Méthodologie d'évaluation pour les types de données répliqués

Mehdi Ahmed Nacer

UNIVERSITÉ DE LORRAINE

École doctoral IAEM Lorraine

Méthodologie d'évaluation pour les types de données répliqués

THÈSE DE DOCTORAT

Pour obtenir le titre de

Docteur de l'université de Lorraine

Spécialité : Informatique

Présentée par

Mehdi Ahmed-Nacer

Soutenue le 5 mai 2015 devant un jury composé de :

Rapporteurs :

M. Achour Mostefaoui	Professeur à l'Université de Nantes
M. Benoit Baudry	Chercheur IRISA - INRIA

Examinateurs :

M. Marc Shapiro	Directeur de Recherche INRIA - LIP6
M. Yannick Toussaint	Chercheur INRIA - LORIA

Directeurs de Thèse :

M. François Charoy	Professeur à l'Université de Lorraine
M. Pascal Urso	Maître de Conférence à l'Université de Lorraine

Laboratoire Lorrain de Recherche en Informatique et ses Applications - UMR 7503

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Remerciements

Cette thèse a été préparée au laboratoire LOrrain de Recherche en Informatique et ses Applications (LORIA). Je tiens tout particulièrement à remercier l'ensemble du jury pour l'intérêt porté à mes travaux.

Tout d'abord, je souhaite remercier les rapporteurs Achour Mostefaoui et Benoit Baudry pour leur lecture attentive de mon manuscrit et leurs nombreux commentaires qui m'ont permis de l'améliorer.

Je souhaite également remercier Marc Shapiro et Yannick Toussaint pour leur rôle d'examinateur. Je tiens, de plus, à remercier Marc Shapiro pour avoir accepté d'être président du jury.

Je tiens à remercier mes encadrants Pascal Urso et François Charoy, qui ont dirigé mes travaux durant mes quatre années de thèse. Je les remercie pour leur grande disponibilité ainsi que les commentaires et suggestions pertinentes dont il m'ont fait bénéficier.

Je tiens aussi à remercier Claude Godart pour m'avoir permis de découvrir le monde de la recherche et de l'enseignement.

Je ne pourrais pas finir sans remercier tous ceux qui m'ont accompagné durant ces années, je pense bien sûr aux membres de l'équipe Coast et tout particulièrement à Gérald Oster, Claudia Ignat, Luc Andé, Ahmed Bouchami, Khalid Benali, Samir Youcef mais aussi à Yacine Bouzidi, Mohamed Ghettassi, Bouguelia Rafik et Mourad Amziani.

Enfin, pour son soutien et ses encouragements durant mes longues années d'études je voudrais témoigner à ma famille toute ma reconnaissance.

A mes parents.

Sommaire

Chapitre 1

Introduction Générale

Sommaire

1.1 Motivation

La réplication optimiste [93] est un moyen qui permet à des sites distribués de partager la même donnée, avec une disponibilité élevée et sans synchronisation. Ce type de réplication maintient une copie locale des données dite *réplique*, sur laquelle des lectures et des écritures sont exécutées. Elle fournit une disponibilité permanente des données même en cas de déconnexion, et réduit la latence du réseau par l'accès local à la réplique [93]. Les systèmes de gestion de base de données NoSQL tels que Cassandra [56], Dynamo [28] ou Riak [3] ; les systèmes de gestion de versions décentralisée tels que Git [110], Mercurial [65] ou Bazaar [19] utilisent la réplication optimiste.

Les systèmes d'édition collaborative [34] permettent à plusieurs utilisateurs géographiquement répartis de travailler ensemble pour produire un document. Pour que les utilisateurs observent immédiatement leurs changements et pouvoir continuer la collaboration même en cas de déconnexion, il est important d'utiliser la réplication optimiste dans ces systèmes. Chaque utilisateur travaille sur une copie du document qu'il détient en local. Les modifications sont exécutées localement, elles sont ensuite transmises aux autres répliques. Google Docs [1], Etherpad [37] et Skydrive [5] sont des systèmes d'édition collaborative temps réel utilisant la réplication optimiste.

Lorsque les données répliquées sont modifiables, le système doit s'assurer que l'exécution des opérations produit le même effet sur toutes les répliques. En effet, les modifications concur-

rentes [1] sur le même objet répliqué peuvent entraîner une divergence [2]. Pour assurer la convergence des données répliquées, les systèmes intègrent des algorithmes de réplication optimiste [99, 82, 85, 90, 119, 34, 105, 113]. Les performances du système d'édition collaborative dépendent des performances de ces algorithmes. Il est donc important de mesurer et d'étudier leurs performances.

Les algorithmes de réplication optimistes utilisent des structures de données différentes et la manière dont ils contrôlent la concurrence n'est pas la même. De plus, plusieurs autres paramètres peuvent influencer les performances des algorithmes tels que les paramètres réseau, les caractéristiques des machines, le comportement des utilisateurs, l'architecture déployée, etc. Une étude théorique ne prend pas en considération ces paramètres et n'est pas suffisante pour comprendre l'exécution des algorithmes de réplication optimiste. Une évaluation expérimentale est donc indispensable.

Jusqu'à présent, aucune étude n'a été réalisée sur des données réelles incluant la concurrence et aucune étude ne démontre l'intérêt des algorithmes de réplication optimiste dans les systèmes asynchrones. La question du choix de l'algorithme le plus performant reste ouverte et leur pertinence est une question de performance et de qualité du résultat.

Le défi de cette thèse est de proposer des méthodes d'évaluation pour les différents algorithmes de réplication sur des traces réalistes incluant la concurrence. Nous développons alors un outil qui unit les différents algorithmes de réplication et garantit une évaluation pertinente. Nous commençons par évaluer les performances des algorithmes et la qualité du résultat qu'ils produisent. Nous analysons ensuite les différents scénarios qui réduisent les performances des algorithmes dans les systèmes synchrones et asynchrones. Enfin, nous proposons des corrections afin d'améliorer leur performance et la qualité du résultat.

1.2 Problématique

La réplication des données constitue la clé des systèmes de partage de données distribuées. Elle consiste à créer des copies de données, sur des sites [3] différents. Cette technique est fondamentale pour augmenter la disponibilité, la fiabilité des données et la tolérance aux pannes.

L'accès concurrent aux copies d'un même objet peut entraîner une divergence entre les états de ces copies. Pour surmonter ce problème, plusieurs algorithmes de réplication ont été proposés. L'application de ces algorithmes dans un système collaboratif nécessite une étude sur leur performance au préalable. Cette étude permettrait de comprendre leur comportement et de savoir quel est l'algorithme le plus adapté pour quelle situation.

Ce chapitre présente les différents problèmes liés à l'évaluation des systèmes collaboratifs en général et aux systèmes d'édition collaborative en particulier. Nous discuterons des propriétés d'expérimentation et d'évaluation qui nous permettront de valider notre méthodologie. Nous aborderons ensuite les différents critères que nous allons évaluer dans les chapitres suivants. Enfin, nous présenterons un aperçu des contributions apportées dans cette thèse.

1.2.1 Réplication des données

On parle de *réplication des données* si les mêmes données sont dupliquées sur plusieurs sites. Ainsi, chaque site possède une copie dite *réplique* sur laquelle les utilisateurs effectuent des modi-

1. ou modifications parallèles. Il n'existe pas de relation de précédence [57] entre ces modifications.
2. Les utilisateurs n'observent pas le même contenu de l'objet.
3. Périphériques informatiques tels que les serveurs, les ordinateurs de bureau, les ordinateurs portables, les dispositifs mobiles, etc.

fications. Le but de la réplication est d'améliorer la disponibilité des données et les performances du système [105, 71, 79]. La disponibilité est améliorée en permettant l'accès aux données même si certains sites sont déconnectés. Tandis que l'amélioration des performances du système consiste à réduire la latence réseau par l'accès local aux données [93].

Dans un système de partage de données, un *objet* est l'unité minimale de réplication. Une *réplique* est une copie d'un objet stocké sur un site. Nous appelons l'*état* la valeur d'un objet sur une réplique donnée. Chaque objet est défini par un *identifiant*, un *contenu*, un *état* et une interface constituée d'*opérations* [100].

Par d'exemple, dans la figure 1.1, l'objet partagé est un document textuel. Le système possède cinq répliques du document sur cinq sites différents, et chaque utilisateur travaille sur une réplique qu'il possède localement. Pour Harry et Sally, le processus de réplication est transparent. Cela donne l'impression aux utilisateurs de travailler sur une seule réplique alors que chacun modifie sa propre réplique localement. Lorsque l'utilisateur modifie l'état d'une réplique, le processus d'exécution de cette modification dépend du type de réplication utilisé.

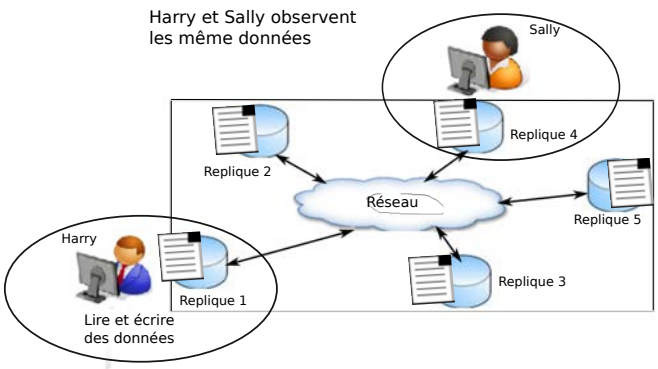


FIGURE 1.1 – La réplication des données

Dans un système collaboratif, il est nécessaire que tous les participants observent à la fin de la collaboration le même état de l'objet partagé. Cependant, en absence d'une horloge globale et en présence de modifications parallèles ou concurrentes, il est difficile de définir un ordre entre ces modifications, ce qui peut entraîner une divergence entre les répliques. Pour assurer la convergence du document, il est nécessaire que l'effet de l'exécution des modifications soient le même sur toutes les répliques. On distingue deux approches de contrôle de concurrence : pessimiste [12] et optimiste [99].

La réplication *pessimiste* donne l'illusion à l'utilisateur qu'il n'existe qu'une seule copie [12]. Il existe plusieurs manières pour atteindre cet objectif. Cependant, le principe reste le même : bloquer l'accès aux données tant qu'il y a des mises à jour en cours [93].

Le théorème CAP [38] qui décrit les dépendances générales entre cohérence (**C**onsistency), disponibilité (**A**vailability) et tolérance aux pannes (**P**artition tolerance), indique que n'importe quel système distribué ne peut à la fois garantir la cohérence des données, la disponibilité et la tolérance aux pannes. Comme les partitions ne sont pas contrôlables et que l'accessibilité est requise par les systèmes et les utilisateurs, un choix consiste à diminuer le niveau de cohérence vers une cohérence à terme. Ce niveau de cohérence signifie que même si les répliques ont à un moment des valeurs différentes, ces valeurs vont inéluctablement converger. Cette propriété est appelée convergence à terme ou convergence inéluctable [114].

Contrairement à la réplication pessimiste, la réplication optimiste ne bloque pas l'accès aux données durant les mises à jour et il n'y a aucune contrainte entre les répliques [99] lors de l'exécution des opérations. Des répliques de l'objet partagé sont stockées sur plusieurs sites. Une réplique peut être modifiée au moyen d'opérations. Quand une réplique est modifiée, l'opération correspondante est immédiatement exécutée en local, puis propagée aux autres sites pour y être ré-exécutée. Lorsque deux répliques de deux sites différents sont modifiées en parallèle, les répliques divergent. Il est donc possible d'observer au même moment une valeur sur un site et une autre valeur sur un autre site. Pour contrôler les modifications concurrentes et assurer la convergence à terme des répliques, les systèmes de partage de données intègrent des algorithmes de réplication optimiste. Cette solution est appliquée sur des systèmes de grandes organisations telles que Cassandra de Facebook [56], Dynamo d'Amazon [28] ou CouchBD de la fondation Apache.

Dans les dernières décennies, les systèmes d'édition collaborative que ce soit pour le développement logiciel ou l'édition collaborative en temps-réel sont très utilisés. Au coeur de ces systèmes, nous trouvons les algorithmes de réplication optimiste. Jusqu'à présent, aucune étude sur les performances de ces algorithmes n'a été réalisée en utilisant des modifications réalistes incluant la concurrence. Pour cette raison, nous nous intéresserons dans cette thèse à l'évaluation des algorithmes de réplication optimiste dans les systèmes d'édition collaborative en utilisant des modifications réalistes incluant la concurrence.

1.2.2 Contexte d'évaluation

Nous présentons dans cette section, le contexte d'évaluation de la performance des algorithmes de réplication.

Les systèmes collaboratifs

Un système collaboratif est constitué d'un ensemble de sites interconnectés par un réseau qui permet à des groupes de personnes de coopérer pour réaliser une tâche commune en fournissant une interface vers un environnement partagé [35].

Dans un environnement distribué, les utilisateurs ne se trouvent pas au même endroit et la coopération ne se fait pas forcément au même moment. Pour créer un environnement favorable à la collaboration, il est nécessaire au système collaboratif de garantir un temps de réponse aussi court que possible. Cela suppose donc que l'accès aux objets de l'environnement partagé ne nécessite pas une communication avec un serveur distant. Pour parvenir à cet objectif, les systèmes collaboratifs se basent sur la réplication des données [93]. Au lieu d'héberger l'objet dans un serveur distant, chaque utilisateur possède une copie locale de l'objet sur laquelle il effectue des modifications. En plus, elle permet l'accès à l'objet partagé même si le serveur tombe en panne ou si l'utilisateur se déconnecte.

Dans notre travail nous nous intéresserons aux systèmes collaboratifs en général et au système d'édition collaborative en particulier.

Les systèmes d'édition collaborative

Un système d'édition collaborative est un système collaboratif dans lequel les utilisateurs collaborent pour rédiger et/ou éditer un document texte, un fichier XML, une image, une vidéo, etc.

Dans une session d'édition collaborative, chaque site possède une réplique du document qui peut être modifiée par des opérations spécifiques. Le nombre de réplique peut être connu ou

inconnu car des sites peuvent se connecter et se déconnecter à tout moment. On suppose que les répliques tolèrent les pannes et que la communication est fiable.

Un système d'édition collaborative est considéré correct s'il maintient les propriétés suivantes [105] :

1. Convergence : la convergence est assurée si à la fin de la collaboration, les répliques qui ont exécuté le même ensemble d'opérations ont le même état document partagé ;
2. Causalité : le respect de la causalité garantit l'exécution des opérations dépendantes dans le même ordre sur toutes les répliques ;
3. Intention : l'effet de l'exécution d'une modification doit être la même sur toute les répliques, et son exécution ne doit pas modifier l'effet d'une modification indépendante.

On distingue deux modes d'édition :

1. Synchrone ou temps-réel : les modifications effectuées par un membre sont immédiatement visibles par les autres utilisateurs. Les systèmes tels que Google Docs et Etherpad sont des systèmes d'édition collaborative synchrones,
2. Asynchrone : les utilisateurs modifient le document en isolation et synchronisent ensuite leur copie pour observer une vision commune du document. Les systèmes Git, SVN et CVS sont des systèmes de gestion de versions et d'édition collaborative asynchrone.

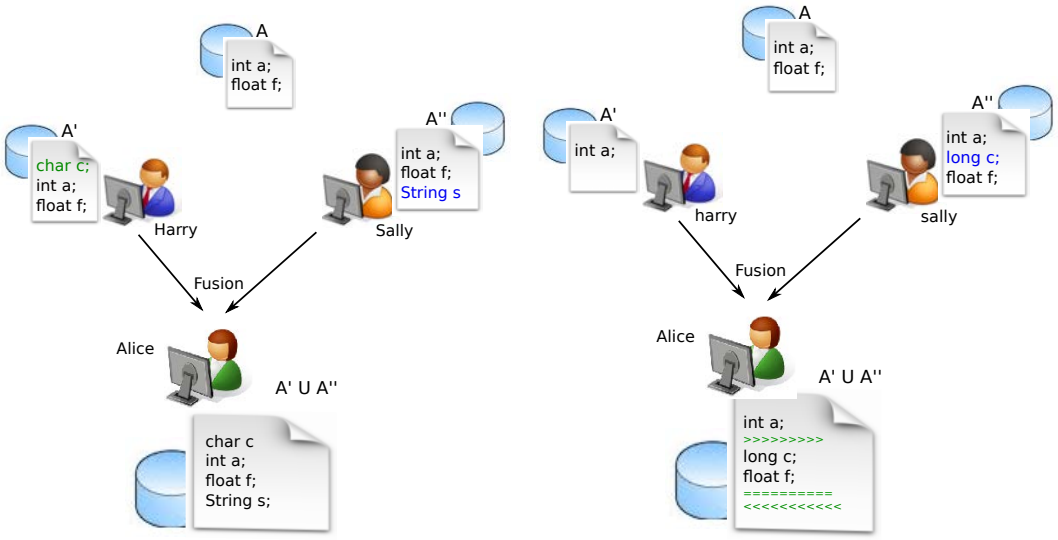
Quel que soit le mode d'édition, les algorithmes de réplication propagent les modifications entre les répliques soit sous forme d'états, on parle alors de système *state-based* [99, 33] soit sous forme d'opérations, on parle alors de système *operation-based* [34, 105, 113].

Propagation par état : State-based

Dans les approches "state-based", la mise à jour est entièrement exécutée dans la réplique source, c'est à dire que toutes les modifications sont exécutées sur la copie source avant d'être transmises aux autres répliques. Ces modifications sont propagées ensuite sous forme d'un seul état entre les répliques. Ces dernières gardent seulement les informations sur les différents états de l'objet mais pas sur l'évolution de l'objet lui-même. Par exemple, dans la figure 1.2a, Sally et Harry partagent initialement le même document `A`. Ensuite, Harry ajoute `"char c;"` dans la première ligne et produit un état A'. En concurrence, Sally ajoute `"String s;"` à la 4ième ligne et produit l'état A". Pour que Alice observe les changements d'Harry et Sally, Alice demande au système de fusionner les deux états. Puisque les modifications ont affecté le document sur deux parties différentes, le système fusionne correctement les modifications. Dans la figure 1.2b, Harry et Sally modifient le document dans la même zone. Harry supprime la deuxième ligne, et en concurrence Sally ajoute `"long c;"` à la même ligne. Lors de la fusion, le système ne peut choisir une version et ignorer l'autre. Afin de tenter de respecter l'intention des utilisateurs, le système génère un conflit, empêche la fusion et retourne le résultat à Alice pour résoudre le conflit. Les systèmes de gestion de versions tels que CVS [112], Git [110] et SVN [24] sont des exemples de système state-based.

