Construction universelle d'objets partagés au-dessus d'un entrepôt clé-valeur

Pierre Sutra

ComPAS'2014 : Parallélisme / Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Construction universelle d'objets partagés au-dessus d'un entrepôt clé-valeur

Pierre Sutra

Université de Neuchâtel, Suisse

Résumé
Un entrepôt clé-valeur (ECV) est un système de stockage réparti accessible via une interface de type *put*/*get*. La simplicité de cette interface permet une élasticité horizontale et une grande robustesse au système. Ces deux atouts font des ECV une des pierres angulaires de l'informatique en nuage. Néanmoins, il est reconnu difficile de mettre en oeuvre des applications au-dessus d'un tel interface, et en particulier de porter des applications existantes. Afin de résoudre ce problème, nous proposons dans cet article une méthode de construction universelle d'objets partagés au-dessus d'un ECV. Notre méthode est non-intrusive, et permet la mise en œuvre d'objets héritant des propriétés du ECV sous-jacent, à savoir sa cohérence, son élasticité et sa durabilité. Afin de démontrer l'intérêt de notre approche, nous présentons et évaluons un prototype au-dessus d'Infinispan, un système de stockage de données libre utilisé aujourd'hui par un grand nombre d'acteurs informatiques.

---

1. Introduction

L'informatique en nuage est un paradigme récent pour assurer le fonctionnement des systèmes informatiques modernes massivement répartis. Ce paradigme est supporté par des centres de traitement de données dédiés, composés de machines virtuelles mises en réseaux. Les usagers de l'informatique en nuage ont accès à de nouveaux types de services à la demande tels que des infrastructures (IaaS), des plateformes de développement (PaaS), des logiciels (SaaS), voir des données (DaaS). De tels services permettent aux entreprises de virtualiser leur capacité de calcul de manière à satisfaire au mieux leurs besoins, accélérerant ainsi l'arrivée sur le marché de leurs produits.
Du fait d'une utilisation intensive et continue, les systèmes de stockage support de l'informatique en nuage doivent répondre à de nouveaux défis. En particulier, ils se doivent d'être robustes et élastiques, c'est-à-dire de permettre une augmentation de leur capacité de calcul et de stockage à la demande. Un entrepôt clé-valeur (ECV) est un système de stockage plat réparti entre plusieurs nœuds. Un tel système répond particulièrement bien à de tels besoins [7], et est couramment employé par un grand nombre de prestataires de l'informatique en nuage.
Un ECV offre un accès à son stockage réparti via une interface de type *put*/*get*. Ce modèle simple est toutefois limitant pour la construction d'un certain nombre d'applications réparties. En particulier, il est difficile de porter une application existante sur un ECV, et ceci en comparaison des modèles établis, tels que les bases de données relationnelles ou la programmation par objets. Par ailleurs, il est classique de construire une application par composition de briques de base. Cette approche a de nombreux bénéfices, notamment de promouvoir une architecture modulaire,

aux frontières bien définies, facilement extensible et dont le développement peut être collaboratif. Une interface simplifiée ne permet d'appliquer une telle démarche.
Dans cet article, nous proposons une nouvelle approche permettant la construction d'objets partagés au-dessus d'un ECV. Notre approche est universelle dans le sens où elle permet le partage d'un objet séquentiel quelconque entre plusieurs processus clients. Les objets ainsi construits héritent des propriétés de l'ECV sous-jacent, à savoir élasticité, tolérance aux fautes, cohérence et durabilité. En offrant une interface de programmation bien établie aux développeurs, notre solution permet la création d'applications au-dessus d'un ECV reposant sur des techniques de développement classiques. Elle facilite aussi le portage de l'existant sur les infrastructures de l'informatique en nuage. Afin de valider ces bénéfices, nous présentons dans cet article une série de résultats expérimentaux mettant en jeu notre approche.
La suite de cet article est organisée ainsi : Dans la Section 2, nous décrivons notre construction en détail. La Section 3 introduit la mise en œuvre que nous avons effectué de ce mécanisme au-dessus d'Infinispan. Nous présentons des résultats expérimentaux dans la Section 4. La Section 5 fournit un aperçu de l'état de l'art. Nous concluons cet article dans la Section 6.

2. Une Construction Universelle

Dans cette section, nous décrivons dans un premier temps les principes d'un entrepôt clé-valeur, Puis, nous expliquons le fonctionnement algorithmique de notre construction universelle, ainsi que plusieurs optimisations assurant sa rapidité en pratique. Ces optimisations seront justifiées dans la partie évaluation.

