Edition collaborative sur réseau pair-à-pair à large échelle

Gérald Oster, Pascal Urso, Pascal Molli, Abdessamad Imine

Édition collaborative sur réseau pair-à-pair à large échelle

Gérald Oster, Pascal Urso, Pascal Molli, Abdessamad Imine
Projets ECOO et CASSIS, LORIA
{oster,urso,molli,imine}@loria.fr

Résumé

Un éditeur collaboratif synchrone permet à plusieurs utilisateurs d'éditer les mêmes données au même moment depuis différents sites répartis sur Internet. Pour garantir un temps de réponse minimum, les données sont répliquées de manière optimiste. Il n'y a ni verrou, ni protocole de sérialisation. Si les utilisateurs génèrent des opérations concurrentes, le système doit assurer que les répliques convergent vers le meilleur état possible.

WOOT est un algorithme de réplication optimiste adapté aux éditeurs collaboratifs. Il assure la convergence des répliques et préserve les intentions. WOOT est plus simple et plus efficace que les approches existantes. WOOT ne nécessite pas de vecteur d'horloges et peut être déployé massivement sur un réseau P2P.

1 Introduction

Pour optimiser les performances, les éditeurs collaboratifs synchrones sont basés sur une réplication optimiste des données. Comme les systèmes traditionnels de réplication optimiste, ils assurent la convergence i.e. quand le système est au repos, les répliques sont identiques. À la différence des systèmes traditionnels, ils doivent aussi préserver les intentions. Cela signifie qu'une opération produit le même effet lorsqu'elle est rejouée que lorsqu'elle a été générée. Sur une structure linéaire, préserver les intentions c'est préserver les relations d'ordre entre les éléments de la structure. Par exemple, si un caractère 'a' a été inséré entre les caractères 'b' et 'c', alors il le sera sur toutes les répliques. L'approche des transformées opérationnelles (OT) cherche depuis des années des algorithmes capables d'assurer la convergence et le respect des intentions. Dans cette approche, une opération reçue est transformée pour pouvoir être rejouée sur l'état courant de la réplique. Cette manière de faire ne défait pas les opérations locales avant d'intégrer les opérations distantes. Malheureusement, ces transformations ne préservent pas les relations d'ordre. Si dans une chaîne de caractères, 'x' est inséré devant 'y' et 'y' devant 'z', il est possible de converger vers un état où 'x' est après 'z' [3].

Pour résoudre ce problème, nous avons décidé de quitter l'approche OT pour une nouvelle approche que nous avons baptisée WOOT. L'idée de cette approche est simple : au lieu de recalculer les relations d'ordre a posteriori comme dans OT, nous diffusons les relations d'ordre avec les opérations. Ainsi, nous sommes déjà sûr de préserver les intentions. Par exemple, quand un utilisateur observe la chaîne de caractères `"ABCDE"` et qu'il insère `"12"` à la position 2, on peut supposer qu'il veut insérer `"12"` entre `"A"` et `"B"`. Respecter les effets de cette opération consiste à préserver les relations `"A"` $\prec$ `"12"` $\prec$ `"B"` sur tous les futurs états de la chaîne. Nous adaptons le profil des opérations. Au lieu de diffuser l'opération $insert(2, \texttt{"12"})$, nous diffusons l'opération $insert(\texttt{"A"} \prec \texttt{"12"} \prec \texttt{"B"})$. Un premier problème se pose : comment exécuter une opération $insert(\texttt{"A"} \prec \texttt{"12"} \prec \texttt{"B"})$ si nous avons localement supprimé le caractère `"A"` ? Nous choisissons de ne pas détruire les caractères, nous les marquons comme invisibles. Cependant la complexité mémoire de notre algorithme reste inférieure ou égale à celle des algorithmes

existants. En effet, nous ne conservons pas le journal des opérations et nous n'utilisons pas de vecteur d'horloges.