Propagation par opérations : Operation-based

Dans les systèmes d'édition collaborative operation-based, lorsque l'utilisateur effectue une modification, cette dernière se transforme en une opération ou un ensemble d'opérations. On parle de *génération d'opérations*. Ces opérations sont *exécutées localement* et sont *diffusées* à toutes les



(a) Modification sans conflit (b) Modification sur la même zone

FIGURE 1.2 – Modification d'un objet répliqué en state-based

autres répliques. Une réplique qui reçoit une opération distante, la ré-exécute localement. C'est ce qu'on appelle l'*intégration* d'une opération.

Contrairement à l'approche state-based, l'approche operation-based garde l'information sur l'évolution de l'objet. Pour chaque modification, un message portant cette information est envoyé à l'ensemble des répliques connectées. La propagation par opérations est souhaitable pour les applications temps réel [7], mais peut être utilisée aussi sur des systèmes de gestion de versions décentralisés [92, 75]. La figure 1.3 présente le même scénario que la figure 1.2b en utilisant la propagation par opérations. Au lieu d'envoyer tout l'état de l'objet, Harry envoie seulement l'opération portant l'information de la suppression : op_1=del(1) pour indiquer la suppression de la première ligne. De même, Sally envoie en parallèle seulement l'opération d'insertion : op_2=ins(1, `"long c;"`). Lorsqu'Alice reçoit op_1 et op_2, elle les exécute localement. Il n'existe pas d'ordre entre les opérations puisqu'elles sont générées en parallèle. Selon l'ordre de leur réception, le résultat pourrait être différent d'une réplique à une autre. Par exemple, si Alice commence par l'exécution de op_1 suivie de op_2, elle produit le document `B`. Alors que si Sally exécute les opération inversement (op_2 suivie de op_1), elle produit le document `A`. Les deux documents sont différents. Pour garantir la convergence du document entre les différentes répliques, les approches transformées opérationnelles (OT) [34, 105, 113] et CRDT [99, 82, 85, 90, 119] ont été proposées.

Les approches transformées opérationnelles (OT) [34, 105, 113] pour le texte changent quand il le faut la position des opérations reçues pour prendre en considération l'effet de la concurrence (opération effectuée en même temps sur une autre réplique). Par exemple, dans la figure 1.3, à la réception de op_1 sur la réplique de Sally, l'algorithme utilisé transforme op_1 par rapport à op_2. Ainsi, op_1 devient del(2), une suppression à la position 2 au lieu de la position 1 puisqu'une opération d'insertion est insérée avant. Pour détecter toutes les opérations concurrentes, les algorithmes OT gardent une trace de toutes les opérations exécutées dans un historique.

Il existe plusieurs applications disponibles au grand public qui adaptent les algorithmes OT. Par exemple, l'application d'édition collaborative Google Docs [1] adapte un algorithme OT centralisé appelé Jupiter[78].

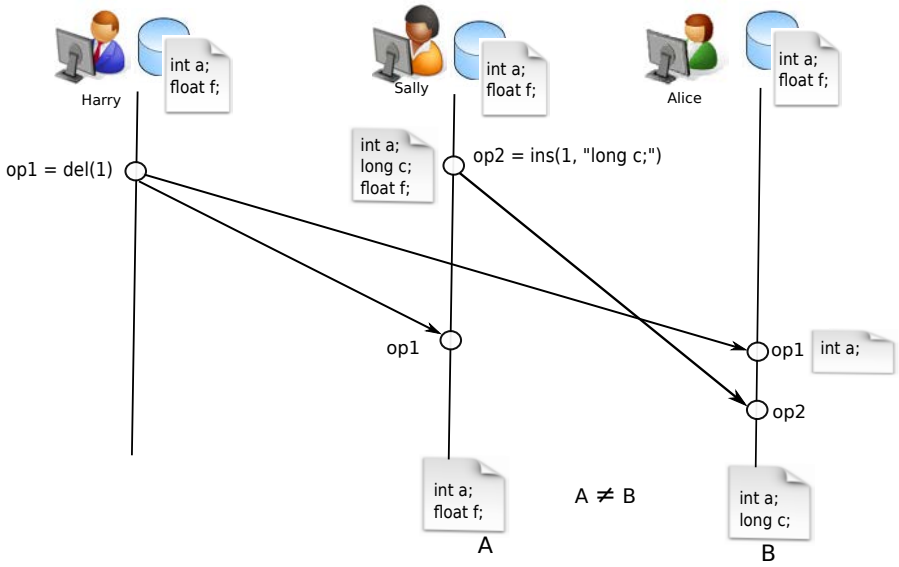


FIGURE 1.3 – Modification d'un objet répliqué en operation-based

L'approche CRDT [99, 82, 85, 90, 119] (**C**ommutative **R**eplicated **D**ata **T**ypes) a été proposée comme une alternative de l'approche OT. Contrairement aux approches OT, les CRDTs ne nécessitent aucune transformation. En effet, les CRDTs ont été conçus pour que l'exécution des opérations concurrentes commute par définition. Cette approche assure un ordre total entre les éléments de l'objet répliqué. Si le système assure que les opérations dépendantes sont reçues dans l'ordre, les types de données CRDT assurent la convergence. Par exemple, dans une édition collaborative, les CRDTs identifient chaque élément de l'objet répliqué par un identifiant de position unique. Un ordre total est assuré entre les identifiants de position. Dans la figure 1.3, soit id_1 et id_2 deux identifiants attribués respectivement à la ligne "int a ;" et à la ligne "float f ;". Ces identifiants sont uniques et un ordre est défini entre eux, soit $id_1 \prec_{id} id_2$. Lorsque Sally exécute op_2 un identifiant id_3 est généré pour la ligne "long c ;" tel que $id_1 \prec_{id} id_3 \prec_{id} id_2$. Lorsqu'Harry exécute op_1, il indique dans l'opération que la ligne supprimée est celle avec l'identifiant id_1. Ainsi, à la réception des opérations, les éléments sont insérés/supprimés correctement et toutes les répliques convergent vers le même document. Cependant, certains algorithmes CRDT doivent garder les traces de toutes les opérations exécutées dans un historique.

Nous détaillerons ces deux approches dans la section 2.3. Les systèmes d'édition collaborative intègrent dans leur système des algorithmes de réplication pour assurer la convergence des répliques. Les performances des applications d'édition collaborative dépendent donc des performances des algorithmes de réplication.

Pour évaluer les performances de ces algorithmes, étudier leur comportement et les améliorer, il faut les confronter à des situations différentes.

1.2.3 Évaluation des performances des algorithmes de réplication optimiste

L'évaluation des performances est essentielle pour le développement de la technologie. Elle permet l'évaluation à la fois de la performance générale des algorithmes de réplication et de mesurer leur progression sur certains objectifs.

Lorsqu'elle est bien structurée, l'évaluation des performances donne l'opportunité de comprendre le comportement des algorithmes et de connaître quel est l'algorithme adapté pour quelle

situation. Cependant, l'investigation sur ces algorithmes doit suivre un modèle d'expérimentation valide.

Une expérimentation [109] permet de tester des hypothèses, une implémentation ou de comparer des algorithmes. Conduire une *bonne expérience* nécessite la vérification de quelques propriétés [43]. Les expérimentations sont produites à travers des outils spécifiques. Ces outils peuvent influencer le résultat de l'expérimentation, et doivent alors respecter quelques propriétés que nous allons voir par la suite.

Propriété d'une expérimentation

Gustedt et al. [43] ont proposé quatre propriétés qui décrivent une *bonne expérience* :

1- Reproductible : une bonne expérimentation doit être reproductible. d'autres chercheurs doivent être capable de reproduire les mêmes résultats avec les mêmes entrées.

Quelques conférences scientifiques exigent aux auteurs d'envoyer leur implémentation avec l'article soumis pour permettre au comité de sélection de reproduire les résultats et de valider l'expérience.

2- Extensible : mesurer les performances d'un algorithme en particulier et décrire l'environnement où les expériences ont été menées n'est pas suffisant et apporte parfois peu d'intérêt à la communauté scientifique. Une bonne expérimentation nécessite une comparaison avec les travaux existants et propose une extension pour un travail futur.

3- Applicable : une bonne expérience doit utiliser des données et des paramètres réalistes. Ceci permet de prédire le comportement des algorithmes et leur applicabilité dans un système réel.

4- Révisable : en raison d'une mauvaise conception, implémentation, modélisation ou de l'environnement expérimental, l'expérimentation peut générer des erreurs. Une bonne expérimentation doit aider l'utilisateur à identifier le problème, l'expliquer et proposer un moyen de le résoudre.

Valider une expérimentation est un défi difficile à relever. Une expérience sur une platforme réelle est souvent non reproductible. Tandis que l'extensibilité, l'applicabilité et la révisabilité sont difficile à réaliser. Pour surmonter ces problèmes, un outil d'expérimentation peut permettre d'accomplir une bonne expérience.

Propriété générale d'un outil d'évaluation

Les outils d'expérimentation existants sont nombreux et sont de nature différente. Cependant, tout outil d'expérimentation devrait partager trois caractéristiques communes [43] :

1- Contrôle : dans le but de savoir quel facteur influence le plus le résultat, il est important que l'outil d'évaluation contrôle les conditions de l'expérimentation. Le contrôle permet de tester et d'évaluer chaque partie indépendamment. Par exemple, rédiger plusieurs scénarii de tests pour étudier le degré d'influence des différents facteurs sur les algorithmes, vérifier leurs limites et proposer des améliorations.

2- Reproductibilité : la reproductibilité est la base du protocole expérimental. Le rôle de l'outil d'expérimentation est d'assurer cette propriété. Cette propriété garantit la reproduction du résultat dans le but de revoir les observations réalisées durant une expérience et d'améliorer les connaissances scientifiques.

3- Abstraction : les conditions expérimentales sont toujours, d'une manière ou d'une autre, une abstraction de la réalité car ils ne peuvent pas prendre en compte tous les paramètres. Un bon outil d'évaluation doit utiliser des données et des paramètres réalistes pour représenter un modèle d'un système réel.

Une fois que l'expérimentation et l'outil d'évaluation sont validés, il est important de décrire les critères d'évaluation des algorithmes. La pertinence des critères d'évaluation utilisés est importante car elle influence de façon conséquente le jugement réalisé. Dans ce qui suit, nous citerons les différentes catégories de critères d'évaluation des algorithmes de réplication optimiste.

Critères génériques d'évaluations

Pour analyser et améliorer les performances des algorithmes de réplication, il est important de comprendre quel sont les paramètres qui influencent ces algorithmes sur les critères suivants :

1- Temps d'exécution Il est important de réduire le temps d'exécution d'une modification dans un système de réplication. Lorsque l'exécution d'une modification ralentit le déroulement d'une tâche, les utilisateurs deviennent frustrés et agacés et peuvent quitter l'application. Si le système prend du temps pour répondre aux utilisateurs, l'affichage des résultats devient lent, ce qui peut conduire à des erreurs.

Dans le contexte d'édition collaborative, des études dans [51, 101] ont démontré que les utilisateurs peuvent observer la mauvaise qualité de l'application, si le système ne répond pas au bout de $50\,ms$. Par exemple, dans une session d'édition collaborative, si l'utilisateur insère un caractère ou un bloc de caractères (copier-coller) et que l'utilisateur ne les observe pas après $50\,ms$, il peut être insatisfait.

2- Occupation de la mémoire Pour garantir la stabilité d'une application et créer une platforme favorable à la production, il est nécessaire de prendre en considération les besoins de stockage requis pour exécuter un algorithme et les dispositions matérielles des ressources utilisées. L'utilisation des dispositifs mobiles dans un travail collaboratif est très courant aujourd'hui. Comme la plupart ont une mémoire de stockage limitée, l'espace mémoire requis par ces algorithmes peut être une contrainte cruciale.

3- La taille des messages Réduire la taille de messages échangés, peut améliorer la qualité de service (QoS). En effet, encombrer le réseau par des messages de taille importante consomme la bande passante et peut ralentir le temps de réponse du système.

4- Délai de synchronisation Dans une application répartie, le délai de synchronisation entre les différents sites peut être réduit par la topologie du réseau. Si les utilisateurs sont géographiquement éloignés les uns des autres, les données qui circulent entre eux peuvent être soumis à une latence importante. Améliorer la topologie du réseau peut réduire la latence. En effet, éviter les consensus entre les serveurs et réduire le temps de calcul au niveau des centres de données peut accélérer le temps de convergence.

Critères spécifiques à l'édition collaborative

Dans ce qui suit, nous présenterons quelques critères spécifiques à l'édition collaborative.

1- Nombre et taille de conflits Si les utilisateurs travaillent en parallèle, il est fréquent qu'ils modifient le même document dans la même zone de texte. Lors de l'intégration des modifications, si le système ne peut pas fusionner les versions des utilisateurs, un conflit est généré. Dans ce cas, l'utilisateur consacre du temps pour le corriger et la productivité de l'équipe est influencé. Si le système collaboratif génère trop de conflits de grande taille, les utilisateurs seront insatisfaits.

Dans le but de réduire le nombre et la taille des conflits, une étude sur le comportement des algorithmes durant la phase de fusion des modifications est importante. Cependant, détecter les conflits durant une session de collaboration n'est pas évident. Cette étude nécessite un mécanisme qui reproduit le résultat avant et après la correction de ces conflits.

2- Qualité du résultat La qualité du résultat nous indique le degré de satisfaction des utilisateurs vis-à-vis du résultat produit par les algorithmes de réplication.

Reproduire la session d'édition collaborative et comprendre les conflits est très important, mais est-ce que le résultat obtenu satisfait l'utilisateur ? Par exemple, dans le cas où deux utilisateurs insèrent en concurrence le même contenu à la même position, certains outils acceptent une version et annule l'autre, alors que peut-être l'intention des développeurs est d'insérer les deux versions. Cette étude est difficile puisqu'il faut comprendre ce que l'utilisateur souhaite comme résultat avant la fusion des modifications.

1.2.4 Problématique

Étudier les performances des algorithmes de réplication optimiste dans les systèmes d'édition collaborative est un besoin élémentaire. Cependant, comparer ces algorithmes et étudier leurs performances est difficile, car ils fonctionnent de manière différentes et sont influencés par des paramètres différents. Par exemple, les algorithmes OT dépendent du nombre d'opérations et des transformations, alors que les algorithmes CRDTs génèrent des identifiants de taille variable. En plus, la prédiction des performances des algorithmes dépend de plusieurs autres paramètres telles que les paramètres réseau, les caractéristiques des machines, le comportement des utilisateurs, l'architecture déployée, etc. L'influence de ces facteurs sur les algorithmes ne peut être vérifiée théoriquement et rendent la prédiction du comportement des algorithmes extrêmement difficile. L'expérimentation permet non seulement de valider les résultats obtenus dans la partie théorique mais aussi de confronter les algorithmes dans des situations réelles et d'étudier ainsi leurs performances. De plus, certains critères n'ont pas de définition théorique tels que l'intention des utilisateurs et la qualité du résultat.

Aujourd'hui, plusieurs algorithmes de réplication optimiste ont été développés et il est essentiel de poser les questions "quel est le meilleur algorithme ? Dans quelle application je peux l'utiliser ? Fusionnent-ils automatiquement et correctement les modifications ? Le résultat satisfait-il l'utilisateur ? Sont-ils applicables sur un système complètement décentralisé ?...". C'est à toutes ces questions que nous allons tenter de répondre dans cette thèse.

1.3 Contributions

Les contributions de cette thèse portent sur les aspects suivants :

1.3.1 Méthodologie

Notre première contribution est une méthodologie d'évaluation pour des algorithmes de réplication optimiste. Cette méthodologie est publiée dans [9, 72]. Notre méthodologie est basée sur un corpus, un simulateur et des mesures.

Corpus L'évaluation des performances des algorithmes de réplication optimiste nécessite un large jeu de données. Pour que le résultat de l'expérience reflète les performances des algorithmes et le comportement des utilisateurs lors d'une édition collaborative réelle, le corpus doit être réaliste et inspiré de situations réelles.

Peu de ressources sont aujourd'hui disponibles pour évaluer les algorithmes de réplication optimiste. Les différents types de données répliqués évalués jusqu'à présent, ont été évalué soit sur des traces sérialisées telles que Wikipedia ou des traces de gestion de versions telle que SVN [119, 85]. Ces traces ne contiennent pas de modifications concurrentes [34] et le résultat d'une expérimentation basée sur ces traces ne reflète guère la réalité. Notre première objectif dans la méthodologie est de mener des expériences pour récupérer un corpus réaliste.

Simulateur Pour évaluer les performances des algorithmes de réplication optimiste, nous avons besoin d'organiser des sessions d'édition collaborative sur lesquelles nous exécuterons les différents algorithmes. Cependant, organiser une session d'édition collaborative massive [4] où les utilisateurs sont géographiquement répartis et sur un système réel n'est pas facile à réaliser. De plus, évaluer les algorithmes sur une architecture distribuée rend les propriétés de contrôle et la reproductibilité difficile à garantir. Certains paramètres sont incontrôlables tels que la déconnexion des répliques et la vitesse de transmission. La simulation nous permet de dérouler une édition collaborative massive sur un modèle d'un système réel, tout en contrôlant les propriétés d'expérimentation. Notre deuxième objectif dans la méthodologie est d'implémenter un simulateur pour simuler des sessions d'édition collaborative avec des algorithmes de réplication optimiste.