2.1. Entrepôt clé-valeur

Un entrepôt clé-valeur (ECV) est un système de stockage plat (dit aussi en ratelier) réparti entre plusieurs nœuds. Le client d'un ECV émet des requêtes pour stocker et récupérer les données via l'intermédiaire d'une API simple qui sera décrite sous peu.
La topologie et le routage entre les nœuds d'un ECV varient en fonction du système considéré. Ainsi, dans Dynamo [14] les nœuds sont organisés en anneau. Chaque nœud possède une réplique de l'anneau, et cette réplique est maintenue à jour en faisant circuler autour de l'anneau des informations relatives aux fautes. Du fait de cette réplication totale de la topologie du système, le routage dans Dynamo s'effectue en un saut. Á l'inverse, des systèmes tels que Pastry [27] effectue un routage en $\log(n)$ sauts, où n est le nombre de nœuds du système, mais nécessite le stockage d'une information partielle sur chaque nœud (en $O(1)$).
Un ECV utilise un indexation implicite des données, c'est à dire que chaque nœud est capable de retrouver localement l'emplacement d'une données sans avoir à consulter un annuaire extérieur. En général, les données stockées dans un ECV sont indexées par l'intermédiaire d'une fonction de hachage qui associe à chaque donnée un ou plusieurs nœuds en fonction du degrée de redondance. La technique la plus communément employée est celle du hachage cohérent [20].
Le client d'un ECV stocke une *valeur* v par l'intermédiaire d'une *clé* de stockage k en effectuant l'appel $put(k, v)$. La fonction $get(k)$ permet de retrouver la valeur stockée à la clé k. Nous considérons par ailleurs qu'un client peut aussi enregistrer un listener sur le ECV par l'intermédiaire de la opération $registerListener(k, F)$ qui prend en paramètres une clé k et une fonction de rappel F. La fonction de rappel F est invoquée avec les arguments k et v lorsque la valeur stockée à la clé k est modifiée à la valeur v.
La cohérence des appels à l'interface d'un ECV est variable. Par exemple, Cassandra assure l'atomicité [18] des opérations en s'appuyant sur le fait que l'ensemble des nœuds de stockage est synchronisé. De son côté, Infinispan [3] s'appuie sur l'algorithme de diffusion atomique mis

Algorithm 1 Construction Universelle – code du client c

```
1:  /* Variable Partagée */
2:      D                                                  // Un entrepôt clé-valeur
3:
4:  /* Variables Locales */
5:      s ∈ États                                          // Initialement, ⊥
6:      r ∈ Valeurs                                        // Initialement, ⊥
7:      Q                                                  // Une file d'attente ; initialement ⊥
8:
9:  When open()
10:     D.registerListener(callBack)
11:     (x, _, f) ← get(k)
12:     if (x, _, f) = ⊥ then
13:         s ← s_0
14:     else if f = PER then
15:         s ← x
16:     else
17:         D.put(k, (⊥, c, RET))
18:         wait until s ≠ ⊥
19:
20: When close()
21:     D.put(k, (s, c, PER))
22:     D.unregisterListener(callBack)
23:     s ← ⊥, r ← ⊥, Q ← ⊥
24:
25: When invoke(op)
26:     r ← ⊥
27:     D.put(k, (op, c, INV))
28:     wait until r ≠ ⊥
29:     return r
30:
31: When callBack(k, (x, c', f))
32:     if f = INV then
33:         if s ≠ ⊥ then
34:             (s, v) ← τ(s, x)
35:             if c = c' then
36:                 r ← v
37:         else if Q ≠ ⊥ then
38:             Q ← Q ∘ ⟨x⟩
39:     else if f = RET then
40:         if s ≠ ⊥ then
41:             D.put(k, (s, c, PER))
42:         else if c = c' then
43:             Q ← ⟨⟩
44:     else if f = PER ∧ s = ⊥ then
45:         s ← x
46:         for x ∈ Q do                                   // Dans l'ordre défini par Q.
47:             (s, v) ← τ(s, x)
48:
```

en œuvre dans l'intergiciel JGroups [11]. Les appels effectués au système Dynamo [14] sont éventuellement cohérents : à terme tous les nœuds détiennent la même valeur pour une clé k, mais l'ordre dans lesquels ces mises-à-jour ont été appliquées varie d'un nœud à l'autre.
Dans la section qui suit, nous allons expliquer comment à partir de l'API d'un ECV décrite ci-dessus nous pouvons construire des objets partagés.

