une réplique r, l'annuaire pourrait assumer que r contient la version récente pour toutes les colonnes de ce row-key. Afin d'éviter cette situation, nous mettons à jour toutes les 10 colonnes lors de chaque mise à jour.

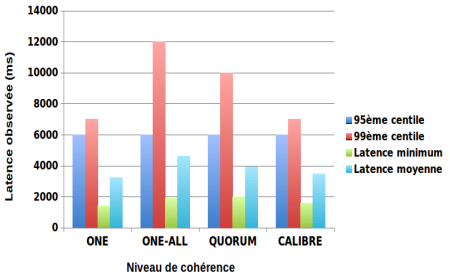
Nous évaluons la performance et la cohérence des 19 instances Cassandra avec différentes options de cohérence (ONE, QUORUM et ALL) et la comparons aux 19 instances CaLibRe avec une cohérence ONE (Lecture et écriture à partir d'une seule réplique). La performance est mesurée en terme de latence (pour l'écriture et la lecture) et la cohérence en terme de nombre de lectures erronées.

Afin de simuler un nombre significatif de lectures erronées, un mécanisme de propagation de mise à jour partielle a été injecté dans le cluster Cassandra et CaLibRe (Kumar et al., 2015). Dans cette extension de YCSB, nous ne propageons les mises à jour qu'à deux instances au lieu des 3 répliques.

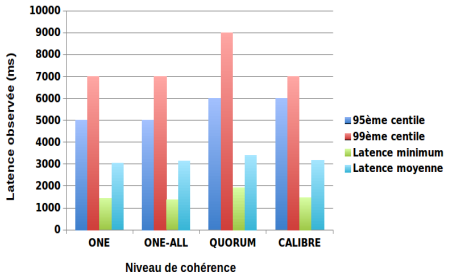
5.2 Résultats

Les figures 4a, 4b et 4c montrent respectivement les résultats en terme de latence de lecture, latence d'écriture et nombre de lectures erronées en utilisant Cassandra avec les différents niveaux de cohérence contre CaLibRe (le protocole LibRe implémenté dans Cassandra). Dans la figure 4a, ONE correspond à la cohérence niveau ONE (lecture et écriture), QUORUM correspond au niveau de cohérence QUORUM, ONE-ALL correspond à une cohérence ONE en écriture et ALL en lecture, CALIBRE correspond à notre implémentation de LibRe dans Cassandra. Nous n'avons pas testé le cas d'une cohérence ONE pour la lecture et ALL pour l'écriture à cause de notre extension de YCSB pour injecter des mises à jour erronées.

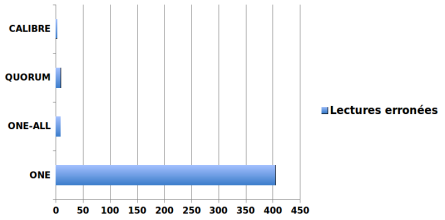
La figure 4a correspondant à la mesure de la latence en lecture montre que le 95e centile de latence de Calibre est similaire aux autres options de cohérence de Cassandra. Le 99e centile de latence CALIBRE et ONE de Cassandra sont au même niveau et sont meilleurs que ONE-ALL et QUORUM. Les latences minimales et moyennes de CALIBRE sont légèrement plus élevées que le niveau ONE de Cassandra mais restent meilleures que les options de co-



(a) Latence en lecture



(b) Latence en écriture



(c) Lectures erronées

FIG. 4: Evaluation de CaLibRe

hérence QUORUM et ONE-ALL. Cela est dû au fait que le protocole LibRe impose un appel additionnel au registre pour les traitements des requêtes.

La figure 4b correspondant à la mesure de la latence en écriture montre que le 95e centile de la latence en écriture de Calibre est similaire à celui du QUORUM, et que Calibre est plus rapide dans d'autres indicateurs : 99e centile, latences minimum et latences moyennes du QUORUM. Cependant, en comparant les entités ONE et ONE-ALL, certaines des mesures de CALIBRE sont légèrement plus élevées (mais pas significativement). Cela est dû au fait que les deux cohérences ONE et ONE-ALL n'effectuent l'écriture que dans une seule réplique. Dans CALIBRE, le système écrit aussi dans une seule réplique, mais cela nécessite une communication avec l'annuaire ce qui implique un coût supplémentaire.