Outils d'évaluation Pour répondre à notre problématique, il est important d'évaluer les différents algorithmes en utilisant les mêmes méthodes et sur les mêmes critères. De plus, les algorithmes doivent être implémentés correctement dans le même langage et suivre la même conception.

Nous mettrons donc en oeuvre notre méthodologie à travers un outil d'évaluation. Cet outil intègre les différents algorithmes de réplication, des mécanismes d'extraction de corpus synchrone et asynchrone, ainsi qu'un simulateur d'édition collaborative et garantit une évaluation pertinente. Cet outil permet de réaliser nos expériences.

1.3.2 Analyse des performances

Notre deuxième contribution consiste à mesurer et à comparer les performances des différents algorithmes de réplication sur les critères décrits dans la section 1.2.3. Ces expérimentations se basent sur des jeux de données issues des données réelles incluant la concurrence. Jusqu'à présent, aucune étude expérimentale n'a été effectuée sur un tel corpus. Les résultats de cette évaluation sont publiés dans [7, 68].

4. avec un grand nombre de participants

1.3.3 Analyse de la qualité du résultat

La troisième contribution consiste à étudier la qualité du résultat produit par les algorithmes de réplication, en estimant l'effort effectué par l'utilisateur durant une session d'édition collaborative asynchrone. Nous utilisons notre outil pour rejouer plusieurs sessions d'édition collaborative réelle. Nous observerons à travers cet outil les modifications concurrentes, nous étudierons les cas fréquents qui créent les conflits et nous identifierons quels algorithmes sont capables de les résoudre automatiquement dans un sens qui réduit les efforts des utilisateurs. Cette étude démontre formellement pour la première fois l'intérêt de certaines catégories d'algorithmes dans les systèmes asynchrones. Les résultats de ce travail sont publiés dans [8, 9]

1.3.4 Améliorations

Après avoir analysé les performances et la qualité du résultat des différents algorithmes, nous identifierons les cas d'exécutions qui réduisent les performances. Comprendre le comportement des algorithmes envers ces situations nous permettra de les améliorer. Notre quatrième contribution consiste à adapter et améliorer des algorithmes de réplication en performance et qualité du résultat. Les solutions proposées et les résultats sont publiés dans [7, 8]

1.3.5 Une nouvelle architecture

Améliorer et adapter un algorithme pour toutes les exécutions est impossible. En effet, certains algorithmes sont adaptés pour des architectures centralisées alors que d'autres sont plus adaptés pour des architectures décentralisées. Les algorithmes décentralisés utilisent des identifiants uniques ou des structures de données spécifiques pour ordonner les modifications. Une telle solution réduit la qualité de service dans le réseau et ne passe pas à l'échelle. Les solutions centralisées ne nécessitent pas d'identifiant unique, mais elles ne tolèrent pas les pannes. Pour interpréter les résultats obtenus durant l'évaluation et tirer profit des deux approches (centralisée et décentralisée), il est préférable dans certains cas de changer complètement l'architecture d'exécution.

Notre dernière contribution consiste à proposer une nouvelle architecture en combinant les deux approches OT/CRDT et à étudier les avantages possibles pour améliorer le délai de synchronisation. L'architecture proposée et les résultats préliminaires de cette proposition sont publiés dans [73].

Chapitre 2

État de l'art

Sommaire

Dans ce chapitre, nous nous intéresserons dans un premier temps à la réplication des données et aux problèmes engendrés par la réplication. Ensuite, Nous présenterons les solutions existantes. Nous discuterons des différents types de données répliqués, de leurs mécanismes de gestion de la concurrence et les types de propagation des modifications. Nous discuterons ensuite sur des travaux existants qui permettent l'évaluation des algorithmes de réplication.

2.1 Réplication des données

La réplication est un processus de sauvegarde et de duplication d'informations entre plusieurs sites [71]. C'est un concept fondamental dans la plupart des systèmes distribués. Cela permet le travail en mode déconnecté, améliore la fiabilité, la tolérance aux pannes et la disponibilité des données [93]. Un objet répliqué pourra être accédé en lecture ou en écriture à partir de plusieurs emplacements qui peuvent être répartis dans le monde entier. Dans ce qui suit, nous définirons les différents termes utilisés dans les systèmes de réplication.

Objets, répliques et sites Un *objet* est l'unité minimale dans un système de réplication. Une *réplique* est une copie de l'objet qui peut être modifiée. Elle est stockée sur un *site*. Les sites tel que les ordinateurs, les tablettes et les téléphones portables peuvent stocker des répliques de plusieurs objets.

Les mises à jour, propagation et exécution Lorsque l'utilisateur change l'état d'un objet, des modifications sont générées. Ces dernières ont un effet immédiat sur la copie locale de l'objet. Ensuite, elles sont propagées aux autres répliques. Selon le type de propagation utilisé, ces mises à jour sont propagées sous forme d'*états* ou de séquence d'*opérations*.

Détection de conflits Le travail parallèle et concurrent peut créer des conflits, si plusieurs utilisateurs modifient le même objet en même temps. Par exemple, si deux voyageurs prennent la dernière place dans un avion en même temps. Une solution simple est d'en accepter une arbitrairement et d'ignorer l'autre. Cependant, ignorer une requête n'est pas souhaitable dans nombreuses applications. Ce problème est appelé *mises à jour perdues*.

Dans le contexte des systèmes collaboratifs, un conflit est généré si les pré-conditions d'une modification ne sont pas respectées. Par exemple, dans un système de fichiers partagés, un utilisateur ne peut pas supprimer un fichier *document/doc.txt* si le répertoire *document* est supprimé en parallèle. Plusieurs systèmes intègrent les pré-conditions dans les algorithmes de réplication. Une suppression ne peut être effectuée que si l'algorithme vérifie que le répertoire et le fichier sont présents. Dans le cas contraire un conflit est généré.

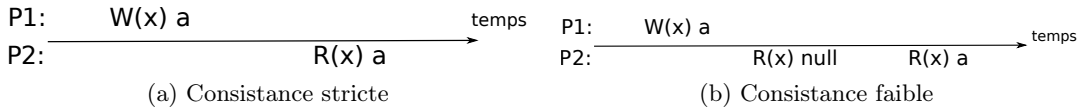
Lorsque les données répliquées sont modifiables, la consistance entre les répliques doit être assurée. La *consistance* est la capacité pour un système à refléter sur une réplique les modifications intervenues sur d'autre répliques pour la même donnée. Ainsi, plusieurs modèles de consistance ont été proposés.

2.1.1 Modèles de consistance

Le modèle de consistance est un contrat entre les processus et l'espace de stockage [106]. Il garantit une lecture correcte du résultat si les processus respectent certaines règles. Chaque modèle de consistance est définie par un ensemble des écritures dont les résultats sont visibles par des opérations de lecture [108]. On note :

1. $R_i(x)a$: le processus i lit la variable x, le résultat est a,
2. $W_i(x)a$: le processus i écrit la valeur a dans la variable x.

Plusieurs modèles de consistances ont été proposées [111, 114, 99, 108] :



(a) Consistance stricte (b) Consistance faible

FIGURE 2.1 – Consistance faible et stricte

1. Consistance stricte : le résultat d'une opération de lecture est l'effet de *toutes* les écritures effectuées. Le système garantit qu'elle retourne la valeur de la dernière mise à jour exécutée sur l'objet. Par exemple, dans la figure 2.1a, un processus écrit dans la variable x la valeur 'a'. Lorsque le processus 2 a effectué une lecture de x, il a lu la dernière valeur écrite qui est

'a'. Dans ce cas, le modèle de consistance présentée dans la figure 2.1a est la consistance stricte ;

2. Consistance faible : une opération de lecture retourne la valeur d'*un ensemble* de mises à jour exécutées sur l'objet. La consistance inéluctable ou à terme est une forme spécifique de la consistance faible. Dans ce type de consistance, les répliques peuvent observer des résultats différents temporairement. Mais, lorsque toutes les mises à jour seront propagées, le système garantit que les répliques observent le même état de l'objet. Dans la figure 2.1b, le processus 1 écrit dans la variable x la valeur 'a'. Lorsque le processus 2 a effectué une lecture de la variable x, la valeur retournée est nulle. Dans ce cas les deux processus n'observent pas le même contenu. Après une période, le processus 2 effectue une autre lecture de la variable x. Cette fois l'opération de lecture retourne la valeur 'a'. Le modèle de consistance présentée dans la figure 2.1a est la consistance faible ;

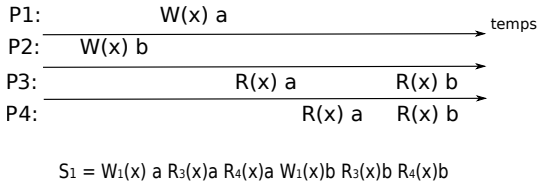
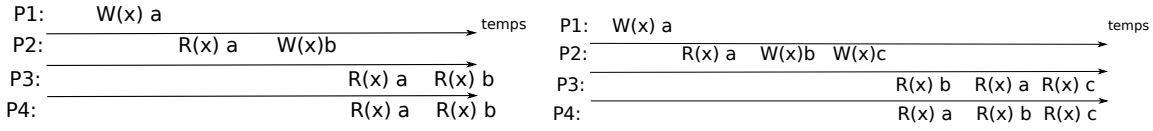


FIGURE 2.2 – Consistance séquentielle

3. Consistance séquentielle : Dans [58], Lamport a défini la consistance séquentielle comme suit :

   « *... le résultat de toute exécution est le même que si les opérations de tous les processeurs ont été exécutés dans un ordre séquentiel, et les opérations de chaque processeur apparaissent dans cette séquence et dans l'ordre défini durant leur génération.* »

   Dans la figure 2.2, le processus 2 écrit dans la variable x la valeur 'b', ensuite le processus 1 écrit dans la même variable la valeur 'a'. Si tous les processus observent la valeur 'a' avant 'b' telle que l'exécution de S_1, alors le modèle de consistance séquentielle est respecté.



(a) Consistance Causale (b) Consistance Fifo

FIGURE 2.3 – Consistance Causale et Fifo

4. Consistance causale : l'exécution des opérations doit respecter l'ordre causal [57] entre opérations. L'exécution de toutes les opérations causalement liées se déroule suivant le même ordre sur toutes les répliques, alors que les opérations concurrentes peuvent être exécutées dans un ordre différent sur les différentes répliques. Dans la figure 2.3a le processus 2 effectue l'opération $W_2(x)b$ après avoir effectué l'opération $R_2(x)$. Donc $W_2(x)b$ dépend causalement de $W_1(x)a$. L'exécution présentée dans la figure 2.3a respecte le modèle de consistance causale car l'ordre causale est respecté. En effet, tous les processus observent 'a' ensuite 'b' ;
5. Consistance FIFO : les écritures effectuées par un processus sont observées dans le même ordre par les autres clients. Par contre, les écritures faites par des processus différents

peuvent être observées dans un ordre différent, même s'ils sont causalement liés. Dans la figure 2.3b, l'exécution respecte la consistance Fifo. En effet, les processus observent le 'b' avant 'c' car $W_2(x)b$ précède $W_2(x)c$.

Selon le modèle de consistance, le système exécute des séquences d'opérations différentes. Les algorithmes de réplication doivent garantir que toutes les répliques observent le même type de résultat, et cela quel que soit le modèle de consistance utilisé. De ce fait, plusieurs techniques de réplication ont été proposées.

2.1.2 Réplication pessimiste

La réplication pessimiste suppose bien sûr l'utilisation de plusieurs copies mais donne l'illusion aux utilisateurs de l'existence d'une copie unique [12]. Les protocoles les plus connus sont basés sur le principe ROWA (Read One Write All). Les lectures peuvent être faites sur n'importe quelle copie tandis qu'une écriture doit être appliquée de manière atomique sur toutes les copies (à un instant donné un seul utilisateur peut écrire sur une copie). Ces protocoles font appel à l'utilisation des verrous dont la distribution est régie par un site central.

La réplication pessimiste est possible tant que le réseau n'est pas large, que la latence du réseau n'est pas importante et que les pannes sont maîtrisées [93].

La réplication pessimiste assure la consistance stricte. Cependant, le théorème CAP (Consistency, Availability and Partition-tolerant) [38, 39] démontre que n'importe quel système distribué ne peut garantir à la fois la consistance stricte, la disponibilité et la tolérance aux pannes. Seules deux propriétés parmi ces trois peuvent être assurées pour n'importe quelle donnée partagée :

1. Consistance : chaque serveur retourne une réponse *correcte* à chaque requête. La réponse correcte dépend du service désiré,
2. Disponibilité : chaque requête reçue par une réplique connectée doit retourner une réponse. Évidemment, une réponse rapide est mieux qu'une réponse lente. Dans la plupart des systèmes réels, une réponse lente est aussi mauvaise qu'une réponse jamais reçue [39],
3. Tolérance aux pannes : Le système continue de fonctionner même si la communication entre les serveurs n'est pas fiable ou bien si les serveurs sont divisés en plusieurs groupes et ne peuvent pas communiquer entre eux. Une communication non fiable peut causer la perte de messages ou un délai long pour le recevoir.

Comme les partitions ne sont pas contrôlables et que l'accessibilité est requise par les systèmes et les utilisateurs, un choix consiste à diminuer le niveau de consistance mais à assurer une disponibilité élevée des données et à tolérer les pannes. Ainsi, de nombreux systèmes distribués mettent en oeuvre la réplication optimiste [93].

2.1.3 Réplication optimiste

La réplication optimiste est défini dans [93] comme un ensemble de techniques développées dans le but de partager efficacement les données dans un large réseau ou dans un environnement mobile. La caractéristique principale qui différencie les algorithmes de réplication optimiste de la réplication pessimiste est la façon de contrôler la concurrence. Contrairement à la réplication pessimiste, la réplication optimiste n'impose aucune contrainte entre les sites lors de l'exécution des modifications.

Dans la réplication optimiste, plusieurs répliques de l'objet partagé sont stockées sur des sites. Ces répliques peuvent être modifiées librement et à tout moment. Lorsqu'un site reçoit

une modification, il tente de l'exécuter dans sa réplique locale. Face aux modifications parallèles, deux sites peuvent donc recevoir et exécuter les mêmes modifications dans un ordre différent. La réplication optimiste laisse les répliques diverger un moment, mais elles finissent par converger vers la même valeur lorsque toutes les mises jour sont reçues. Ce type de convergence est appelée convergence à terme ou convergence inéluctable [93]. Ce type de consistance est une forme de consistance faible.

Généralement, dans les systèmes d'édition collaborative, chaque site possède une réplique de l'objet partagé sur laquelle il effectue des modifications. Par exemple dans la figure 2.4, quatre répliques partagent initialement le même document "ab". La réplique 1 supprime le caractère 'a' qui se trouve à la position 0. En concurrence, la réplique 2 ajoute 'x' à la deuxième position. Les deux modifications sont concurrentes et il n'existe pas un ordre entre eux. Donc, les modifications peuvent être reçues dans un ordre différent sur les différentes répliques. Supposons que la réplique 3 reçoit la suppression et produit le document "b". Et la réplique 4 reçoit l'insertion de 'x' et produit le document "abx". Á ce moment, les répliques 3 et 4 divergent car ils n'observent pas le même document. Cependant, lorsque toutes les modifications sont reçues, les répliques convergent vers le même document "bx".

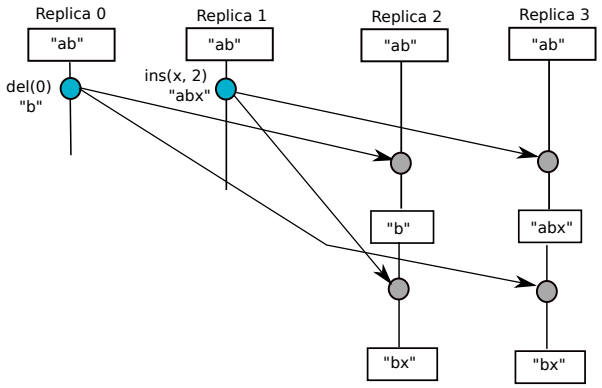


FIGURE 2.4 – Convergence à terme

Pour assurer la convergence à terme des répliques, les approches transformées opérationnelles (OT) [34, 105, 113] et CRDT [99, 82, 85, 90, 119] ont été introduites. Ces approches garantissent que les modifications concurrentes produisent le même effet sur tous les sites.

Nous détaillerons ces deux approches dans la section 2.3

Les approches optimistes offrent plusieurs avantages par rapport aux approches pessimistes :

1. Elles améliorent la fiabilité et la disponibilité des données : plusieurs sites stockent des répliques de l'objet partagé. En cas de déconnexion, les approches optimistes laissent la possibilité à l'utilisateur sur un site de continuer les modifications. Une fois connecté il pourra propager ces modifications ;

2. Elles sont plus flexibles : les approches optimistes permettent différentes techniques de propagation de modifications telle que la propagation épidémique [36]. Cette technique permet de diffuser les modifications de manière fiable à toutes les répliques même si le graphe de communication est inconnu ou variable ;

3. Elles passent à l'échelle : la plupart des approches optimistes n'utilisent pas de contraintes (verrous, coordination, etc) entre les sites lors de l'exécution des opérations [93].

Quel que soit le mode de réplication (pessimiste ou optimiste), les mises à jour effectuées sur un objet sont propagées entre les répliques sous forme d'état ou de séquence d'opérations. Dans ce qui suit, nous discuterons de chaque type de propagation.

2.2 Propagation des états : state-based

Certains systèmes de partage de données traitent les modifications comme un ensemble d'états. Dans ces systèmes, chaque site possède une copie locale des objets partagés. Les modifications sont entièrement exécutées dans la réplique locale, et propagées ensuite entre les répliques, comme illustré dans la figure 2.5. Lorsqu'une réplique reçoit une mise à jour sous forme d'état, la réplique compare l'état reçu avec l'état local pour extraire les mises à jour nécessaires. Ensuite, des pré-conditions sont vérifiées avant de fusionner les deux états. Cette vérification consiste à détecter les conflits potentiels.