2.2. Algorithme

Notre algorithme est une variante de la réplication de machine à états (SMR) [28]. L'approche SMR est *universelle*, c'est-à-dire qu'elle décrit le partage d'un objet séquentiel quelconque entre des processus. La sémantique de partage est la linéarisabilité, ou *atomicité*, c'est à dire que l'objet partagé se comporte comme s'il était invoqué localement [18]. Dans ce qui suit, nous présentons dans un premier temps une définition formelle d'un objet. Puis, nous décrivons notre algorithme de construction universelle au-dessus d'un ECV, ainsi que des optimisations en vue d'améliorer ses performances en pratique.
Un objet est une instance d'un type de donnée. Nous formalisons un type de donnée par un automate déterministe dont la spécification est donnée par un ensemble d'états *États*, un état initial $s_0 \in$ *États*, un ensemble d'opérations *Ops*, un ensemble de valeurs de retour *Valeurs*, et une fonction de transition τ : *États* $\times$ *Ops* $\rightarrow$ *États* $\times$ *Valeurs*. Par la suite, et sans perte de généralité, nous supposons que toute opération *op* est *totale*, c'est à dire que *États* $\times \{op\}$ est inclus dans le domaine de τ.
Notre construction universelle est décrite par l'Algorithme 1. Le cœur de notre algorithme est hérité de l'approche SMR, et fonctionne comme suit : Lorsqu'un client c invoque une opération *op* sur un objet partagé o, *op* est transmis sur le ECV par l'intermédiaire de la opération $put(k, op)$, où k est la clé unique identifiant o. Quand la fonction de rappel $callBack(k, (op, c'))$ est invoquée localement par le ECV, l'opération *op* est exécutée sur une copie locale de l'état logique de o. Par ailleurs, si *op* avait été enregistrée comme un appel du client, i.e., $c' = c$, le client est notifié de la valeur de retour.
De manière plus détaillée, notre algorithme assure non seulement la cohérence des accès aux objets, mais aussi leur pérénnité sur le ECV. Pour ce faire, il emploie quatre variables : D est une variable partagée représentant le ECV, s est l'état logique de l'objet partagé sur c, r est une variable locale stockant la valeur de retour de la dernière opération invoquée par c, et Q représente une file d'attente locale. Initialement, le client assigne la valeur nulle ($\perp$) aux variables locales.
Dans un contexte général, par exemple pour une application concurrente déjà existante, les clients créent et détruisent des objets partagés à des instants différents. Afin d'assurer la cohérence des accès lors de la création et la destruction d'un objet, notre algorithme emploie les opérations *open* et *close*. De plus pour ce faire, la variable D stocke des triplets de la forme (x, c, f) où (i) x est soit une opération, soit un état de l'objet, (ii) c identifie le client ayant exécuté cette insertion dans le ECV, et (iii) f est un drapeau indiquant le type d'insertion. Une insertion de type `INV` dénote l'invocation d'une opération sur l'objet, et x est alors une opération. Dans le cas où f égale `RET`, le client c souhaite retrouver l'état persistent de l'objet. Cet état est véhiculé par un client par une insertion de la forme $(s, c,$ `PER`$)$ dans le ECV.
Lorsqu'un client c initialise un objet o via l'opération *open*(), il enregistre d'abord la fonction de rappel *callBack*, puis retrouve dans le ECV le triplet contenu à la clé k associée à o. Trois cas sont à considérer :

1. Le ECV ne contient aucune valeur pour la clé k. Dans un tel cas, s est assigné à s_0, la valeur initiale de l'objet (ligne 13).

2. Si maintenant le triplet retrouvé dans le ECV est de la forme $(x, \_, \texttt{PER})$, alors x est un état de l'objet, et c peut assigner x à la variable s (ligne 15). Notons ici que s est l'état de l'objet suite à l'ensemble des opérations antérieures à l'ouverture de o par c. L'enregistrement de *callBack* auprès du ECV avant l'exécution de *get* assure que s sera par la suite mis-à-jour pour les opérations postérieures à cette ouverture.

3. Enfin, dans le cas où le triplet stocké à la clé k ne contient pas l'état de l'objet, le client attend qu'un autre client lui transmette cette information. Pour ce faire, il initialise d'abord la file d'attente Q à une valeur vide (ligne 43). Cette variable contiendra tous les appels linéarisés après la réception de sa demande d'ouverture (ligne 38). Une fois l'état de l'objet reçu, les appels sont dépilés et appliqués à la variable s (lignes 44 à 47). Si aucun client n'est disponible, l'ouverture échoue après un certain temps et le client est notifié par une exception (non décrite dans l'Algorithme 1).