Chaque opération d'insertion génère deux nouvelles relations. L'ensemble des relations constitue un ordre partiel. Considérons trois sites qui génèrent chacun une opération comme décrite dans la figure 1. Le diagramme de Hasse de la figure 2 représente l'ensemble des relations entre les caractères.

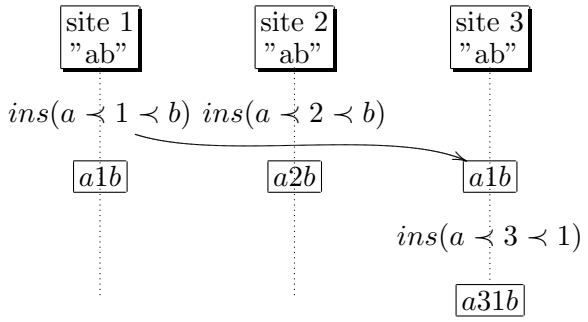

FIG. 1 – Génération d'un ordre partiel.

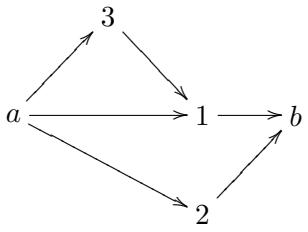

FIG. 2 – Diagramme de Hasse des relations d'ordre entre les caractères.

Les états de convergence où les intentions sont préservées correspondent à l'ensemble des extensions linéaires de cet ordre partiel i.e. `a312b`, `a321b`, `a231b`. Pour assurer la convergence des répliques, tous les sites doivent trouver la même extension linéaire.

L'objectif de cette nouvelle approche est donc de garantir la convergence en utilisant une fonction de linéarisation. Dans cet article, nous nous intéressons à une chaîne de caractères, mais cette approche peut être adaptée à n'importe quelle structure linéaire. Cette structure linéaire peut être complexe comme un document XML modélisé sous la forme un arbre ordonné. Dans cet article, nous présentons le modèle et les algorithmes composant WOOT. Puis, nous calculons sa complexité et déterminons sa correction. Enfin, nous comparons WOOT avec les approches existantes. Le détail des algorithmes ainsi que les preuves formelles sont disponibles dans [4].

2 WOOT : le modèle

Notre modèle de réplication optimiste est correct si et seulement si :

Respect des pré-conditions Les opérations sont intégrées si leurs pré-conditions sont satisfaites.

Convergence des répliques Lorsque tous les sites ont exécuté les mêmes opérations, les répliques sont identiques.

Respect de l'intention Pour toute opération *op*, les effets de l'exécution de *op* sur tous les sites sont les mêmes que les effets de l'exécution de l'opération *op* sur son état de génération.

Nous appelons ce modèle de cohérence PCI pour Pré-conditions, Convergence et Intention.

WOOT gère des W-caractères. Un W-caractère possède un identifiant unique, un booléen indiquant si le caractère est visible ainsi que les identifiants du caractère précédent et du suivant. L'identifiant unique est constitué du numéro de site de génération et de la valeur de l'horloge logique de ce site au moment de la génération.

Chaque site possède donc un identifiant unique, une horloge logique, une séquence de W-caractères et un ensemble d'opérations en attente d'intégration. Il existe deux types d'opérations affectant une chaîne de caractère.

ins$(c_p \prec c \prec c_n)$ insère le W-caractère c entre son précédent c_p et son suivant c_n. Les W-caractères précédent et suivant doivent exister.

del(c) supprime le W-caractère c. Le W-caractère c doit exister.

La fermeture transitive des relations entre un caractère, son précédent et son suivant forment un ordre partiel appelé *relation de précédence* $\prec$. Pour obtenir une chaîne de caractères à partir de cet ordre partiel, nous devons trouver une extension linéaire i.e. un ordre total. Cet ordre $\leq_S$ est défini localement sur chaque site S. Il doit être compatible avec l'ordre partiel $\prec$ ainsi qu'avec l'ordre total des autres sites. Lorsque nous ne pouvons pas établir de relation de précédence entre deux caractères, nous les ordonnons en comparant les identifiants des W-caractères. Cela forme l'ordre $<_{id}$.