La figure 4c montre le nombre de lectures erronées pour chaque niveau de cohérence. Cassandra avec le niveau de cohérence ONE montre le plus grand nombre de lectures erronées. Il y avait également quelques lectures erronées dans les autres options de cohérence, mais ces chiffres sont négligeables par rapport au nombre total de requêtes. De ces résultats, il est possible de conclure que CaLibRe offre un niveau de cohérence similaire à celui fourni par le niveau ONE-ALL et QUORUM et avec une meilleure latence.

6 Conclusion

Nous avons présenté dans cet article des méthodes de gestion de la cohérence dans les systèmes de stockage de données à large échelle. L'objectif de cette étude et de trouver le meilleur compromis entre la cohérence, la latence et la disponibilité dans ces systèmes. Après étude

des différents types de cohérence existants et des algorithmes implémentés dans les systèmes actuels, nous nous sommes rendus compte qu'il existait une possibilité non encore explorée : assurer la cohérence la plus forte possible tout en ne consultant qu'une seule réplique. Pour cela, nous avons proposé un algorithme appelé LibRe que nous avons évalué par simulation grâce à la plateforme Simizer mais aussi dans un cas réel en l'intégrant au système de stockage Cassandra. Les résultats de ces expérimentations ont démontré l'efficacité de notre approche.

Cependant, les performances de LibRe n'ont pas été testées dans le cas où des noeuds rejoignent ou quittent le cluster. Au cours de ces événements, le protocole LibRe peut subir des incohérences temporaires. Ceci fait partie de notre étude en cours. Une autre perspective de ce travail est d'étudier l'influence du choix de la version des répliques. En effet, nous avions choisi le timestamp comme critère pour décider de la version la plus récente, mais cela pourrait être aussi une horloge vectorielle par exemple. Enfin, nous envisageons d'expérimenter Libre dans un cas d'application réel.

Références

Abadi, D. J. (2012). Consistency tradeoffs in modern distributed database system design : Cap is only part of the story. *Computer 45*(2), 37–42.

Amazon (2012). Amazon elastic compute cloud (amazon ec2). http ://aws.amazon.com/ec2/.

Bell, W. H., D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, et F. Zini (2003). Optorsim - a grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing Applications*, 2003.

Calheiros, R. N., R. Ranjan, A. Beloglazov, C. D. Rose, et R. Buyya (2011). Cloudsim : A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software : Practice and Experience (SPE)*.

Casanova, H., A. Legrand, et M. Quinson (2008). Simgrid : a generic framework for large-scale distributed experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, Washington, DC, USA, pp. 126–131. IEEE Computer Society.

Chandra, T. D., R. Griesemer, et J. Redstone (2007). Paxos made live : an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, New York, NY, USA, pp. 398–407. ACM.

Cooper, B. F., A. Silberstein, E. Tam, R. Ramakrishnan, et R. Sears (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, New York, NY, USA, pp. 143–154. ACM.

Gilbert, S. et N. Lynch (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*(2), 51–59.

Herlihy, M. P. et J. M. Wing (1990). Linearizability : A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*(3), 463–492.

Kumar, S. P., R. Chiky, S. Lefebvre, et E. G. Soudan (2015). Calibre : A better consistency-latency tradeoff for quorum based replication system. In *8th International Conference on Data Management in Cloud, Grid and P2P Systems.*, Globe. Springer LNCS.

Lakshman, A. et P. Malik (2010). Cassandra : a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*(2), 35–40.

Lefebvre, S., S. P. Kumar, et R. Chiky (2014). Simizer : Evaluating consistency trade offs through simulation. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, New York, NY, USA. ACM.

Mosberger, D. (1993). Memory consistency models. *SIGOPS Oper. Syst. Rev. 27*(1), 18–26.

Saito, Y. et M. Shapiro (2005). Optimistic replication. *ACM Comput. Surv. 37*(1), 42–81.

Vogels, W. (2009). Eventually consistent. *Commun. ACM 52*(1), 40–44.

Zhang, H., Y. Wen, H. Xie, et N. Yu (2013). *Distributed Hash Table - Theory, Platforms and Applications.* Springer Briefs in Computer Science. Springer.

Summary

In this paper, we present a consistency management protocol called LibRe (for Library Replication) adapted to Cloud storage systems (such as NoSQL databases). This protocol ensures access to the latest data version, while only one replica is consulted. This algorithm is evaluated by simulation and is also implemented within the Cassandra storage system. The results of these experiments have demonstrated the effectiveness of our approach.