Quand un client c souhaite clôturer son accès à un objet partagé, il exécute un appel à *close*(). Cette opération insère un triplet $(s, c, \texttt{PER})$, puis désenregistre la fonction de rappel de l'ECV. Les variables locales peuvent alors être effacées (lignes 21 à 23).
Les objets partagés mis en œuvre par notre approche héritent des propriétés de l'ECV sous-jacent ; à savoir :

1. Les objets sont élastiques verticalement (plusieurs clients sur une même machine physique) et horizontalement (plusieurs clients sur plusieurs machines).

2. La cohérence des objets partagés via notre approche dépend du ECV sur lequel repose sa mise en œuvre. En particulier, si les opérations *put* et *get* sont atomiques, tout objet partagé est lui aussi atomique. Notons que dans un tel cas, nous pouvons améliorer les performances en considérant des objets séquentiellement cohérents [22]. Pour ce faire, nous procédons comme suit : Les opérations d'un objet sont annotées afin d'indiquer si elles modifient, ou non, l'état de l'objet. Lors d'un appel à une opération *op*, si celle-ci est en lecture seule, *op* est exécutée localement et son résultat est immédiatement retourné. De manière moins intrusive, il est aussi possible de cloner l'état local de l'objet, exécuter l'opération de manière optimiste et comparer l'état final avec l'état stable, afin de savoir s'il a été modifié. Nous utilisons ces deux optimisations dans la mise en œuvre de notre approche, décrite sous peu.

3. Le ECV assure aussi la durabilité des objets lorsque les clients clôturent proprement leurs accès. En effet, lorsqu'un client effectue un appel à *close*(), l'état courant de l'objet est enregistré dans le ECV. Notons cependant qu'une défaillance du dernier client accédant à un objet peut mettre en péril cette durabilité, et précisément si celui-ci effectue une panne franche avant un appel à *close*(). Il est possible de se prémunir d'un tel évènement, en synchronisant l'ouverture et la fermeture d'un objet par $f + 1$ clients, où f est le nombre maximal de pannes franches pouvant intervenir conjointement au cours d'une exécution. Cette synchronisation peut avoir lieu au sein de l'objet lui-même sous la forme d'une barrière exécutée de la manière suivante :

   — On augmente l'état local de l'objet avec un compteur indiquant le nombre de clients ayant ouvert l'objet.

   — Une opération prend effet si, et seulement si, l'objet est ouvert par au moins $f + 1$ clients. Dans le cas où l'opération n'a pas d'effet, une exception est levée.

3. Mise en œuvre

Nous avons mis en œuvre notre construction universelle au-dessus d'Infinispan, un système de stockage de données libre, largement répandu aujourd'hui dans les infrastructures supportant l'informatique en nuage. Le code source fait environ 1000 lignes ; il est ouvert et disponible publiquement [6].
Notre mise en œuvre de l'Algorithme 1 s'appuie sur la librairie Javassist [4] et l'interface Java d'Infinispan. Les clients partagent un objet (logique) par l'intermédiaire d'un proxy [29]. Le proxy est un objet Javassist, local au client et du même type de données que l'objet. En interne, un proxy utilise Infinispan pour exécuter les opérations qui sont invoquées par le client. en suivant l'approche que nous avons décrite à la Section 2.
De manière plus détaillée, nous utilisons la réflexion offerte par Javassist pour intercepter les appels aux opérations de l'objet. Ces appels sont transformés en un tableau d'octets, avant d'être transmis vers Infinispan conjointement avec l'identifiant du client et le drapeau approprié. Pour construire un objet partagé, un client passe la classe de l'objet à un constructeur. Cette classe doit être serialisable au sens de Java, afin que l'état de l'objet puisse être transmis sur le réseau. Le constructeur instancie localement la classe, et retourne le proxy Javassist sur lequel le client effectuera ces appels. Dans l'état actuel de notre code, la destruction d'un objet nécessite un appel explicite, et n'est pas intégrée dans le ramasse-miettes de Java.

4. Evaluation

Dans cette section, nous présentons une série de résultats expérimentaux mettant en jeu notre construction universelle. Ces résultats consistent dans un premier temps à valider les optimisations qui ont été décrites dans la Section 2. Nous illustrons ensuite comment notre approche permet l'ajout de fonctionnalités à un ECV existant, en dotant Infinispan d'un système de versionage. Enfin, nous détaillons comment notre approche permet de porter facilement une application concurrente existante, précisément un robot d'indexation, vers une infrastructure du type informatique en nuage.