Dans ces systèmes, les répliquent observent seulement l'état de la dernière version de l'objet après une mise à jour. Les systèmes de fichiers distribués tels que Ficus [44] et Coda [55] sont des exemples des systèmes state-based. Chaque système est représenté par un arbre où les noeuds sont des répertoires et les feuilles sont des fichiers. Pour une question de performance, il n'est pas souhaitable de propager les mises à jour en temps réel après chaque modification. Les utilisateurs modifient l'état du système en local et le système propage ensuite cet état. Avant de fusionner un état reçu et un état local, l'algorithme de réplication vérifie des pré-conditions. Par exemple, un utilisateur ne peut pas ajouter un fichier sous un répertoire supprimé en parallèle, ou deux personnes ne peuvent pas créer deux fichiers avec le même nom sous le même répertoire, etc. Si une pré-condition n'est pas respectée, le système annule la fusion, génère un conflit et retourne le résultat à l'utilisateur pour le résoudre. Si aucun conflit n'est détecté, le système fusionne les deux états.

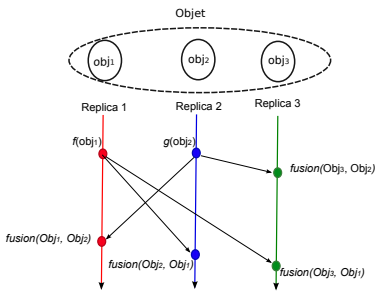


FIGURE 2.5 – Propagation par état

2.2.1 Présentation des conflits

Les outils de fusion utilisés dans les systèmes collaboratifs basés sur des états produisent souvent des conflits lors de la fusion automatique des modifications. Pour cela, les systèmes collaboratifs state-based intègrent un mécanisme dit *de conscience de groupe* [10], qui permet non seulement de visualiser les activités des autres utilisateurs mais aussi de suivre les modifications effectuées par les autres utilisateurs sur l'objet partagé, au cours du temps. Ce mécanisme collecte des informations pour répondre à des questions classiques telles que : où les modifications ont-elles été faites ? qui a fait les modifications ? dans quelle partie de l'objet ?, etc. Répondre à ces

questions aide les utilisateurs à trouver le conflit facilement dans l'objet partagé et le résoudre rapidement.

Le mécanisme de conscience du groupe [10] peut être intégré indépendamment du mécanisme interne de la fusion. Dans [94, 46], lorsque le conflit est généré, le système intègre les deux modifications conflictuelles contenant un délimiteur de début "`>>>>>>>>`", le contenu de la première modification, un délimiteur central "`========`", le contenu de la deuxième modification et enfin un délimiteur de fin "`<<<<<<<<`". Cette représentation permet de localiser facilement les modifications concurrentes dans le document. Le mode d'affichage de cette information est différent d'un outil à un autre. Par exemple, l'outil So6 [75] ajoute les marqueurs au dessus du résultat de la fusion, alors que diff3 [41] ajoute les marqueurs à la position où le conflit est généré. D'autres outils comme Palantir [95] ou Crystal [15], informe les développeurs dans une nouvelle fenêtre.

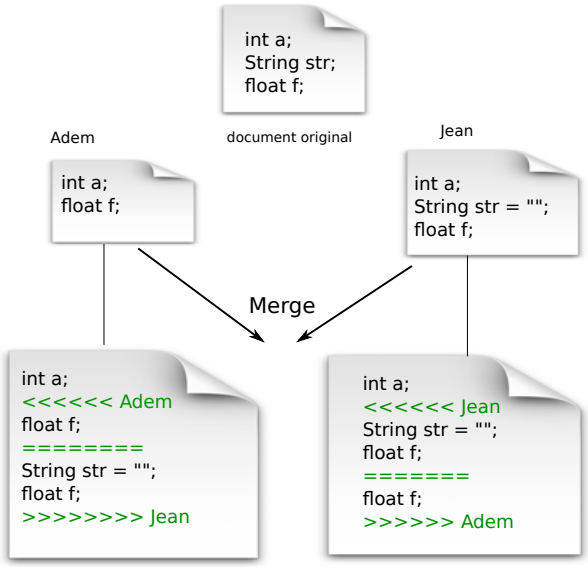


FIGURE 2.6 – Fusion de modification concurrente

La figure 2.6 donne un exemple d'un document textuel en conflit produit par deux utilisateurs Adem et Jean. Initialement les deux utilisateurs partagent le même document. Jean a modifié la *deuxième* ligne en insérant la chaîne de caractères `String str="0";`, en concurrence Adem a supprimé la même ligne. Lors de la procédure de fusion, le système détecte que Adem et Jean ont modifié le même document dans la même zone, et donc un conflit est généré. Ainsi, le système retourne le résultat avec des informations complémentaires à l'utilisateur qui a effectué la fusion pour l'aider à mieux comprendre le conflit et le résoudre. Les systèmes de gestion de versions peuvent être configurés pour qu'ils intègrent différents outils de fusion tels que diff3 [54], emerge ou gvimdiff.

2.2.2 Systèmes actuels et détection de conflits

Les algorithmes state-based sont largement utilisés dans les systèmes de développement de logiciels et dans les systèmes de fichier distribués. Parmi les critères importants pour un système state-based performant il y a la réduction du nombre et de la taille des conflits. Les conflits les plus importants dans un projet sont ceux qui ont une plus grande probabilité de se reproduire et ceux qui peuvent causer un grand impact sur le projet.

Dans ce qui suit, nous présentons quelques systèmes state-based qui s'intéressent à la gestion des conflits, ainsi que leurs limites lors de la fusion des modifications.

Gestion de version

La gestion de versions consiste à maintenir l'ensemble des versions d'un ou plusieurs fichiers. Les logiciels de gestion de version tels que CVS [21], Git [110] et SVN[24] sont utilisés essentiellement dans le domaine du développement de logiciels. Durant la collaboration, plusieurs problèmes peuvent intervenir et peuvent conduire l'utilisateur à quitter la collaboration si le système ne fusionne pas correctement les modifications. Pour éviter toute confusion, assurer la cohérence des données et augmenter la productivité de l'équipe, les systèmes de gestion de versions actuelles contrôlent l'accès en limitant soit le nombre de participants ou la manière d'envoyer/fusionner les modifications.

Orian

Orian [86] est une approche de contrôle d'accès concurrent pour les systèmes de gestion de versions. Elle vise à optimiser le temps de développement et les efforts des développeurs en réduisant le nombre de conflits. L'implémentation de Orian utilise Odyssey-VCS [80] comme système de gestion de versions et EvolTrack [27] pour la visualisation des versions concurrentes et les conflits.

Orian calcule l'effort des utilisateurs en calculant une métrique basée sur le temps de modification et le nombre d'opérations effectuées pour résoudre les conflits. Cette approche nécessite quelques informations qui ne sont pas disponibles dans certains systèmes de gestion de versions tels que le moment de récupération des fichiers au sein du dépôt ainsi que la version initiale utilisée. De plus, le niveau de concurrence ne prend pas en considération un facteur important qui est le nombre de développeurs qui travaillent en concurrence [86].

Palantir

Palantir [95] est un outil de détection de conflits dans les systèmes de gestion de configuration logicielle. Il vise à réduire le nombre de conflits en analysant automatiquement et en arrière plan les modifications concurrentes, avant que les utilisateurs envoient leurs modifications.

Lorsque des utilisateurs modifient le même document dans la même partie en parallèle, Palantir indique aux utilisateurs qu'il y a une possibilité d'avoir un conflit dans cette partie. Cela permet aux développeurs d'être prudent, de surveiller les conflits et de les résoudre de manière préventive avant qu'ils ne deviennent compliqués. Plus tôt le conflit est détecté et plus tôt les développeurs seront en mesure de résoudre le conflit facilement et d'éviter un travail inutile.

En cas de conflit, une nouvelle fenêtre est ouverte pour aider l'utilisateur à détecter le conflit. Le but de cette solution est de réduire le nombre de conflits et de réduire l'effort des utilisateurs puisque le conflit en est encore dans son début.

Cependant, pour que l'utilisateur observe les conflits avant la fusion totale des modifications, Palantir fusionne les modifications en arrière plan. Il est limité aux fichiers Java.

Cassandra

Comme Palantir, Cassandra [53] est un outil de détection de conflits dans les systèmes de développement logiciel. Il détecte et identifie les conflits le plus tôt possible tant qu'ils sont

petits. Palantir est un outil proactif pour l'identification des conflits mais il reste réactif puisque la résolution du conflit est effectuée seulement lorsque le conflit est déjà généré.

Cassandra utilise une méthode de planification des opérations qui vise à minimiser les conflits en recommandant un ordre entre les opérations dépendantes ou les opérations effectuées sur le même fichier partagé.

Durant la collaboration, Cassandra capture des informations sur les changements effectués sur les fichiers. L'utilisateur possède des informations supplémentaires sur le déroulement de la collaboration telles que le nombre de fusions conflictuelles, le nombre de compilations et tests échoués, le nombre de conflits évités ainsi que le temps d'exécution.

Comme dans Palantir, Cassandra fusionne les modifications en arrière plan pour détecter les opérations dépendantes et les opérations effectuées sur le même fichier.

2.3 Propagation par opération : operation-based

Contrairement à la propagation par état, les approches basées sur les opérations transmettent seulement les modifications effectuées sur l'objet local. Pour chaque modification, un message portant cette information est envoyé à l'ensemble des répliques connectées. Cette approche est souhaitable pour les systèmes collaboratifs temps réels [100, 7].

Dans le contexte d'édition collaborative, une modification effectuée dans le document est transformée en un ensemble d'opérations. Cette dernière est exécutée localement et diffusée ensuite aux autres répliques. Quand une réplique reçoit l'opération distante, elle la ré-exécute localement. En absence d'une horloge globale et en présence des opérations concurrentes, il est difficile de définir un ordre entre ces opérations. De ce fait, les répliques peuvent exécuter les opérations dans un ordre différent, ce qui peut entraîner une divergence entre les répliques. Pour garantir la convergence du document entre les différentes répliques, les algorithmes OT [34, 105, 113] et CRDT [99, 82, 85, 90, 119] ont été proposés.

2.3.1 Transformées Opérationnelles (OT)

Dans cette approche, chaque site possède une copie locale des objets partagés. Lorsqu'un utilisateur génère une opération, cette dernière est traitée en cinq étapes :

1. exécution locale,
2. envoi de l'opération aux autres sites,
3. réception de cette opération par les sites,
4. transformation avec les opérations concurrentes,
5. exécution de l'opération sur la copie locale,

La transformation de l'opération distante consiste à modifier ses paramètres pour prendre en considération l'effet des opérations concurrentes. L'approche des transformées opérationnelles met en jeu deux composants majeurs :

1. L'algorithme d'intégration : responsable de la réception, de la diffusion et de l'exécution des opérations. Si nécessaire, il appelle les fonctions de transformation. Parmi les algorithmes d'intégration on trouve notamment, SOCT2, SOCT3, SOCT4 [102, 113] et GOTO [78],
2. les fonctions de transformation : les fonctions de transformation permettent de transformer les opérations reçues en tenant compte des opérations concurrentes.

Par exemple, dans la figure 2.7, deux répliques partagent initialement le même document `Ubnttu`. Les utilisateurs collaborent dans le but de créer un document correct `Ubuntu`. Pour cela, l'utilisateur sur la réplique 0 insère `'u'` à la position 2 (soit op_0) et crée le document `Ubunttu`. Au même moment, l'utilisateur de la réplique 1 supprime le caractère à la position 3 (soit op_1) et crée le document `Ubntu`. À la réception d'op_1 par la réplique 0, l'opération est transformée par rapport à l'opération concurrente op_0. En effet, pour supprimer à la bonne position, la suppression n'est plus à la position 3 mais à la position 4 puisqu'il y a eu une insertion concurrente. Tandis que la réplique 1 ne change pas la position de l'opération op_0 puisque la position de l'insertion est avant la position de la suppression. Finalement, les deux répliques convergent et créent un document correct `Ubuntu`.

Il est donc nécessaire de transformer les opérations reçues avec les opérations locales concurrentes avant de les exécuter.

Pour garantir la convergence, les fonctions de transformation doivent respecter deux contraintes TP1 et TP2 [89].

1. TP1 : cette condition définit une *identité d'états* [47]. Elle vérifie si l'état produit en exécutant op_1 avant op_2 est le même que celui résultant de l'exécution de op_2 avant op_1. Pour chaque paire d'opérations concurrentes op_1 et op_2, la fonction de transformation T satisfait TP1 si et seulement si :

$$(S \odot op_1) \odot T(op_2,\ op_1) \ =\ (S \odot op_2) \odot T(op_1,\ op_2)$$

   où S est un état, op_n est la séquence d'opérations définies sur l'objet répliqué ; et $\odot$ l'opérateur d'exécution d'une opération sur un état.

2. TP2 : cette condition définit une *identité d'opérations* [47]. Elle assure que la transformation d'une opération par rapport à une séquence d'opérations ne dépend pas de l'ordre de transformation des opérations sur la même séquence.

$$T(op_3,\ op_1\ o\ T(op_2,\ op_1)) \ =\ T(op_3,\ op_2\ o\ T(op_1,\ op_2))$$

   où op_n est la séquence d'opérations définies sur l'objet répliqué et $op_{n-1}\ o\ op_n$ est la séquence d'opérations contenant op_{n-1} suivit par op_n.

L'inconvénient de cette approche est le nombre de transformations effectuées pour garantir la convergence et aussi la difficulté de proposer des fonctions de transformation qui vérifient TP1 et TP2 [83].

Il existe plusieurs applications commercialisées qui adaptent les algorithmes OT. Par exemple, l'application d'édition collaborative Google Docs [1] adapte un algorithme OT centralisé appelé Jupiter [78].

Les approches OT centralisées telles que SOCT4 [113], So6 [75] et Jupiter [78] impose un ordre total entre les opérations par un séquenceur centralisé. En plus des inconvénients de la centralisation, cette solution génère une forte latence si le nombre de réplique est important.

De l'autre côté, les approches OT décentralisées tels que SOCT2 [102], GOTO [104] et dOPT [34] utilisent :

1. un mécanisme pour garder toutes les opérations exécutées sur l'objet,
2. un vecteur d'horloge [87] pour le respect de la causalité entre les opérations [105],
3. vérification de la condition TP2 [102]

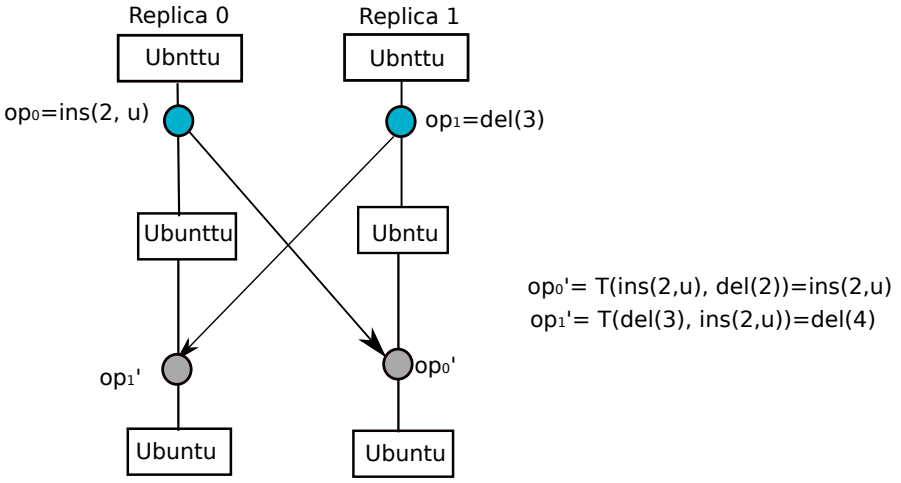


FIGURE 2.7 – Transformées opérationnelles

Malheureusement, cette solution occupe une mémoire importante, ne passe pas à l'échelle en nombre de répliques ou en nombre d'opérations et augmente la complexité en temps exécution.

À titre d'exemple, nous présentons dans cette thèse SOCT2 pour les algorithmes décentralisés et SOCT4 pour les algorithmes centralisés.

SOCT2

SOCT2 [102] est un algorithme OT. Il utilise les propriétés TP1 et TP2 pour garantir la convergence des répliques. Pour respecter la causalité, l'algorithme utilise les vecteurs d'horloge. Lors de la génération d'une opération locale, cette dernière est exécutée immédiatement sur le site et envoyée à toutes les autres répliques y compris lui-même. Cela a pour effet d'ajouter l'opération dans l'historique du site. L'opération est diffusée aux autres répliques sous forme d'un vecteur avec trois paramètres $< op, S_{op}, SV_{op} >$ où op représente le type d'opération, S_{op} est l'identifiant du site qui a généré l'opération et SV_{op} est le vecteur d'horloge de op. On notera que toutes les opérations locales précèdent causalement *op*.

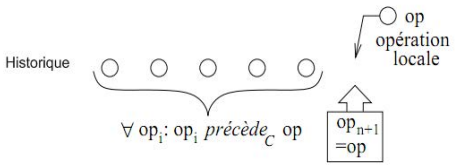


FIGURE 2.8 – Exécution d'une opération locale

Livraison des opérations Lors de la réception d'une opération locale ou distante par le site S_{op}, la fonction livraison causale est exécutée ; elle garantit que si op_1 précède causalement op_2, alors op_1 sera délivré avant op_2 et garantit aussi la livraison des opérations à la fonction d'intégration.

L'intégration Cette procédure réalise l'exécution de l'opération distante (op). Le problème qui se pose est comment détecter les opérations concurrentes à l'opération délivrée ? Comme signalé précédemment l'historique de chaque site contient toutes les opérations qui précèdent causalement une opération locale (figure 2.8). Pour prendre en compte le problème de convergence et de préservation de l'intention, SOCT2 utilise un mécanisme de transposition en avant afin d'obtenir une opération op_{n+1} exécutable sur l'état courant tout en tenant compte des opérations concurrentes de l'historique.