3 Algorithme

Quand une réplique est modifiée, une opération correspondante est générée. Cette opération est 1. immédiatement intégrée en local, 2. diffusée aux autres sites, 3. reçue par les autres sites, 4. intégrée par les autres sites.

Du point de vue de l'utilisateur, la chaîne ne contient que les caractères visibles. Une opération générée doit être traduite en une opération basée sur les relations d'ordre. Par exemple, l'opération $ins(2, a)$ effectuée dans la chaîne de caractères `'xyz'` sera traduite en l'opération $ins(x \prec a \prec y)$. L'ordre de réception des opérations peut être différent de l'ordre de génération. Ainsi, une opération peut être reçue sur un site, mais ne pas pouvoir y être exécutée car ses pré-conditions ne sont pas vraies. L'intégration est alors retardée jusqu'à ce que les pré-conditions soient vraies.

```
1  IntegrateIns(c, c_p, c_n)
     Soit S la chaîne de caractères maintenue localement
3    Soit S' la sous-séquence de S entre c_p et c_n
     Si S' est vide Alors insérer c dans S devant c_n
5    Sinon
       Soit L := d_0 d_1 ... d_m où d_0 ... d_m sont les W-caractères de S'
7            qui ont leur précédent avant c_p et leur suivant après c_n
       Soit L' := c_p L c_n ; i := 1
9      TantQue (i < |L'| - 1) et (L'[i] <_id c) Faire
           i := i + 1
11     IntegrateIns(c, L'[i-1], L'[i])
     FinSi
```

FIG. 3 – Procédure d'intégration d'une opération d'insertion.

Comme nous ne détruisons pas les caractères, l'intégration d'une opération $del(c)$ est triviale. Nous rendons simplement le caractère invisible, et ce, qu'il soit visible ou non. L'intégration d'une opération $ins(c_p \prec c \prec c_n)$ est plus délicate. Nous devons placer c parmi tous les caractères situés entre c_p et c_n. Ces caractères peuvent être des caractères qui ont été supprimés précédemment ou des caractères qui ont été insérés en concurrence. Ce traitement est réalisé par la procédure $IntegrateIns(c, c_p, c_n)$ décrite dans la figure 3.

L'algorithme doit ordonner les caractères qui n'ont pas de relation de précédence selon la relation $<_{id}$. L'algorithme filtre tous les caractères de S' qui possèdent leur prédécesseur ou leur successeur dans S'. Ces caractères sont ordonnés selon la relation de précédence. Tous les caractères restants sont triés selon la relation $<_{id}$. L'algorithme insère donc le caractère c à la bonne position selon la relation $<_{id}$. Des caractères peuvent exister entre $L'[i-1]$ et $L'[i]$. Le

caractère c doit être ordonné par rapport à ces caractères, c'est pourquoi l'algorithme effectue un appel récursif.

Exemple Soient trois sites site 1, site 2 et site 3 possédant chacun une réplique d'une chaîne vide "$c_b c_e$". Nous considérons le scénario décrit dans la figure 4. Le diagramme de Hasse correspondant est décrit par la figure 5.

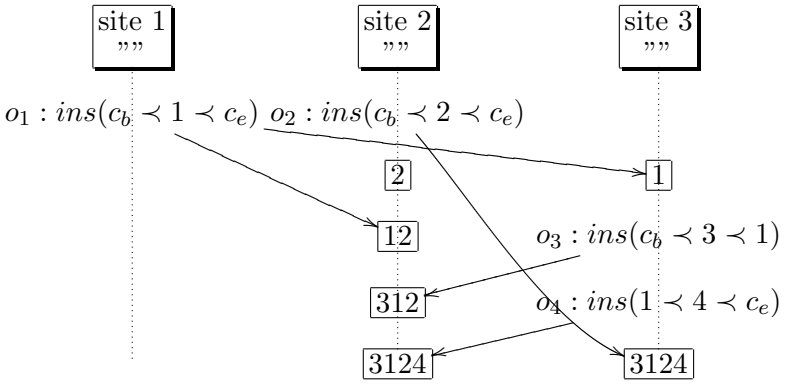

FIG. 4 – Scénario d'exemple.