4.1. Optimisation des lectures

Notre première expérience consiste à la validation des optimisations décrites à la fin de la Section 2.2. Pour ce faire, nous avons effectué des accès successifs à des types de données différents en activant, ou non, le support des annotations pour les opérations en lecture seule. Ces expériences sont réalisées sur une machine de 4 cœurs cadencés à 2.9GHz, et sur laquelle nous exécutons 4 nœuds Infinispan répliquant les même données. Il est à noter ici que cet environnement est celui dans lequel notre optimisation est la moins utile, du fait du coût quasi nulle de la communication par le réseau entre les nœuds de l'ECV.
La Table 1 présente nos résultats pour un client effectuant 10^4 accès. Les types considérés sont une table de hachage, une file d'attente et un compteur. La proportion d'opérations ne modifiant pas l'état varie de 10 à 80%. Comme escompté, nous observons que pour ces trois types de données, l'exécution en local des opérations de lecture seule améliore les performances de l'objet partagé de manière importante.

4.2. Support du multi-version

Notre approche permet une extension aisée des possibilités d'un ECV existant. Afin de valider ce point, nous avons ajouté un support du multi-version au-dessus d'Infinispan. L'opération $put(k, v)$ (respectivement $get(k)$) a été étendue afin d'insérer de manière transparente une nou-

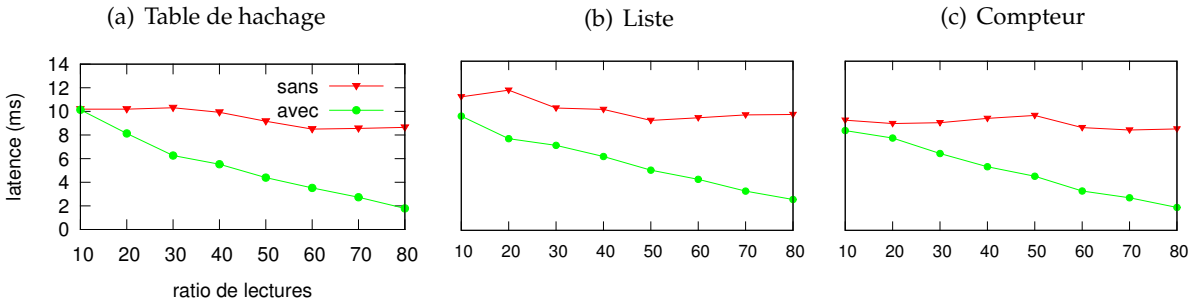

TABLE 1 – Effet de l'optimisation des lectures

velle version de la clé k (resp. retourner la dernière version de k). Par ailleurs, l'opération $get(k, v_1, v_2)$ a été ajoutée. Cette opération retourne l'ensemble des valeurs de la clé k comprises entre les versions v_1 et v_2.

Cette interface étendue a été construite de trois manières distinctes : D'une part, en utilisant notre approche pour partager une relation indexant de façon ordonnée les versions disponibles d'une même clé (classe TreeMap de Java). D'autre part, via une table de hachage à grains fins disponible dans Infinispan. La table est stockée entièrement dans la valeur de la clé. Notons ici qu'une opération d'ajout/suppression est effectuée à l'aide d'un delta, et ne nécessite donc pas une recopie entière de la table. Enfin, en utilisant le support qu'offre Infinispan de l'indexation des objets par Lucene, et les requêtes Hibernate associées [2]. Le code source de ces trois constructions fait environ 1100 lignes ; il est accessible en ligne [6].

La Table 2 détaille les résultats que nous avons obtenus dans différents scénarios, pour les trois mises en œuvre du versionage décrites ci-dessus. Les paramètres expérimentaux sont les mêmes qu'au cours des expériences menées dans la Section 4.1, à savoir une machine multi-cœurs sur laquelle nous exécutons plusieurs nœuds Infinispan. Le support de la persistance pour Lucene et Hibernate est désactivé.

Nous observons dans la Table 2 que l'approche utlisant les annotations Hibernate est la moins performante. Ceci provient du fait que le stockage est couteux : un document est créé pour chaque nouvelle version. Par ailleurs, les requêtes Lucene ne supportent pas les opérateurs *max* et *min*. Par conséquent, trouver la dernière version d'une clé nécessite de parcourir la totalité de l'index. Ce dernier problème se retrouve aussi dans le support du multi-version à l'aide d'une table de hachage à grains fins. A l'opposé, la mise en œuvre d'une collection via l'Algorithme 1 permet un tri à l'insertion, assurant des performances proches des opérations natives de l'ECV.

4.3. Robot d'indexation

Dans cette section, nous cherchons à démontrer que notre approche facilite le portage d'applications existantes vers les infrastructures de l'informatique en nuage. Pour ce faire, nous avons modifié le code d'un robot d'indexation, Flaxcrawler [1], afin que celui-ci utilise des objets partagés répartis mis en œuvre à l'aide de l'Algorithme 1.