Afin de réduire le temps de recherche des opérations et l'insertion des éléments, la taille de l'historique est compressée en utilisant le mécanisme de ramasse de miette ou garbage collector [4]. Cependant, puisque le déclenchement de ce mécanisme dépend des données en entrée, si une réplique ne reçoit pas d'opérations d'une autre réplique un moment, le ramasseur de miette ne peut rien purger pendant ce temps.

Dans [81], il a été démontré que les fonctions de transformation proposées jusque là ne garantissaient pas la convergence des documents. Les seules fonctions de transformation existantes qui répondent aux conditions $TP1$ et $TP2$ sont celles proposées dans l'approche TTF (Fonctions Tombstone de transformation) [83].

TTF

Les algorithmes reposant sur le modèle des transformés opérationnels entraînent très souvent une divergence des copies. En effet, la satisfaction de la condition TP2 n'est pas toujours réalisable. Un exemple de divergence est illustré dans la figure 2.9.

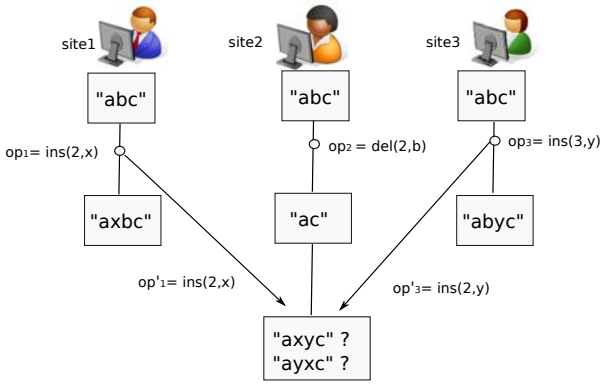


FIGURE 2.9 – Problème de divergence

En appliquant les fonctions de transformations décrites précédemment, lorsque $op_1 = ins(2, x)$ est reçue sur le site 2, elle est transformée en op_1'. Cette transformation ne change pas la position d'insertion de op1, et retourne comme résultat l'opération $op_1 = ins(2, x)$. Maintenant, lorsque $op_3 = ins(3, y)$ est reçue sur le site 2, elle est transformée en op$_3$'. Sa position d'insertion est diminuée puisque le site 2 a exécuté une opération $del(2, b)$ en concurrence et donc op_3 devient $op'_3 = ins(2, y)$. Dans ce cas, le site 2 reçoit deux opérations d'insertion à la même position ce qui entraîne une divergence des copies. Plusieurs approches ont été proposées pour régler ce problème dont Suleiman et al. [102] ou Li et al. [61]. Malheureusement, toutes ces approches ne parviennent pas à ordonner correctement "x " et "y" dans certains cas. Les seules fonctions de transformations qui existent et qui vérifient TP1 et TP2 sont celles proposées par l'approche TTF.

L'idée de base de l'approche TTF [83] consiste à garder la position pour chaque caractère supprimé et de le marquer comme invisible. Les caractères ne sont pas supprimés physiquement mais seulement cachés à l'œil de l'utilisateur.

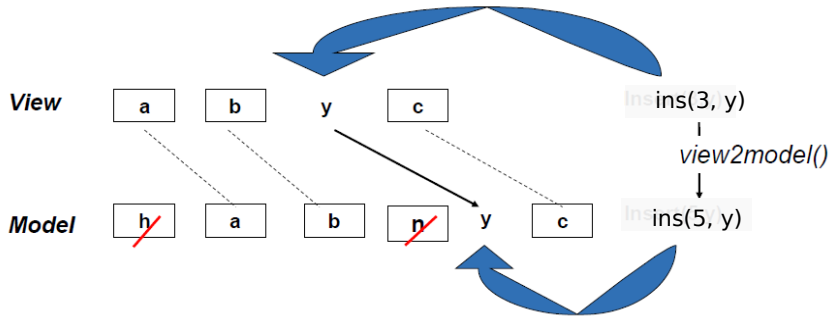


FIGURE 2.10 – modèle dans l'approche TTF.

Dans la figure 2.10, un utilisateur souhaite insérer "y" dans la position 3 entre "b" et "c", or, dans le modèle où tous les caractères sont enregistrés, la position 3 correspond à la position 5, puisque les deux caractères supprimés ("h" et "n") existaient auparavant sont toujours présents dans le modèle, mais ne sont pas visibles à l'utilisateur. En tenant compte de ces deux caractères invisibles, l'opération $ins(3, y)$ est transformée en $ins(5, y)$ et diffusée aux autres sites.

SOCT4

L'algorithme SOCT4 [113] utilise un séquenceur central pour assurer un ordre global entre les opérations. Contrairement aux algorithmes précédents, pour tenir compte des opérations concurrentes, SOCT4 effectue la transposition en avant des opérations lors de la génération de l'opération plutôt qu'à la réception.

Lors de la génération d'une opération, cette dernière est exécutée immédiatement sur le site émetteur pour satisfaire les contraintes de temps réel. Par contre, l'opération n'est diffusée qu'après avoir reçu un estampille et que toutes les opérations qui la précèdent aient été reçues par le site émetteur. En outre, l'opération n'est diffusée qu'après avoir tenu compte des opérations concurrentes par la transposition en avant.

À la réception d'une opération distante, le site récepteur détermine s'il y a des opérations locales en concurrence. Dans ce cas, SOCT4 transforme l'opération reçue pour tenir compte de l'effet des opérations locales, dans le cas contraire, l'opération est intégrée directement.

SOCT4 présente plusieurs avantages importants :

1. Pas besoin de vecteur d'horloge pour ordonner les opérations,
2. L'opération distante peut être rangée telle qu'elle dans l'histoire, sans avoir besoin d'être transposé en arrière,
3. L'histoire peut être purgée une fois que toutes les opérations locales qui précédent l'opération reçue aient été exécuté.

2.3.2 Les types de données répliqués commutatifs (CRDT)

Récemment, une nouvelle approche dite CRDT [99, 82, 85, 90, 119] (**C**ommutative **R**eplicated **D**ata **T**ypes) a été proposée comme une alternative de l'approche OT. Contrairement aux ap-

proches OT, les CRDTs ne nécessitent aucune transformation. En effet, les CRDTs ont été conçus pour que l'exécution des opérations concurrentes commutent.

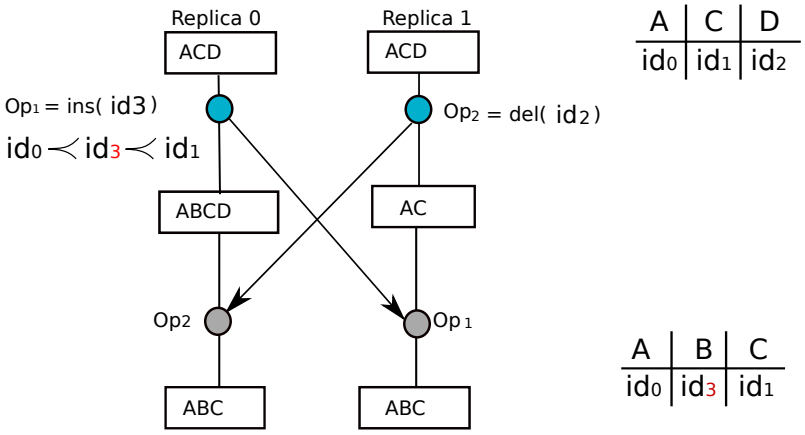


FIGURE 2.11 – Commutative Replicated Data Types

Dans le cas d'édition collaborative, cette approche identifie chaque élément de l'objet répliqué par un identifiant unique durant toute la session. Un ordre lexicographique est assuré entre les identifiants. Si le système assure la causalité, les types de données CRDT assurent la convergence [98]. La figure 2.11, présente un scénario d'exécution d'un CRDT, durant une édition collaborative d'un document textuel. Comme dans l'exemple précédent (figure 2.7), deux utilisateurs partagent initialement le même document `Ubnttu` et collaborent dans le but de créer le même document `Ubuntu`. Contrairement à OT, les CRDT identifient chaque caractère par un ID unique. Lorsque l'utilisateur insère le caractère 'u' à la position 3, il lui associe un identifiant unique id_6 entre id_1 et id_2. L'utilisateur de la réplique 1 supprime le caractère 't' avec id_4. Lorsque la réplique 0 reçoit l'opération op_1, le CRDT cherche le id_4 dans le document et le supprime. À la réception d'op_0 par la réplique 1, le CRDT cherche la position adéquate à id_6 et l'insère entre id_1 et id_2. À la fin, les deux utilisateurs créent le même document `Ubuntu`.

2.3.3 Les types de données répliqués

Les protocoles qui permettent que toutes les répliques convergent vers le même résultat sont encapsulées dans des *types de données répliqués* [3, 17, 25, 28, 100, 90]. Il existe plusieurs types de données répliqués tels que les objets, compteurs, ensembles et listes, avec différentes stratégies pour assurer la convergence des états de l'objet partagé. Une stratégie simple est d'établir un ordre total entre toutes les opérations et de donner une priorité à la dernière écriture. Certains types de données laissent l'utilisateur intervenir pour résoudre les conflits, tandis que d'autres le font automatiquement.

Les types de données répliqués doivent garantir non seulement la cohérence de l'état de l'objet, mais aussi de détecter et résoudre les opérations conflictuelles, ce qui rend leur implémentation non évidente. Dans ce qui suit, nous discutons sur différents types de données répliqués existants.

2.3.4 Les ensembles (Set)

Les ensembles constituent la base des structures de données. En effet, les graphes, les arbres et les maps sont basés sur les ensembles. Une opération d'*ajout* consiste à ajouter l'élément dans l'ensemble si ce dernier n'existe pas. L'opération de *suppression*, supprime l'élément de

l'ensemble. Dans [99], les auteurs ont proposé une variété de types de données répliqués pour les ensembles, répartie en deux groupes : state-based qui propagent les modifications par des états et operation-based qui propagent les modifications par des opérations. Dans ce rapport, on s'intéresse seulement aux operation-based. Dans ce qui suit, nous présentons quelques uns.

LWW-Set

Op-based LWW-Set un élément est un couple $< x, ts >$ ou x est le contenu et ts est une horloge logique associée à chaque élément. Chaque réplique possède deux ensembles d'éléments A et R. Une horloge logique pour ordonner les éléments est partagée entre les deux ensembles. Lorsqu'un utilisateur ajoute un élément, l'horloge correspondante est incrémentée et l'élément est inséré dans l'ensemble A. Lors d'une suppression, l'horloge correspondante à l'élément est incrémenté et l'élément est inséré dans l'ensemble R et supprimé de A.

À la réception d'une opération distante, l'horloge de l'élément local est comparée avec l'horloge de l'élément reçu. L'opération la plus récente est exécutée. Le contenu de l'objet est l'ensemble des éléments dans A.

Counter-Set

Op-based Counter-Set Un élément est un couple $< a, d >$ ou a est le contenu et d est un compteur associé à chaque élément. Initialement à 0, le compteur s'incrémente à chaque ajout, et se décrémente à chaque suppression. Un élément est visible seulement si son compteur est supérieur à 0. Cependant, un ajout après plusieurs suppressions, peut ne pas être visible à l'utilisateur même si les répliques convergent à terme. Pour certaines applications, le fait d'ajouter et de ne pas observer l'élément est une erreur sémantique, alors que d'autre le considère comme un bug.

Une solution proposée consiste à incrémenter le compteur négatif d par $|d| + 1$ en cas d'ajout, et d'envoyer la valeur -d en cas de suppression.

OR-Set

Op-based OR-Set Chaque élément est représenté par un couple $< a, Stag >$ où a est le contenu et $Stag$ est un ensemble d'identifiant unique associé à l'élément a. Lors d'un ajout, un nouvel identifiant est généré et l'élément est inséré dans l'ensemble. Lors d'une suppression, l'élément ainsi que tout les $tags$ observés en local sont supprimés. Même si dans l'ensemble, un élément peut avoir plusieurs $tags$, l'utilisateur n'en observe qu'un seul.

À la réception d'une opération distante. Si l'opération est un ajout, l'élément est ajouté directement dans l'ensemble. Dans le cas contraire, l'élément est supprimé ainsi que tout les tag observé en local lors de la suppression.

OR-Set optimisé

Dans [13], les auteurs ont optimisé l'algorithme traditionnel OR-Set en minimisant les métadonnées exigées. Dans $OptORSet$, chaque réplique maintient un vecteur qui indique les n identifiants générées successivement observés dans les autres répliques.

Lorsqu'une réplique génère une opération, elle incrémente son compteur local. Toutefois, lorsqu'un $ajout$ est livré, l'élément doit avoir un effet seulement si l'opération n'a pas été déjà délivrée. Dans le cas contraire, il met à jour le compteur de la réplique distante et supprime les

tags du même élément délivré précédemment par la même réplique. *OptORSet* réduit la taille de mémoire occupée en ne gardant que le dernier *tag* de l'élément par réplique.

Ces algorithmes permettent la résolution de certains types de conflit dans les structures de données de base telles que les arbres, les graphes, les maps, etc. Dans ce qui suit, nous présentons les différents algorithmes proposées pour les types de données *document linéaire*.

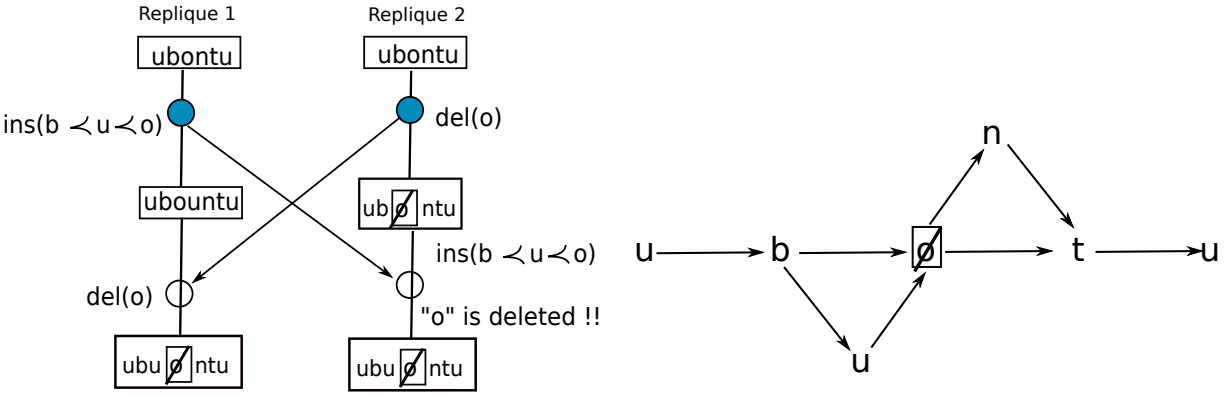
2.3.5 Document linéaire

Il existe plusieurs types de données répliqués dédiés à l'édition collaborative pour un document textuel linéaire. Dans ce qui suit, nous présentons plusieurs algorithmes.

WOOT

WOOT [82] est le premier algorithme de type CRDT. Les opérations utilisées par cet algorithme sont : *insertion* et la *suppression* des éléments d'une structure linéaire. L'idée de cette approche est la suivante : au lieu de recalculer les relations d'ordre lors de l'intégration comme dans OT, WOOT diffuse les relations d'ordre (l'élément précédent et suivant) avec les opérations.

Un identifiant unique est attribué à chaque élément du document. Si les éléments n'ont pas une relation de précédence, l'algorithme les ordonne selon la relation $\prec_{id}$. L'insertion est définie par l'identifiant de l'élément, le contenu, l'identifiant précédent et l'identifiant suivant.



(a) Insertion/suppression dans Woot

(b) Diagramme de Hasse

FIGURE 2.12 – Scénario d'exemple dans WOOT

Dans la figure 2.12a, deux utilisateurs partagent initialement le même document "ubontu". Ils collaborent dans le but de créer un document avec "ubuntu" comme contenu. L'utilisateur 1 insère 'u' à la position 3, alors que l'utilisateur 2 supprime en concurrence le caractère 'o' qui se trouve aussi à la position 3. Afin de préserver l'intention de l'utilisateur sur toutes les répliques, le 'u' doit être inséré entre 'b' et 'o'. Cette approche exécute insertion(2,'u') et suppression(o) localement mais diffuse une insertion de ['b'$\prec$'u' $\prec$ 'o'] et une suppression identifiée par [id$_o$].

La question qui se pose dans cet algorithme est comment intégrer une insertion si l'élément suivant ou précédent est supprimé ? Pour régler ce problème, on remplace l'élément supprimé par une pierre tombale possédant le même identifiant. En réalité, le caractère 'o' n'est pas supprimé mais seulement invisible à l'utilisateur.

Lors de la génération des opérations de nouvelles relations sont créées. Ces relations représentent un ordre partiel. En effet, Les opérations sont intégrées dans n'importe quel ordre sur un

site et pour obtenir de la convergence, tous les sites doivent observer le même ordre total. De ce fait un document WOOT est représenté par un diagramme de Hasse. La figure 2.12b présente un diagramme de Hasse pour l'exemple présenté dans la figure 2.12a.

L'algorithme WOOT assure la convergence des répliques et préserve les intentions sans utilisation de vecteurs d'horloges, ni site central, ni ordre global. Par contre, les éléments ne sont jamais supprimés physiquement, et par conséquent, les performances de l'algorithme se dégradent avec le temps.

WOOTO

WOOTO [116] est une optimisation de WOOT, il introduit la notion de degré pour comparer les éléments non ordonnés. L'utilisation de la notion de degré pour la construction du document réduit l'ensemble de recherche. Par conséquent, WOOTO nécessite moins de comparaison avec les caractères suivants et précédents, ce qui réduit la complexité spatiale et la complexité temporelle de WOOT.

Replicated Growable Array (RGA)

Replicated Growable Array [90] est un CRDT qui supporte l'insertion, la suppression des opérations ainsi qu'une opération de mise à jour. Il maintient une liste chaînée d'éléments, avec laquelle les opérations locales trouvent leurs éléments cibles en utilisant des index entiers. Les opérations distantes récupèrent leurs cibles via une table de hachage en utilisant un index unique.