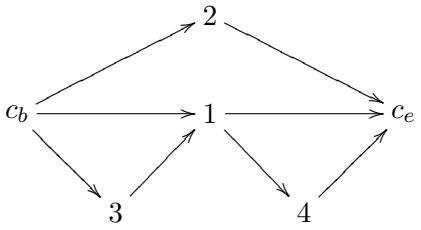

FIG. 5 – Diagramme de Hasse résultant du scénario d'exemple.

La relation $<_{id}$ ordonne les caractères de la manière suivante : '1' $<_{id}$ '2' $<_{id}$ '3' $<_{id}$ '4'. Le site 2 reçoit les opérations o_1, o_3, o_4 dans cet ordre. L'intégration se déroule de la manière suivante :

1. $IntegrateIns(1, c_b, c_e)$: $S' = L =$ "2" et $1 <_{id} 2$, WOOT intègre '1' entre c_b et '2'. Au cours de l'appel récursif $IntegrateIns(1, c_b, 2)$, nous obtenons $S' = \varnothing$. Et donc, le résultat obtenu est la chaîne "$c_b 12 c_e$".
2. $IntegrateIns(3, c_b, 1)$: $S' = \varnothing$, WOOT insère '3' entre c_b et '1'.
3. $IntegrateIns(4, 1, c_e)$: $S' = L =$ "2" et $2 <_{id} 4$, WOOT intègre '4' entre '2' et 'c_e' . Nous obtenons au final la chaîne "$c_b 3124 c_e$".

Sur le site 3, l'appel $IntegrateIns(2, c_b, c_e)$ est résolu : $S' =$ "314" et $C_N(3) = C_P(4) = 1$, nous avons $L =$ "1". Comme $1 <_{id} 2$, WOOT intègre '2' entre '1' et c_e. Au cours de l'appel récursif $IntegrateIns(2, 1, c_e)$: nous avons $S' = L =$ "4" et $2 <_{id} 4$, WOOT intègre '2' entre '1' et '4'. Sur le site 3, nous obtenons la chaîne "$c_b 3124 c_e$" comme résultat final.

Sur le site 1, quel que soit l'ordre d'arrivée des opérations o_2, o_3, o_4, nous obtenons la chaîne "$c_b 3124 c_e$".

Ainsi tous les sites convergent vers l'état final attendu "3124".

4 Correction et complexité

Une opération op est intégrée uniquement si la fonction $isExecutable(op)$ retourne vrai. Cette fonction retourne vrai lorsque les pré-conditions de l'opération sont satisfaites. Clairement, les pré-conditions des opérations sont respectées.

La linéarisation calculée par notre algorithme doit respecter les intentions des opérations. Autrement dit, elle doit préserver la relation de précédence définie sur les opérations à leur génération. Notre algorithme construit sur chaque site S une relation $\leq_S$ qui est une extension linéaire de la relation de précédence $\prec$.

Cependant, respecter les intentions n'est pas suffisant. Par exemple, si deux sites insèrent respectivement 'a' et 'b' entre les mêmes caractères 'x' et 'y', nous pouvons calculer deux chaînes différentes sur les deux sites ; respectivement "axyb" et "ayxb". Ces deux extensions linéaires préservent les intentions, mais les deux répliques ne convergent pas. Nous n'avons pas encore de preuve de convergence de notre algorithme. Cependant, pour vérifier sa correction nous avons utilisé le système formel de vérification de modèle (model-checker) TLC [9] sur une spécification modélisée en TLA+. Les techniques de vérification de modèle sont particulièrement bien adaptées pour vérifier des systèmes concurrents. La spécification utilisée est disponible dans [4]. L'invariant permettant de garantir la convergence est :

Conjecture 1 *Soient deux sites distincts S_1 et S_2, pour chaque paire de W-caractères $\{c, d\}$ qui sont présents dans S_1 et dans S_2, nous avons $c \leq_{S_1} d \Leftrightarrow c \leq_{S_2} d$.*

Toujours dans [4], nous avons démontré que notre algorithme d'intégration terminait. Concernant la complexité algorithmique, notre algorithme a une complexité linéaire en espace et cubique en temps, en fonction du nombre total d'opérations générées.