Dans les grandes lignes, le fonctionnement de Flaxcrawler est le suivant : Une table de hachage stocke les pages traversées par le robot et les indexe par leurs adresses. Lorsque le robot rencontre une page, il extrait les liens de la page et les insère dans une file d'attente. Dans l'état initial, la file d'attente est peuplée par un ensemble de liens définissant le point de départ du robot. Flaxcrawler est paramétrable, et élastique verticalement. L'utilisateur peut définir lors du lancement du robot, le type de liens conservés, l'ensemble des domaines navigués par le robot,

Scénario	Sans multi-version	Table de hachage à grains fins	Algorithme 1	Hibernate
$put(1$ clé, 10^4 vers.$)$	0.68	2.11	1.46	13.74
$put(10$ clés, 1000 vers.$)$	0.78	1.28	1.56	3.99
$put(1000$ clés, 10 vers.$)$	0.90	1.27	2.73	2.45
$get(10^4$ vers.$)$	0.014	1.28	0.03	21.56
$get(v_1, v_2, 10^4$ vers.$)$	x	2.49	0.094	21.83

TABLE 2 – Support du multi-version *– temps moyen (en ms) pour effectuer une opérartion.*

ainsi que la profondeur de la navigation.
Nous avons changé le code de cette application afin de (i) stocker les pages traversées par le robot à l'aide de l'ECV Infinispan, et (ii) utiliser une file atomique mise en œuvre par notre approche afin de maintenir la liste des liens à parcourir. L'ensemble des modifications nécessaires pour passer d'une exécution locale à une exécution répartie sur plusieurs machines utilisant des objets partagés fait quelques dizaines de lignes de code [5].

5. État de l'art

Plusieurs systèmes employés dans l'informatique en nuage présentent une interface pour synchroniser les processus d'une application répartie. Microsoft Azure [13] offre pour ce faire un mécanisme à base de verrous partagés. Les clients de Google Chubby [12] font usage d'un système de baux, afin de garantir un accès exclusif à une ressource partagée. L'interface de ZooKeeper [19] consiste en un arbre concurrent, similaire à un système de fichiers UNIX. Ces systèmes utilisent la journalisation des opérations pour assurer la cohérence. Corfu [9] met en œuvre un journal atomique par l'intermédiaire d'un support physique dédié. Avec Tango [10], les développeurs ont accès à une construction universelle où la cohérence de chaque objet est assuré par un journal mis en œuvre par Corfu.
La réplication de machine à états (SMR) [28] est une technique classique dans la construction de systèmes répartis fiables. Elle permet à un ensemble de répliques de se mettre d'accord sur l'ordre dans lequel appliquer les opérations sur leur copie locale d'un objet partagé. L'approche SMR s'appuie sur le consensus, et cette primitive de communication de groupe est au cœur des systèmes que nous avons vus précédemment. Des travaux récents, tels que [23, 26], remarquent qu'il n'est pas nécessaire de garantir un ordre pour les opérations commutatives. Cette observation a été utilisée récemment [21] dans la construction d'une base de données répartie.
COPS [24] met en œuvre un ECV causalement cohérent à l'aide d'une gestion des conflits entre opérations concurrentes proche des mécanismes proposés par Bayou [33]. Les auteurs de Eiger [25] étendent cette idée en offrant un accès aux données de type famille de colonnes. Ce modèle est intermédiaire entre le modèle relationel hérité des bases de données, et les stockages plats des ECV. COPS supporte aussi des transactions non bloquantes en écriture et en lecture quand celles-ci sont locales à une partition de données.
Le système Scatter [17] utilise des groupes de répliques synchronisés par consensus afin de construire une topologie en anneau. Il supporte la fusion et la division des groupes à l'aide de transactions entre les groupes. Au-dessus de cette topologie, le système fournit une interface clé-valeur. Le routage s'effectue en $O(1)$ sauts entre les groupes du fait que chaque participant maintient de manière cohérente une table d'assignation entre les morceaux de l'anneau et les groupes.
De manière similaire à Scatter, CATS [8] propose une interface composée d'un ensemble de