RGA utilise aussi un identifiant unique appelé *s4vector* composé de quatre nombres entiers (ssn, sid, sum, seq) où ssn : est le numéro de la session globale, sid : un numéro unique qui identifie le site, sum : la somme du vecteur d'horloge, seq : réservé pour les pierres tombales.

Par exemple, on suppose le vecteur d'horloge du site1 est [1, 2, 3] dans la session 3. Dans ce cas *s4vector* du site 1 est <3, 1, 6, 1>. L'ordre entre deux *s4vector* s1 et s2 est définie comme suit : s1 précéde s2 si est seulement si : $(ssn1 > ssn2)$ ou $(ssn1 = ssn2$ et $sum1 < sum2)$ ou $(ssn1 = ssn2$ et $sum1 = sum2$ et $sid1 > sid2)$. Le vecteur *s4vector* est délivré à chaque opération, il peut être utilisé comme index pour trouver une cible dans la table de hachage ; il est également utilisé pour résoudre les conflits entre les insertions concurrentes à la même position.

Dans cet algorithme, les éléments sont représentés par un nœud. Un nœud est un quadruplet de (obj, S_k/S_p, Lien, suivant) tel que présenté dans la figure 2.13.

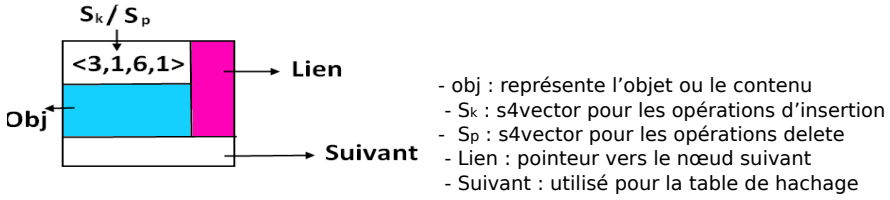


FIGURE 2.13 – S4vector

Le fonctionnement global de RGA est illustré dans la figure suivante : Par exemple le site2 génère l'opération insert(3,O_x) avec le vecteur d'horloge [3,1,2]. Un nouveau nœud est créé localement avec les paramètres suivants : $obj = O_x$, *s4vector* = <1,2,6,2>, S_k et S_p. Ce noeud est placé dans la table de hachage en utilisant le *s4vector* comme clé et en le rattachant avec la suite de la liste chaînée (avec la suite du document).

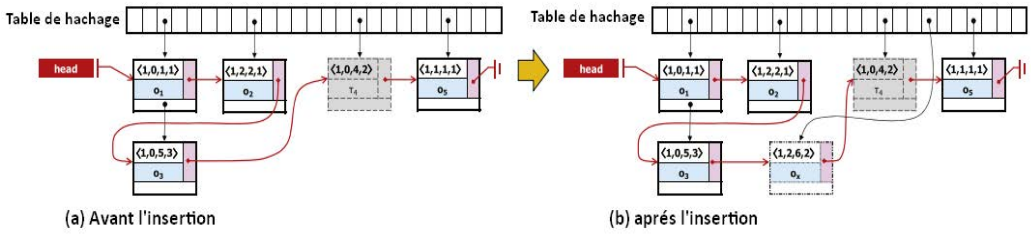


FIGURE 2.14 – Exemple d'une structure de donnée dans RGA.

Une fois que S_k (*s4vector*) d'un nœud est défini, il reste fixe, on l'utilise donc comme indice dans les opérations distantes. Lors de la génération d'une opération locale, par exemple, Insertion (3, O_3) (figure 2.14 (a)) sera diffusée sous la forme insertion (<1, 0, 5, 3>, O_3) aux autres répliques.

Quand une opération distante est reçue, son *s4vector* est comparé avec les *s4vector* des autres éléments. La figure 2.15 suivante montre les étapes d'insertions.

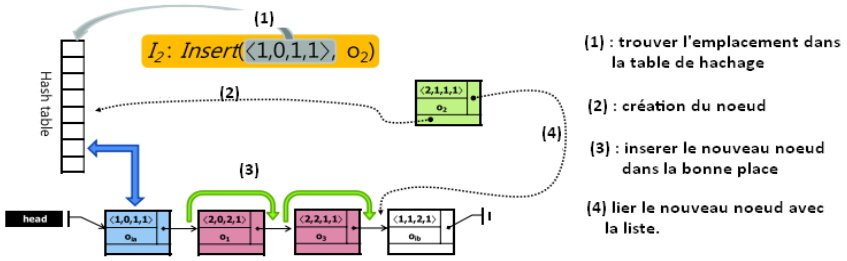


FIGURE 2.15 – Les étapes d'insertion dans RGA.

On suppose dans la session 2 que 3 répliques génèrent en concurrence 3 insertions à la même position : La réplique 0 génère $insert(1, O_1)$ avec un vecteur d'horloge : [1,0,1] –> *s4vector* = <2,0,2,1> La réplique 1 génère $insert(1, O_2)$ avec un vecteur d'horloge : [0,1,0]–> *s4vector* = <2,1,1,1> La réplique 2 génère $insert(1, O_3)$ avec un vecteur d'horloge : [0,0,1] –> *s4vector* = <2,2,1,1>.

D'après la définition de l'ordre que l'on a mentionné précédemment, l'opération générée par la réplique 1 est la plus prioritaire ($sum1$=2, $sum2$=1 et $sum3$=1), ensuite celle de la réplique 2 et enfin celle de la réplique 3.

En se basant sur l'utilisation des tables de hachage, la recherche d'un élément à insérer ou à supprimer est rapide. Ce qui réduit la complexité temporelle de cet algorithme par rapport aux autres. Malheureusement, l'utilisation des vecteurs d'horloge peut être un obstacle si le nombre d'utilisateurs est assez grand. De plus, RGA repose sur l'utilisation des pierres tombales, ce qui réduit sa performance au fil du temps.

Logoot

Logoot [119] est un algorithme CRDT destiné à l'édition des documents texte dans les réseaux pair à pair. L'idée de base est d'associer à chaque élément du document un identifiant unique. L'ensemble de ces identifiants avec le contenu est enregistré dans une table d'une façon ordonnée suivant une relation d'ordre. La figure 2.16 est un exemple d'un document Logoot.

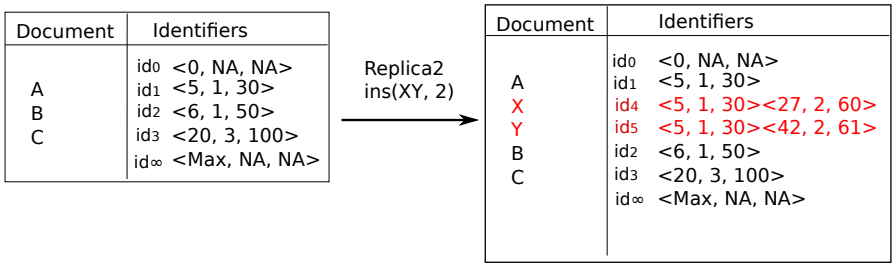


FIGURE 2.16 – Document Logoot.

Un document Logoot est borné par deux identifiants représentant le début et la fin du document, dans notre exemple c'est $< 0, NA, NA >$ et $< Max, NA, NA >$. Les autres identifiants représentent le contenu du document.

Une position P_i est un triplet : $< id, num, h >$

id un entier supérieur à 0 et inférieur à Max,

num identifiant de la réplique qui a généré l'opération,

h horloge logique d'un site, incrémenté à chaque opération générée,

Un identifiant Logoot est une suite de composant ordonné lexicographiquement. Par définition, dans la figure 2.16 : $id_0 \prec id_1 \prec id_4 \prec id_5 \prec id_2 \prec id_3 \prec id_\infty$

Création des identifiants On distingue deux stratégies pour la création des identifiants [118] :

1. Stratégie Aléatoire : Les identifiants Logoot sont générés aléatoirement entre deux identifiants.
2. Boundary : La stratégie Boundary limite la distance entre deux identifiants successifs. La distance entre deux identifiants ne doit pas dépasser la valeur boundary.

L'insertion locale dans Logoot : Lorsqu'un utilisateur insère un élément, un nouvel identifiant est créé, puis inséré dans la table en respectant l'ordre lexicographique. De plus, Les utilisateurs peuvent choisir arbitrairement leur stratégie. Les répliques peuvent utiliser des stratégies différentes ou modifier leur stratégie à tout moment pendant la session.

Une insertion ne se fait pas forcément élément par élément. Dans le cas d'un copier/coller par exemple l'utilisateur insère un ensemble d'élément dans la même opération. Dans ce cas un ensemble d'identifiant est créé qu'on appelle *patch*. Les identifiants insérés dans le patch sont contiguës et ordonnés. Créer des identifiants d'une façon optimale c'est-à-dire les plus courts possibles, correspond à trouver l'index le plus court de telle façon que le nombre d'identifiants disponibles entre l'identifiant précédent et suivant soit suffisant.

Dans la figure 2.16, si un utilisateur souhaite insérer un ensemble de caractères "XY" entre l'identifiant <5, 1, 30> et <6, 1, 50>, le nombre de place n'est pas suffisant. Dans ce cas, une augmentation de la longueur des identifiants est nécessaire. Les nouveaux identifiants auront une taille de 2. Soit id_4 et id_5 les 2 identifiants représentant X et Y tel que $id_1 \prec id_4 \prec id_5 \prec id_2$. Le document Logoot "AXYBC" avec les identifiants correspondants sont présentés dans la figure 2.16.

Suppression locale dans Logoot : Comme pour l'insertion, la suppression dans le document local est faite directement en utilisant la position de l'élément dans le document texte ; une opération delete(id) est créée et envoyée aux autres répliques afin qu'elles mettent à jour leur table et leur document. Contrairement aux algorithmes précédents, dans Logoot la notion de dépendance entre les éléments n'existe plus. Quand un utilisateur supprime un élément, ce dernier est supprimé définitivement du document. Il n'y a donc pas de pierre tombale.

Gestion des opérations distantes : Lorsqu'une réplique reçoit une opération distante, cette dernière est traitée selon son type :

1. Si l'opération est une insertion, la réplique réceptrice cherche la position du nouveau identifiant reçu à insérer avec la recherche dichotomique. Une fois la position trouvée, l'opération est intégrée dans le document local
2. Si l'opération reçue est une suppression, la réplique réceptrice cherche la position de l'identifiant à supprimer avec la recherche dichotomique. Si elle n'est pas déjà supprimée par une opération concurrente, l'identifiant reçu est supprimé du document local.

Les deux avantages majeur de Logoot par rapport aux autres approches sont : (i) l'absence de pierre tombale, puisque il n'existe pas de dépendance entre les éléments, le caractère est supprimé physiquement. (ii) Le gain en temps de complexité, puisque la recherche des caractères est effectuée avec la recherche dichotomique. Cependant, le désavantage de Logoot est la taille des identifiants qui augmente indéfiniment, ce qui affecte le temps de recherche des éléments et l'espace mémoire.

Logoot Split

Contrairement à Logoot qui prend un seul élément comme granularité, l'algorithme LogootSplit [11] identifie une séquence contiguë d'opération générée par l'utilisateur. Une séquence continue peut être produite en temps réel par un simple copier/coller.

Un identifiant LogootSplit est composé d'une liste de positions. Pour permettre l'insertion au sein d'une séquence existante, chaque position est un 4-uplet contenants un chiffre dans une base numérique spécifique, un identifiant unique de la réplique , un nombre de décalage qui spécifie la position dans la séquence, et une valeur d'horloge logique. LogootSplit définit trois types d'opérations : insertion, suppression et split. Cette dernière casse l'identifiant d'une chaîne de caractères en deux lorsqu'un utilisateur insère au milieu de cette chaîne.

Dans la figure 2.17, l'utilisateur 2 génère sa 5ième opération et insère la chaîne de caractères *Helo* à la position 0 dans le document avec <1, 2, 0, 5> comme identifiant. Une fois cette opération reçu par la réplique 3, l'utilisateur insère le caractère 'l' à la position 3. Une opération split est générée pour insérer au milieu de *Helo*. L'identifiant de *Helo* est cassé en deux et un nouveau identifiant est inséré.

Comme dans Logoot, la suppression se fait directement par l'identifiant.

L'avantage majeur de l'algorithme LogootSplit est le nombre d'identifiants qui est réduit, alors que les performances de l'algorithme dépend de la proportion de séquences continues insérées.

Treedoc

TreeDoc [85] est un algorithme CRDT qui représente le document par une structure d'arbre binaire. Chaque élément est identifié par son chemin unique dans l'arbre. Il est représenté par une liste de triplet : le chemin vers l'élément dans l'arbre (gauche ou droite), un identifiant de la réplique et une horloge.

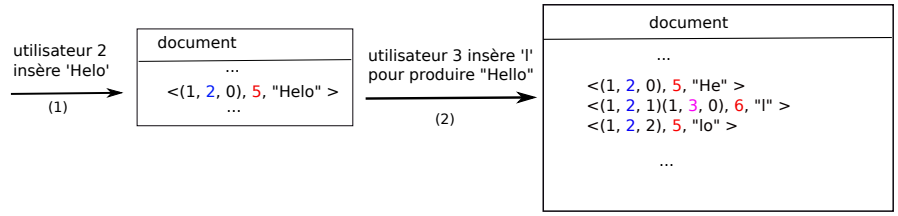


FIGURE 2.17 – Insertion dans LogootSplit.

Un nouveau noeud peut toujours être inséré entre deux noeuds consécutifs. L'insertion est réalisée comme le fils droit du noeud précédent ou comme le fils gauche du noeud suivant. Le contenu du document est obtenu en parcourant l'arbre par l'ordre infixe. Par exemple, la figure 2.18(a) est une représentation possible d'un document *"abcdef"*.

TreeDoc supprime uniquement les noeuds qui sont des feuilles, sinon il les garde comme des pierres tombales. Si plusieurs utilisateurs insèrent simultanément un élément à la même position, TreeDoc étend le noeud de base de l'arbre à un *noeud-majeur* contenant plusieurs noeud-mini. Ces derniers sont ordonnés selon l'identifiant de la réplique. Dans la figure 2.18, les deux utilisateurs 1 et 2 insèrent respectivement *'Y'* et *'X'* en même temps et à la même position (position 4). Treedoc crée un noeud-majeur qui contient deux mini-noeud *'Y'* et *'X'*. La figure 2.18(b) représente le document *"abcdYXef"*.

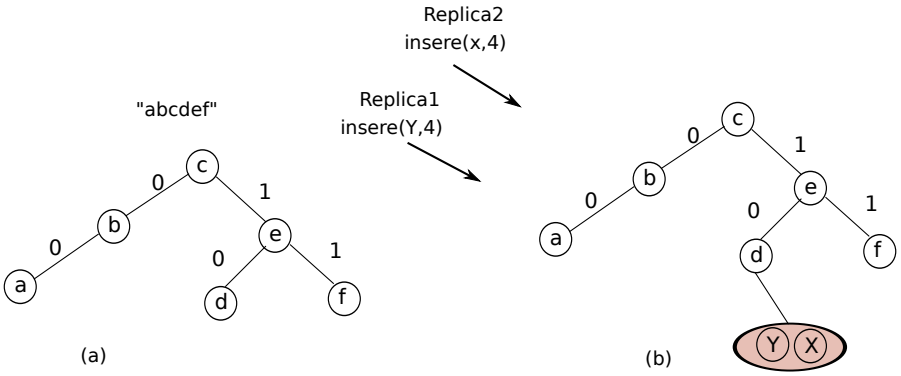


FIGURE 2.18 – Insertion dans Treedoc.

Pour améliorer les performances, éviter les pierres tombales, les noeud-majeurs, et l'arbre déséquilibré, les auteurs proposent une solution appelée *nettoyage structurel* qui consiste à créer un arbre équilibré sans pierre tombale. Toutefois, une telle opération nécessite un consensus.

Nous avons décrit précédemment les algorithmes CRDT dédiés à l'édition collaborative pour les document linéraires. Dans ce qui suit nous présentons quelques algorithmes de réplication dédiés aux documents (semi-) structurés.

2.3.6 Document structuré

Les structures de données de type arbre sont fondamentales dans plusieurs domaines de l'informatique et de l'ingénierie des systèmes. La consistance du document est plus difficile à garantir par rapport au document linéaire. En effet, plus le type de données est complexe, plus

les conflits apparaissent. Par exemple, dans une structure de données telle que les fichiers XML et les systèmes de fichiers, les modifications telles que l'ajout et la suppression du même élément, l'ajout d'un noeud enfant en supprimant le noeud père, ou l'ajout concurrent de deux fichiers avec le même nom sous le même répertoire, génèrent des erreurs. Dans ce qui suit, nous présentons quelques algorithmes de réplication qui assure la consistance d'un document (semi-) structuré.

TreeOpt

L'algorithme TreeOPT (Tree OPerational Transformation) [45] est un algorithme générique conçu pour les documents hiérarchiques et semi-structurés. La structure d'arbre est définie par 4 niveaux de granularité : niveau(0) le document ; niveau(1) paragraphe ; niveau(2) les phrases ; niveau(3) les mots, et niveau(4) les caractères.

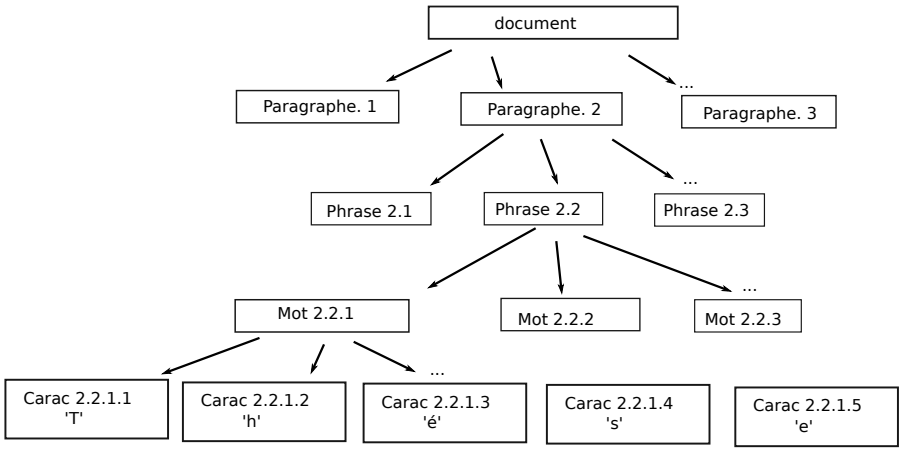


FIGURE 2.19 – Exemple d'une structure d'un document TreeOpt.