5 Comparaison avec les approches existantes

Plusieurs éditeurs collaboratifs synchrones reposent sur l'approche des transformées opérationnelles [5, 6, 7]. Cette approche consiste à sérialiser les opérations concurrentes à l'aide d'une fonction de transformation. Pour assurer la convergence des répliques, la fonction de transformation doit respecter certains critères bien définis. Malheureusement, aucun de ces travaux n'a pu assurer le respect de ces critères. Les contre-exemples se trouvent dans [1, 3, 4].

SOCT4 [8] est basé sur les transformées opérationnelles. Il utilise un serveur central pour maintenir un ordre global continu et assure la convergence. Cependant, un ordre global continu est trop coûteux pour être déployé sur un réseau P2P de grande taille.

SDT [3] est basé sur l'approche OT. Cet algorithme est plus bien plus complexe que les autres approches OT. Il semble assurer la convergence des répliques et préserver les intentions. SDT a une complexité en temps de $O(n^3)$ comme WOOT. Cependant, SDT utilise des vecteurs d'horloges et un journal d'opérations. Chaque opération possédant son estampille, la complexité en mémoire est donc proportionnelle au nombre de sites. SDT ne peut donc pas passer à l'échelle sur un réseau P2P de grande taille.

IceCube [2] traite le problème de la réconciliation comme un problème de satisfaction de contraintes. WOOT peut être envisagé comme un problème de satisfaction de contrainte. De ce fait, il est peut-être possible d'exprimer WOOT dans IceCube. Cependant la complexité de WOOT est en $O(n^3)$ alors que celle de IceCube est NP-difficile. De plus IceCube calcule la résolution de contrainte sur un site central. La réconciliation est divisée en plusieurs phases : 1. Chaque site envoie son journal sur un site central, 2. ce site calcule le journal optimal, 3. le journal optimal est renvoyé sur tous les sites pour y être exécuté.

IceCube ne peut donc pas passer à l'échelle sur un réseau P2P de grande taille.

6 Conclusion

WOOT garantit la convergence des répliques et la préservation des intentions pour des données structurées linéaires. Il ne nécessite ni site central, ni ordre global, ni vecteur d'horloges. WOOT peut donc être déployé sur un réseau pair à pair de grande taille. L'édition collaborative massive est une réalité. La Wikipédia est une encyclopédie rédigée collaborativement par des milliers d'internautes et contient déjà plus de 600 000 articles. La Wikipédia repose sur un

éditeur collaboratif centralisé et souffre de problème important de performances. WOOT permet de résoudre ces problèmes en distribuant les tâches d'édition sur un réseau de sites Wikipédia.

Références

[1] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of 8th ECSCW*. ACM Press, 2003.

[2] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the 20th PODC*, pages 210–218. ACM Press, 2001.

[3] D. Li and R. Li. Preserving operation effects relation in group editors. In *Proceedings of CSCW*, pages 457–466. ACM Press, 2004.

[4] G. Oster, P. Urso, P. Molli, and A. Imine. Real time group editors without operational transformation. Technical Report RR-5580, INRIA, 2005.

[5] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of CSCW*, pages 288–297. ACM Press, 1996.

[6] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of GROUP Conference*, pages 435–445. ACM Press, 1997.

[7] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1) :63–108, 1998.

[8] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of CSCW*. ACM Press, 2000.

[9] P. Manolios Y. Yu and L. Lamport. Model checking TLA+ specifications. In *Proceedings of 10th CHARME*, pages 54–66. Springer, 1999.