registres atomiques. La différence fondamentale entre les deux systèmes tient dans le routage, qui est basé dans le cas de CATS sur un hachage cohérent [20]. Afin d'assurer la cohérence forte des opérations, les groupes dynamiques de CATS exécutent un algorithme à la Paxos à l'intérieur de la DHT. La définition d'un groupe est donnée par le facteur de réplication des données. Une telle approche est réminiscente des systèmes de stockage à base de DHT tels que Pastry [27].
Le système géo-réparti Walter [32] permet à chaque site de se comporter comme une réplique fiable des données par l'intermédiaire d'un algorithme de réplication synchrone. Walter offre aux usagers un support des transactions, ces transactions étant isolées par une nouveau critère dénommé PSI. En comparaison de l'isolation instantanée (SI), PSI améliore les performances lors du passage à l'échelle en exposant les clients à des instantanées non-monotones du système. Walter fait aussi usage d'objets de type CRDT [30] afin d'exécuter rapidement les opérations conflictuelles.
Le système Gemini propose de contourner le résultat d'impossibilité CAP [16] en adjoignant au niveau du stockage une information relative à la sémantique applicative. De manière plus détaillée, le développeur d'une application annote les opérations sur les données afin de distinguer les opérations faiblement cohérentes (de couleur bleue), et les opérations fortement cohérentes (annotées rouge). Le système exécute les opérations bleues localement, leurs modifications étant propagées en tâche de fond. Les opérations rouges sont elles sérialisées les unes par rapport aux autres. Une telle approche est héritée des travaux menés par Fekete et al. [15] sur le système ESDS. La différence entre les deux systèmes est la division par Gemini de chaque opération entre un générateur, qui retourne une valeur mais n'a aucun effet de bord, et une ombre qui execute les modifications a properment parlées. Seules les ombres sont annotées par une couleur. L'idée d'isoler l'effet de bord dans une opération a été proposée précédement par Shapiro et al. dans leurs travaux sur les types de données répliqués commutatifs [31].

6. Conclusion

Dans cet article, nous présentons une méthode de mise en œuvre d'objets partagés au-dessus d'un entrepôt clé-valeur (ECV). Ce mécanisme est universel car il permet la construction d'objets quelconques. Par ailleurs, il est non intrusif et assure aux objets ainsi construits d'hériter des propriétés du ECV sous-jacent, à savoir sa cohérence, son élasticité et sa durabilité. Nous validons l'intérêt de notre approche, par une série de résultats expérimentaux. En particulier, nous illustrons comment cette construction permet l'ajout de fonctionnalités à un ECV existant, et facilite le portage d'une application concurrente existante vers une infrastructure du type informatique en nuage.

Références

[1] Flaxcrawler, *http ://code.google.com/p/flaxcrawler*.

[2] Hibernate, *http ://hibernate.org*.

[3] Infinispan, *http ://infinispan.org*.

[4] Javassist, *http ://www.javassist.org*.

[5] Leads crawler, *http ://github.com/otrack/leads-crawler*.

[6] Leads infinispan, *http ://github.com/otrack/leads-infinispan*.

[7] Wikipedia - Cloud Database, *http ://en.wikipedia.org/wiki/cloud_database*.

[8] Arad (C.), Shafaat (T.) et Haridi (S.). – Brief announcement : Atomic consistency and partition tolerance in scalable key-value stores. *In : Distributed Computing*, éd. par Aguilera (M.), pp. 445–446. – Springer Berlin Heidelberg, 2012.

[9] Balakrishnan (M.), Malkhi (D.), Davis (J. D.), Prabhakaran (V.), Wei (M.) et Wobber (T.). – Corfu : A distributed shared log. *ACM Trans. Comput. Syst.*, vol. 31, n4, décembre 2013, pp. 10 :1–10 :24.

[10] Balakrishnan (M.), Malkhi (D.), Wobber (T.), Wu (M.), Prabhakaran (V.), Wei (M.), Davis (J. D.), Rao (S.), Zou (T.) et Zuck (A.). – Tango : Distributed data structures over a shared log. – In *24th ACM Symposium on Operating Systems Principles*, *SOSP*, SOSP, 2013.

[11] Ban (B.). – Jgroups : A Toolkit for Reliable Multicast Communication, 2007.

[12] Burrows (M.). – The chubby lock service for loosely-coupled distributed systems. – In *7th symposium on Operating systems design and implementation*, *OSDI*, OSDI, 2006.

[13] Calder (B.), Wang (J.), Ogus (A.), Nilakantan (N.), Skjolsvold (A.), McKelvie (S.), Xu (Y.), Srivastav (S.), Wu (J.), Simitci (H.), Haridas (J.), Uddaraju (C.), Khatri (H.), Edwards (A.), Bedekar (V.), Mainali (S.), Abbasi (R.), Agarwal (A.), Haq (M. F. u.), Haq (M. I. u.), Bhardwaj (D.), Dayanand (S.), Adusumilli (A.), McNett (M.), Sankaran (S.), Manivannan (K.) et Rigas (L.). – Windows azure storage : A highly available cloud storage service with strong consistency. – In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, *SOSP '11*, SOSP '11, pp. 143–157, New York, NY, USA, 2011. ACM.