Chaque opération est définie par un $< niveau, type, position, contenu, vecteurd'tat, siteID >$ où le $niveau \in \{1, 2, 3, 4\}$ est la granularité, $type \in \{insertion, suppression\}$, *position* est la position du noeud dans l'arbre, *contenu* est le contenu du noeud, *vecteur d'état* est le vecteur d'horloge du site émetteur et $siteID$ est l'identifiant de la réplique. Pour simplifier l'opération, nous présentons une opération par son *type*, *niveau*, *position* et le *contenu* en ignorant les autres attributs. Par exemple, *insertMot(2,2,1,"Thése")* est une opération de type insertion, avec le niveau mot, dans le paragraphe 2, phrase 2, la position 1 et insérer un noeud avec un contenu "Thèse". La figure 2.19 est un exemple de la structure d'un document TreeOpt après l'insertion.

L'algorithme applique un mécanisme OT de façon récursive sur les différents niveaux du document. Chaque noeud possède une instance d'un algorithme OT. L'algorithme peut être combiné avec d'autres algorithmes tels que GOT(O) [104], dOPT [89] et SOCT2 [102].

TreeOpt ne garde pas l'historique des opérations dans un seul buffer mais il est partagé dans tout l'arbre. Lorsqu'une nouvelle opération est transformée, seul l'historique du chemin est influencé. Cette structure améliore les performances. L'autre avantage est que l'algorithme ne supporte pas seulement les caractères mais il s'adapte à n'importe quelle granularité à savoir, les mots, les phrases ou les paragraphes.

FCEdit

l'algorithme FCEdit [69] est un CRDT conçu pour l'édition collaborative des documents semi-structurés. Il associe à chaque élément un identifiant unique : $identifiant \rightarrow noeud$. Il utilise une table de hachage pour trouver un élément dans l'arborescence. Chaque noeud fils est ordonné par un identifiant de position. Contrairement à TreeOpt, FCEdit n'a pas besoin de stocker les pierres tombales, les éléments sont réellement supprimés de l'arbre, ce qui rend l'algorithme plus efficace en mémoire.

Contrôle des conflits

Dans [68], nous avons proposé des politiques de gestion de conflits pour les type de données structurés répliqués. Par exemple, dans un système de fichiers, si un utilisateur ajoute un fichier sous un répertoire qui est supprimé en concurrence, le fichier devient un noeud orphelin dans l'arbre.

Dans l'exemple présenté dans la figure 2.20, deux répliques partagent initialement le même arbre. La réplique 2 ajoute d sous b. En concurrence, la réplique 1 supprime b de l'arbre. Après la diffusion des messages, la réplique 1 ne peut pas ajouter d sous b puisqu'il est supprimé, et devient un noeud orphelin. Dans ce cas, il existe différentes façons d'adapter l'arbre et de corriger le conflit.

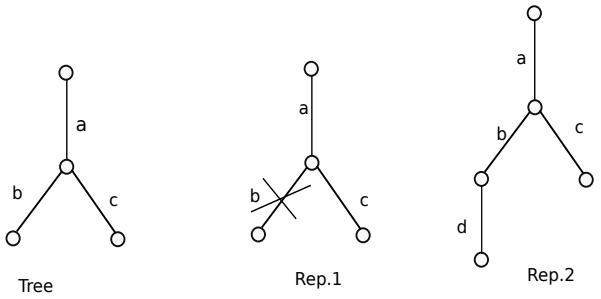


FIGURE 2.20 – Opérations concurrentes sur les arbres répliquées

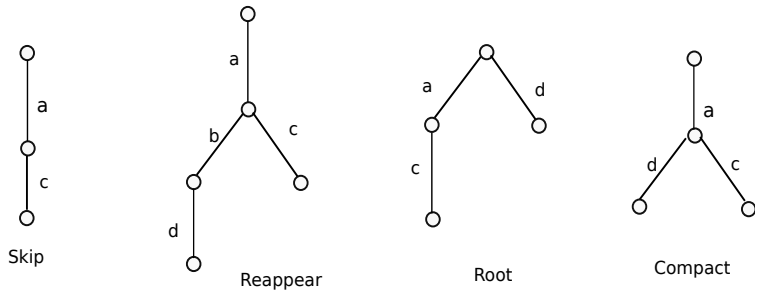


FIGURE 2.21 – Comportement différent pour la résolution des conflits dans les arbres

Pour résoudre ces conflits, nous avons proposé des politiques de gestion de concurrence pour les types de données arbre. Les différentes politiques sont présentées dans la figure 2.21 :

1. Skip : supprimer le noeud orphelin,
2. Reappear : recréer le chemin qui mène vers le noeud orphelin,

3. Root : placer le noeud orphelin sous le noeud racine,
4. Compact : placer le noeud orphelin sous le dernier noeud prédécesseur

2.3.7 Les applications existantes

Les différents algorithmes de réplication que nous avons décrits précédemment sont destinés à l'élaboration d'un système d'édition collaborative massive, c'est à dire avec un grand nombre de participants. Certains systèmes existants sont susceptibles d'assurer le bon fonctionnement du système sous les contraintes d'une édition massive. Dans ce qui suit, nous présentons quelques systèmes ainsi que les limites qu'ils présentent.

Google Docs Google Docs est un service lancé par la société Google pour le stockage et le partage de fichiers dans le cloud. Un fichier GoogleDoc permet un travail en ligne collaboratif. Un document textuel est mis en ligne et peut être partagé et modifié entre plusieurs utilisateurs. La société Google applique quelques restrictions sur le document :

— Seules 50 personnes peuvent modifier un document en même temps. Au-delà, les utilisateurs peuvent seulement observer le document mais pas le modifier. Google limite le nombre de personnes qui peuvent partager le document à 200 personnes y compris les observateurs. Si le document est partagé avec un grand nombre de personnes, les utilisateurs peuvent observer un retard durant l'édition [63].

— Un document GoogleDoc ne peut dépasser 1,024,000 caractères quels que soient le nombre de pages [42].

— Lors de l'intégration des mises à jour effectuée pendant les périodes déconnectées avec la version actuelle, le résultat peut être inattendu pour les utilisateurs. Soit le contenu du document présente une erreur sémantique ou l'intention des utilisateurs est violée.

Parmi les problèmes qui peuvent survenir, on trouve :

1. les mises à jour concurrentes sur la même phrase : deux utilisateurs peuvent modifier la même phrase simultanément sans être averti du changement de l'autre utilisateur, ce qui peut conduire à une erreur sémantique et la phrase n'est pas compréhensible ;
2. Les erreurs typographiques : lorsque les utilisateurs essayent de corriger la même erreur en insérant un caractère, deux caractères identiques peuvent apparaître dans le document ;
3. La position du curseur : durant l'édition, la position du curseur peut changer d'une façon inattendue lors de l'arrivée des mises à jour.
4. Les perte des mises à jour : dans certains cas, lorsqu'un utilisateur modifie un bloc de texte en mode déconnecté, ces modifications peuvent être perdu si un autre utilisateur supprime en concurrence le même bloc de texte. Par exemple, un utilisateur insère une ligne au milieu d'un bloc de texte et un autre utilisateur supprime tout le bloc. La ligne insérée peut être supprimée.

Etherpad Etherpad [37] est un outil d'édition collaborative en temps réel. Comme dans Google Docs, Etherpad offre à chaque utilisateur une copie de document pour la modifier localement. Pour permettre aux utilisateurs d'organiser leur travail, une fenêtre de messagerie instantanée est disponible.

Pour établir un ordre entre les opérations, Etherpad utilise une architecture centralisée. Toutes les modifications sont reçues par un serveur. Ce dernier associe à chaque opération une date qui lui définit un ordre d'exécution sur les autres sites.

En plus des inconvénients de la centralisation, Etherpad supporte au maximum seize personnes sur une session d'édition collaborative.

Microsoft Sky Drive Microsoft SkyDrive [5] utilise un support de collaboration plus stricte. Dans ce système, les utilisateurs sont obligés de se synchroniser de manière explicite. Lorsqu'un utilisateur enregistre le document, le document est envoyé au serveur. Si les modifications sont conflictuelles, l'utilisateur est invité à résoudre les conflits manuellement. Les autres utilisateurs ne verront pas le document fusionné (avec les mises à jour conflictuelles) jusqu'à ce que l'utilisateur résout les conflits. Pendant que l'utilisateur continue la modification de l'ancienne version du document, des mises à jour peuvent être exécutées sur le serveur ce qui peut générer d'autres conflits.

Cette solution maintient la consistance du document par l'utilisation d'un serveur centralisé. Cette solution ne tolère donc pas les pannes.

Au cœur de ces applications d'édition collaborative nous trouvons des algorithmes de réplication. Améliorer les performances de ces applications revient à améliorer les performances des algorithmes de réplication et à les adapter selon l'application.

2.4 Évaluation des algorithmes de réplication optimistes

Comme nous l'avons précisé dans le chapitre 1.2.2, le contexte de notre étude est l'édition collaborative. Pour étudier le comportement des algorithmes de réplication et mesurer leur performance selon les critères d'évaluation vus dans la section 1.2.3, il faut dérouler une session d'édition collaborative sur une application qui implémente les différents algorithmes. Cependant, organiser une session d'édition collaborative à grande échelle en respectant les propriétés d'expérimentation est presque impossible car certains paramètres sont incontrôlables tels que la déconnexion des utilisateurs, la vitesse de transmission, etc. Il faut choisir une technique d'évaluation précise.

Critère	Modélisation	Simulation	Mesure
Étape	Tout	Tout	Prototype connu
Temps	Peu	Moyen	Varie
Outil	Analyste	Développeur	Technicien
Précision	Faible	Bonne	Varie
Difficulté	Facile	Moyenne	Difficile
Coût	Faible	Moyen	Élevé

Table 2.1 – Critères de sélection d'une technique d'évaluation [50].

Jain et al. [50] classent les techniques d'évaluation en trois catégories : modélisation, simulation et mesure. La modélisation consiste à évaluer les performances d'un système à partir d'un modèle paramétré et d'un environnement utilisant des théorèmes et des lois mathématiques. Par exemple, les chaînes de Markov. La simulation consiste à implémenter un modèle d'un système et son environnement et ensuite exécuter ce modèle et observer ses caractéristiques. Dans l'approche mesure, l'évaluation utilise un système réel. Il existe plusieurs paramètres à prendre en

considération qui nous aident à choisir la technique la plus appropriée à notre méthode. Ces paramètres sont présentés dans le tableau 2.1.

Le paramètre clé "étape" nous permet de suivre l'évaluation de l'exécution à tout moment. Suivre une exécution dans l'approche mesure est possible seulement si le comportement du système est connu au préalable. Si l'exécution consiste à évaluer un nouveau concept (algorithme, application, etc), seulement la modélisation et la simulation qui sont possibles. Le second paramètre est le temps d'évaluation. Généralement, le résultat est désirable le plus tôt possible. Si le temps est le critère le plus important, la modélisation est la meilleure solution. La simulation prend plus de temps, alors que l'approche mesure change selon le type du système à évaluer et l'environnement de l'évaluation. Le niveau de précision est un autre critère important. En général, la modélisation se base sur plusieurs hypothèses et néglige certains détails, ce qui rend le résultat plus faible en précision. La simulation prend en considération plus de détails et moins d'hypothèses que la modélisation, ce qui rend son résultat plus proche de la réalité. Même si l'approche mesure utilise un système dans un environnement réel, le résultat peut être influencé par les paramètres de l'environnement tels que la latence réseau et le débit.

Dans cette thèse, nous avons besoin d'évaluer les performances des algorithmes de réplication dans un environnement contrôlé, tout en respectant les propriétés d'évaluation décrites précédemment. Puisque la modélisation retourne un résultat peu précis et l'approche mesure complexifie la reproductibilité de l'expérience, nous utiliserons la simulation.

Plusieurs simulateurs génériques ont été conçus et mis à disposition des développeurs pour évaluer leurs algorithmes.

2.4.1 Les simulateurs pour les applications distribuées

La simulation est un outil puissant et important car il fournit un moyen d'exécuter un modèle d'une application sur un modèle d'un système réel.

Dans le contexte des applications d'édition collaborative, la simulation permet d'élaborer une session d'édition collaborative massive sur un modèle d'un système réel, et d'évaluer ensuite les performances des algorithmes utilisés. Dans ce qui suit, nous présentons quelques simulateurs proposés pour l'évaluation des performances des systèmes distribués.

SimGrid

SimGrid [20] est un simulateur qui permet d'étudier le comportement des systèmes distribués à grande échelle tels que les grilles, les systèmes pair à pair, les platformes de calcul de hautes performances ou les clouds. Il fournit des fonctionnalités de base pour la simulation d'applications réparties dans des environnements distribués hétérogènes.

L'outil SimGrid permet de réaliser une simulation en reposant sur un modèle d'un système dont la validité a été expérimentalement évaluée et démontrée. Ces modèles prennent en considération plusieurs caractéristiques telles que la latence réseau, l'architecture et la dynamicité du système, la charge du réseau, la disponibilité des sites, etc.

SimGrid offre une application permettant la visualisation des traces de simulation pour observer le déroulement de l'expérimentation, et la visualisation des résultats pour bien comprendre le comportement des algorithmes.

Grid5000 et Planetlab

Grid'5000 [60] et Planetlab [23] sont des platformes réelles qui offrent à l'utilisateur des machines réparties dans le monde entier pour dérouler une expérimentation. L'utilisateur doit effectuer une demande auprès des administrateurs pour réserver des machines dans l'endroit où il le souhaite.

L'avantage principal des expérimentations sur Grid'5000 ou Planetlab est d'obtenir un résultat à partir de ces systèmes réels. Cependant, l'expérience n'est pas reproductible en raison des ressources dynamiques et de non contrôle des caractéristiques d'expérimentations. Par conséquent il est très difficile de reproduire le résultat.

PeerSim

PeerSim [76] est un logiciel de simulation de protocoles pair à pair. Il simule une interaction simultanée à l'aide d'un principe de cycle. PeerSim appelle à chaque cycle une méthode *nextCycle* de chaque réplique dans le réseau. Les actions effectuées dans le même cycle sont effectuées en parallèle. PeerSim prend un fichier de configuration en entrée qui spécifie le nombre de répliques, le protocole à utiliser, le nombre de répliques voisin, etc.

PeerSim permet de mesurer le temps d'exécution, le nombre de requête traitée ainsi que le nombre de message envoyé durant la simulation.

Cependant, PeerSim présente quelques inconvénients tels que, les jeux de données qui occupent une mémoire considérable puisque PeerSim doit les stocker dans une base de données, et l'application est parfois lente vu que chaque cycle, PeerSim demande à chaque réplique si elle a une action à effectuer.

NS-2

Le simulateur NS-2 [48] est le logiciel libre le plus populaire dans la simulation des réseaux informatiques. Il est principalement utile pour simuler les réseaux complexes à grande échelle, ainsi que pour tester les nouveaux protocoles. Le simulateur NS-2 intègre les fonctionnalités nécessaires à l'étude des algorithmes décentralisés.

Les algorithmes de réplication optimistes que nous allons évaluer ne nécessitent pas un système compliqué, ni des protocoles spécifiques. En effet, chaque algorithme passe par 4 étapes simples : *a*) exécution locale ; *b*) envoi de l'opération en mode *broadcast* ; *c*) traitement de l'opération reçue ; *d*) intégration dans la copie locale.

Les simulateurs décrits précédemment sont très compliqués pour nos besoins. De plus, évaluer les performances des algorithmes sur ces simulateurs nécessite des modules complémentaires à intégrer tels que les systèmes de génération d'opérations et un mécanisme pour le respect de la causalité.

L'idée est de construire notre propre simulateur qui simulera le système qui nous intéresse de façon rapide et peu coûteuse. Dans le chapitre 3, nous détaillerons notre outil d'évaluation.

Jusqu'à présent, aucun simulateur ou outil n'est mis à disposition pour l'évaluation des algorithmes de réplication optimiste. Les algorithmes existants sont évalués sur des outils spécifiques et privés.

2.4.2 Évaluation des algorithmes de réplication existants

Les algorithmes de réplication existants sont évalués alors sur des méthodes ad hoc. Ces méthodes visent à tester et exécuter les algorithmes sur des traces spécifiques. Généralement, ces méthodes sont utilisées par les chercheurs pour présenter le résultat de leur implémentation rapidement.

Framework pour un type de données répliqué : ensemble

Dans [29], les auteurs ont proposé un outil d'évaluation pour un type de données répliqué "ensemble" appelé SOR-Set. Contrairement à OR-Set décrit précédent, SOR-Set fragmente l'*ensemble* de l'objet en plusieurs *sous-ensemble* et le partage entre plusieurs machines. Lorsqu'un client génère une opération, cette dernière est transférée vers les autres en *sous-ensemble*. Dans le cas de state-based, un vecteur d'horloge est transmis entre les clients. Ceci a pour objectif d'envoyer seulement la différence des états et non pas l'état complet.

Une librairie client SOR-Set en Java est implémentée qui permet à chaque client de se connecter à n'importe quelle réplique et d'accéder à ces éléments ou pour se synchroniser avec d'autres répliques. L'état de chaque réplique est stocké dans une base de données Redis [2]. L'outil mesure le temps moyen nécessaire pour la fusion de deux ensembles, la taille de la base de données ainsi que le débit moyen de la communication. Cependant, l'outil utilisé n'est pas public et l'étude a été menée seulement sur SOR-Set .