[14] DeCandia (G.), Hastorun (D.), Jampani (M.), Kakulapati (G.), Lakshman (A.), Pilchin (A.), Sivasubramanian (S.), Vosshall (P.) et Vogels (W.). – Dynamo : amazon's highly available key-value store. – In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, *SOSP '07*, SOSP '07, pp. 205–220, New York, NY, USA, 2007. ACM.

[15] Fekete (A.), Gupta (D.), Luchangco (V.), Lynch (N.) et Shvartsman (A.). – Eventually-serializable data services. *Theoretical Computer Science*, vol. 220, nSpecial issue on Distributed Algorithms, 1999, pp. 113–156.

[16] Gilbert (S.) et Lynch (N.). – Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, vol. 33, n2, 2002, pp. 51–59.

[17] Glendenning (L.), Beschastnikh (I.), Krishnamurthy (A.) et Anderson (T.). – Scalable consistency in scatter. – In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, *SOSP '11*, SOSP '11, pp. 15–28, New York, NY, USA, 2011. ACM.

[18] Herlihy (M. P.) et Wing (J. M.). – Linearizability : A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, vol. 12, n3, juillet 1990, pp. 463–492.

[19] Hunt (P.), Konar (M.), Junqueira (F. P.) et Reed (B.). – Zookeeper : wait-free coordination for internet-scale systems. – In *2010 USENIX technical conference*, *ATC*, ATC, 2010.

[20] Karger (D.), Lehman (E.), Leighton (T.), Panigrahy (R.), Levine (M.) et Lewin (D.). – Consistent hashing and random trees : distributed caching protocols for relieving hot spots on the world wide web. – In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, *STOC '97*, STOC '97, pp. 654–663, New York, NY, USA, 1997. ACM.

[21] Kraska (T.), Pang (G.), Franklin (M. J.), Madden (S.) et Fekete (A.). – MDCC : Multi-data center consistency. – In *8th ACM European Conference on Computer Systems*, *EuroSys*, EuroSys, 2013.

[22] Lamport (L.). – How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, vol. 46, n7, 1997, pp. 779–782.

[23] Lamport (L.). – *Generalized Consensus and Paxos*. – Rapport technique nMSR-TR-2005-33, Microsoft, March 2005.

[24] Lloyd (W.), Freedman (M. J.), Kaminsky (M.) et Andersen (D. G.). – Don't settle for eventual : Scalable causal consistency for wide-area storage with cops. – In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, *SOSP '11*, SOSP '11, pp. 401–416, New York, NY, USA, 2011. ACM.

[25] Lloyd (W.), Freedman (M. J.), Kaminsky (M.) et Andersen (D. G.). – Stronger semantics for low-latency geo-replicated storage. – In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, *nsdi'13*, nsdi'13, pp. 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[26] Pedone (F.) et Schiper (A.). – Handling message semantics with generic broadcast protocols. *Distributed Computing*, vol. 15, 2002.

[27] Rowstron (A. I. T.) et Druschel (P.). – Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. – In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, *Middleware '01*, Middleware '01, pp. 329–350, London, UK, UK, 2001. Springer-Verlag.

[28] Schneider (F. B.). – Implementing fault-tolerant services using the state machine approach : a tutorial. *ACM Comput. Surv.*, vol. 22, n4, 1990, pp. 299–319.

[29] Shapiro (M.). – Structure and Encapsulation in Distributed Systems : the Proxy Principle. – In *icdcs*, pp. 198–204, Cambridge, MA, USA, États-Unis, 1986. IEEE.

[30] Shapiro (M.), Preguiça (N.), Baquero (C.) et Zawirski (M.). – Conflict-free replicated data types. – In Défago (X.), Petit (F.) et Villain (V.) (édité par), *Stabilization, Safety, and Security of Distributed Systems (SSS)*, *Lecture Notes in Comp. Sc.*, volume 6976, pp. 386–400, Grenoble, France, octobre 2011. Springer-Verlag.

[31] Shapiro (M.), Preguiça (N.), Baquero (C.) et Zawirski (M.). – *Conflict-free Replicated Data Types*. – Research Report nRR-7687, INRIA, juillet 2011.

[32] Sovran (Y.), Power (R.), Aguilera (M. K.) et Li (J.). – Transactional storage for geo-replicated systems. – In *Symp. on Operating Systems Principles*, *SOSP '11*, SOSP '11, pp. 385–400, New York, NY, USA, 2011.

[33] Terry (D. B.), Theimer (M. M.), Petersen (K.), Demers (A. J.), Spreitzer (M. J.) et Hauser (C. H.). – Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, vol. 29, n5, décembre 1995, pp. 172–182.