Un algorithme OT pour une collaboration asynchrone sur mobile

ABST [97] est un algorithme OT proposé pour la fusion des séquences de mises à jour dans le même fichier, souhaitable pour les dispositifs mobiles. Les auteurs récupèrent les traces d'exécution à partir d'une édition collaborative en mode asynchrone. À partir de ces traces, les chercheurs ont évalué les performances de l'algorithme en fonction du pourcentage d'insertion seulement. De plus, l'évaluation a été faite sur une implémentation privée et l'algorithme (ABST) a été comparé avec seulement un autre (ABT).

Logoot et TreeDoc sur des traces asynchrone

Les approches Treedoc et Logoot ont été évaluées à partir des modifications produites sur des systèmes centralisés asynchrones [85, 120]. Logoot a été évalué sur des traces Wikipedia, alors que Treedoc a été évalué sur des traces récupérées à partir des répertoires SVN et des pages Wikipedia. Ces traces sont sérialisées et ne contiennent pas d'opérations concurrentes.

L'évaluation des deux approches n'a pas été faite sur la performance, mais seulement sur l'occupation mémoire et les méta-données.

FCEdit sur les traces simulées

L'évaluation de performance de l'algorithme FCEdit a été menée sur des traces simulées [69]. Pour prendre en considération la concurrence, l'auteur simule la concurrence sur les messages de la manière suivante : Chaque site reçoit 100 messages, puis il les permute d'une manière aléatoire sur une liste pour simuler l'ordre de réception. Donc, chaque site aura une liste différente de messages. Le processus continu ensuite avec les 100 messages suivants ...etc.

L'auteur a mesuré le temps global d'exécution en fonction du nombre d'opérations, de répliques et de la taille du document. Cependant, aucune évaluation sur l'occupation mémoire n'a été effectuée et aucune comparaison avec d'autres algorithmes existants.

Mesurer la qualité du résultat

Comme nous l'avons décrit dans la section 2.2, plusieurs approches de détection de conflits ont été proposées. Dans le but d'améliorer la qualité du résultat, ces approches mesurent à travers des métriques l'effort des utilisateurs durant une session d'édition collaborative asynchrone.

Par exemple, Orian [86] mesure deux métriques : le niveau de concurrence et l'effort de fusion. Le niveau de concurrence mesure sur une version le temps où plusieurs développeurs modifient à la fois la même version. L'effort de fusion mesure le nombre d'opérations effectuées pour résoudre un conflit après une opération de fusion. Une autre approche dite Palantir [95] propose deux mesures : la gravité et l'impact. La gravité est calculée en se basant sur le pourcentage des changements présents dans le fichier partagé. Pour un fichier donné, la gravité est le nombre de lignes de code modifiées divisé par le nombre total de lignes de code. Pour calculer l'impact, Palantir analyse tous les fichiers et les répertoires pour détecter les conflits entre les fichiers. L'impact est le nombre de fichiers modifiés par le nombre total de fichiers. Par exemple, sur un répertoire avec 20 classes, un utilisateur modifie le nom d'une méthode qui se trouve dans la *classeA* alors qu'un autre utilisateur fait appel à cette méthode à partir de la *classeB*. Palantir détecte une erreur de compilation avec un impact de 0.1 (2 fichiers sur 20).

D'autres approches proposées dans [53, 31, 96] visent à réduire le nombre de conflits en essayant de les détecter tant qu'ils sont encore à leur début.

Ces approches n'ont pas pour but d'évaluer des algorithmes différents mais nous pouvons nous inspirer de leurs métriques pour conduire notre évaluation.

2.5 Conclusion

Dans un premier temps, nous nous sommes intéressés à la réplication des données et nous avons décrit les problèmes liés à la réplication. Nous avons expliqué que les systèmes de réplication ne peuvent pas garantir à la fois la cohérence et la disponibilité des données, ainsi que la tolérance aux pannes. Nous avons alors discuté sur la solution proposée qui est la réplication optimiste.

Ensuite, nous avons présenté les différents types de données répliqués qui sont supportés par la réplication optimiste tels que les ensembles, les documents textuels linéaire et les documents structurés. Nous avons décrit pour chaque type de données quelques algorithmes susceptibles de résoudre le problème de la gestion de la concurrence, ce qui influence sur leur performances ainsi que leurs avantages et inconvénients.

Une étude de ces algorithmes nécessite une comparaison entre eux sur les différents critères décrites dans la section 1.2.3. À notre connaissance, il n'existe aucun outil qui intègre les différents algorithmes de réplication et respecte les propriétés d'expérimentation décrites précédemment.

Dans le chapitre suivant, nous allons proposer une méthodologie et un outil d'évaluation pour les différents algorithmes de réplication.

Chapitre 3

Méthodologie

Sommaire

Les algorithmes de réplication optimistes existants cherchent à assurer la cohérence à terme. Pour chaque type de donnée répliqué, qu'il soit ensemble, linéaire ou structuré, plusieurs algorithmes de réplication ont été proposés. Nous avons vu dans le chapitre 2.3 qu'il existe deux types d'approches (OT et CRDT). Chaque approche englobe plusieurs algorithmes différents. Évaluer ces algorithmes et les améliorer par la suite nécessite une méthode d'évaluation adéquate. Dans ce chapitre, nous proposons une méthode d'évaluation qui respecte les propriétés décrites dans la section 1.2.3.

Ce chapitre est organisé autour de quatre principales sections. La première section présente une vue globale sur le déroulement de l'évaluation. Cette section identifie les besoins nécessaires à l'expérimentation. Dans la deuxième section, nous présenterons le déroulement de notre méthode. Nous discuterons des différents types de données répliqués, des types de traces d'exécution, du déroulement de l'exécution et des mesures de performances. Avant de conclure, nous validerons notre méthode suivant les propriétés générales d'évaluation, nous présenterons un outil implémentant cette méthode et ses limites.

3.1 Analyse des besoins

Analyser les performances des algorithmes de réplication et comprendre le comportement des utilisateurs durant une collaboration nécessitent une étude préalable. Ceci a pour but d'identifier les besoins nécessaires à l'expérimentation.

3.1.1 Corpus

Mesurer les performances des algorithmes de réplication nécessite des traces d'exécution ou un corpus qui reflète une édition collaborative réelle.

Selon le type de donnée, les modifications effectuées sur l'objet répliqué sont différentes. Dans le contexte d'édition collaborative, les traces d'exécution sont les modifications effectuées sur le document répliqué. Pour une question de confidentialité, il est très difficile d'obtenir des traces d'exécution réelle d'une édition collaborative.

Les traces d'exécution de beaucoup de systèmes d'édition collaborative ne sont pas disponibles au large public. Donc, nous devons mettre en œuvre des mécanismes d'extraction de traces. Ces derniers doivent refléter une édition collaborative réelle.

Nous distinguons deux types de corpus : synchrone et asynchrone. Les corpus synchrones représentent les modifications effectuées sur un système d'édition collaborative en temps réel tels que Etherpad ou GoogleDoc. Alors que les corpus asynchrones sont les modifications effectuées sur un système de collaboration asynchrone tels que les systèmes de gestion de versions Git, SVN ou CVS.

Pour obtenir des traces synchrones nous proposons de modifier le code source d'une application d'édition collaborative et d'organiser ensuite des sessions d'édition collaborative sur cette application. À partir de cette expérience, nous pouvons récupérer des traces d'exécutions réelles.

Les systèmes de gestion de versions tels que Git, Mercurial ou Bazaar, sont largement utilisés pour le développement logiciel. L'accès à l'historique de beaucoup de projets développés sur ces systèmes est public. Ils contiennent des informations précieuses pour évaluer les performances des algorithmes de réplication et comprendre le comportement des développeurs durant la collaboration. Cependant, ces systèmes stockent les informations sous forme d'états, alors que certains algorithmes de réplication sont basés sur des opérations. L'idée est d'implémenter un mécanisme qui parcourt l'historique de ces projets et de créer à partir des états un corpus utilisable par les algorithmes de réplications basés sur les opérations.

3.1.2 Simulation

Pour mesurer et comparer les performances des algorithmes de réplication, il est nécessaire de les exécuter dans le même environnement, sous les mêmes conditions et avec le même corpus. Cependant, reproduire une session d'édition collaborative avec un grand nombre de participants sur les différents algorithmes, dans un environnement réel, en contrôlant les conditions d'expérimentation est impossible. En effet, certains paramètres sont incontrôlables tels que la déconnexion des répliques ou la vitesse de transmission. Ceci ne peut pas garantir une reproduction correcte de l'expérimentation.

Donc, pour réaliser une édition collaborative massive sur un modèle du système, tout en contrôlant les propriétés d'expérimentation, nous avons besoin d'un simulateur dans un environnement contrôlé. Ce simulateur simule une session d'édition collaborative en utilisant le corpus récupéré dans l'étape précédente. Selon la réplique qui produit les modifications et l'ordre dans le corpus, le simulateur demande à la réplique correspondante de produire une opération dans le

format de l'algorithme utilisé. Le simulateur demande à chaque réplique d'exécuter les modifications en premier lieu sur la copie locale avant d'être transmises aux autres répliques.

Puisque les traces d'exécution contiennent de la concurrence, le simulateur doit intégrer un mécanisme qui assure une diffusion causale des opérations entre les répliques. Le simulateur demande ensuite à chaque réplique qui recevra cette opération, de l'intégrer dans sa copie locale.

Ce simulateur garantit la reproductibilité de l'expérience tout en contrôlant les caractéristiques de la collaboration. Ainsi, nous pouvons reproduire l'expérience, analyser le comportement des utilisateurs et déduire quel facteur influence le plus les performances des algorithmes.

Durant l'exécution, nous mesurons les performances des algorithmes de réplications.

3.1.3 Mesures

Comme nous l'avons décrit dans la section 1.2.3, il existe plusieurs critères d'évaluation des algorithmes de réplication. Ces mesures nous permettent de comprendre le comportement des algorithmes de réplication et des utilisateurs durant la collaboration.

Le temps d'exécution des modifications durant une session d'édition collaborative est très important. Si l'utilisateur effectue une modification sur le document et si l'application ralentit son exécution et son affichage, l'utilisateur peut quitter la collaboration. De plus, si le système prend du temps pour intégrer les modifications des autres utilisateurs, des anomalies peuvent apparaître et perturber la collaboration. Par exemple, le changement inattendu de la position du curseur, des mises à jour concurrentes du même contenu qui produisent un contenu incorrect, etc. Donc, mesurer le temps d'exécution local et distant est important. Cependant, le temps de réponse peut être influencé par l'exécution d'autres processus sur la machine. Il est donc important pour cette mesure d'utiliser des exécutions isolées [5].

Certaines applications d'édition collaborative, telles que les applications sur les dispositifs mobiles ne peuvent utiliser que des algorithmes qui n'occupent pas une mémoire importante. L'espace mémoire requis par les algorithmes de réplication est donc une contrainte cruciale à mesurer.

Pour que les mesures soient correctes et significatives, la même méthode d'évaluation doit être utilisée pour tous les algorithmes. Ces derniers doivent suivre la même conception algorithmique. De plus, leur implémentation doit être efficace, c'est-à-dire que les algorithmes doivent effectuer seulement le traitement nécessaire et aucun autre traitement supplémentaire. Pour mesurer uniquement l'empreinte mémoire des objets stockés par les algorithmes de réplications et ignorer toute autre dépendance, nous utilisons un mécanisme de sérialisation.

Dans le cas d'une édition asynchrone, le système collaboratif doit générer le moins possible de conflit. Sinon, les développeurs seront obligés d'effectuer des efforts pour résoudre les conflits survenus après la fusion des mises à jour concurrentes. Il est donc important de comprendre le comportement des utilisateurs pour fusionner correctement les modifications concurrentes, réduire le nombre de conflits et améliorer la productivité de l'équipe. En se basant sur le corpus des gestions de versions et la simulation d'édition collaborative asynchrone, nous suivons les modifications jusqu'à la génération du conflit. Nous calculons ensuite l'effort effectué par les utilisateurs pour résoudre ces conflits suivant l'algorithme de réplication utilisé.

Dans cette section, nous avons décrit de manière générale les besoins nécessaires pour notre méthodologie. Dans la section suivante, nous détaillons chaque critère.

5. L'utilisation des ressources durant l'exécution n'est pas influencée par d'autres applications

3.2 Déroulement de la méthodologie

Dans cette section, nous discutons sur le déroulement de la méthodologie suivant les trois étapes décrites précédemment : corpus, simulation et mesure. Cependant, avant l'extraction du corpus, nous présentons les différents types de modifications supportables par les types de données répliqués.

3.2.1 Types de données

Chaque type de données est défini par un ensemble de modifications et un objet répliqué (ensemble, document textuel ou document structuré) sur lequel les modifications sont exécutées. Pour chaque type de données, plusieurs algorithmes de réplications ont été proposées. Selon l'algorithme utilisé, les modifications prennent un format différent et le contenu de l'objet répliqué n'est pas le même.

Cependant, pour utiliser les mêmes méthodes de mesure de performance sur tous les algorithmes, nous devons nous s'assurer qu'ils suivent la même conception. Pour cette raison, nous définissons une interface qui doit être implémentée par chaque type de données. Cette interface définit trois méthodes fondamentales :

- *applyLocal*(x) : [y_n] : cette méthode prend en paramètre une modification (x) et renvoie une liste d'opérations (y_n) déjà exécutées sur la réplique locale,
- *applyRemote*(y_n) : reçoit une opération en paramètre et l'intègre dans la copie locale,
- *lookup* : [x_n] : affiche le contenu visible de l'objet. Le contenu est l'effet de toutes les modifications exécutées sur l'objet.

Pendant l'exécution locale, l'algorithme de réplication génère l'opération correspondante à la modification effectuée, et l'exécute sur la réplique locale. Le format des opérations est différent pour chaque type de données. En effet, une opération effectuée sur les arbres est différente d'une opération effectuée sur les ensembles. En fonction du type de données et l'algorithme de réplication utilisé, le processus de génération et de l'intégration des opérations est différent, ainsi que le contenu de la copie locale.

Ensemble ou Set

Un type de donnée "ensemble" est un ensemble mathématique d'éléments. Un *élément* est une composante définie dans un vocabulaire.

Les modifications appliquées sur un ensemble sont :

- add(x) : ajout d'un élément (x) dans l'ensemble ;
- remove(x) : suppression d'un élément (x) de l'ensemble ;

Le *lookup* retourne un ensemble d'éléments.

Texte

Lorsqu'un utilisateur modifie le document textuel, il peut générer trois types de modifications :

- insert(*contenu*, *pos*) : insertion du *contenu* à la position *pos* ;
- delete(*pos*, *offset*) : suppression de *offset* éléments à partir de la position *pos*,

— replace(*pos*, *offset*, *contenu*) : remplacer les éléments qui se trouvent entre *pos* et *pos*+*offset* par le *contenu*.

Le *contenu* de la modification peut être un seul élément ou une suite d'éléments, par exemple lors d'un copier-coller. La *pos* est un entier qui indique l'emplacement de la modification dans le texte. Le *lookup* retourne une liste ordonnée d'éléments.

Arbre

Un arbre est une liste de noeuds. Un *noeud* est une interface implémentée par chaque algorithme. Chaque noeud est un chemin dans l'arbre. Dans le cas d'arbre ordonné, chaque noeud possède en plus un identifiant qui lui définie une position entre les noeuds du même père.

Les modifications appliquées sur un arbre sont :

— add($noeud_{pere}$, *contenu*) : créer un nouveau noeud avec un *contenu* et l'insérer sous le $noeud_{pere}$,

— del(*noeud*) : supprimer le sous arbre à partir du *noeud*,

— move($noeud_{orig}$, $noeud_{dest}$) : déplacer le sous-arbre qui se trouve à partir du $noeud_{orig}$ sous $noeud_{dest}$.

Le *lookup* retourne le noeud racine *root*. À partir du root, l'arbre peut être traversé pour afficher l'ensemble des éléments.

La modification *move* complexifie les problèmes d'édition [67]. Elle crée des boucles dans l'arbre et peut engendrer une structure inconsistante. Ce qui nécessite des traitements supplémentaires pour les détecter et les résoudre. Pour cette raison, la modification *move* n'est pas souvent prise en compte dans les éditeurs collaboratifs.

3.2.2 Générateur du corpus synchrone

Pour observer le comportement des différents algorithmes de réplication optimiste et étudier leur performance durant une session d'édition collaborative en temps réel, nous devons exécuter ces algorithmes sur un large corpus de traces d'édition collaborative *synchrone*.

Puisqu'il n'existe pas de traces publiques synchrones réelles incluant la concurrence, nous proposons un générateur de corpus. Pour que ce dernier génère des modifications proches d'une édition collaborative réelle, nous avons dans un premier temps récupérer des traces d'édition réelles. Ensuite, le générateur de corpus peut s'inspirer de ces traces pour générer des modifications plus larges ou légèrement différentes. Afin de récupérer ces traces d'exécution réelle, nous avons modifié le code source d'un outil d'édition collaborative appelé TeamEdit et nous avons mené une expérimentation avec un groupe d'étudiants.

TeamEdit

TeamEdit [107] est un éditeur de texte collaboratif en temps réel [6]. Nous avons modifié le code source de TeamEdit afin de récupérer le journal des modifications générées par les utilisateurs pendant la collaboration. Nous avons ajouté quelques fonctionnalités telles que Undo/Redo [7]. TeamEdit utilise un serveur central, mais seulement pour établir la communication entre les différents sites. Il ne sérialise pas les modifications, et n'utilise pas un mécanisme de fusion des

6. Il est écrit en Java, il fonctionne sur Mac OS X, Windows et Linux. C'est un logiciel open-source, sous licence GNU GPL.

7. Annulé/Rétablir

modifications concurrentes. Nous enregistrons les modifications effectuées par les utilisateurs en tenant compte des copier-coller et des suppressions d'un bloc de caractères.

Nous avons ensuite effectué deux expériences d'édition collaborative. Ces expériences ont été construites dans le but de récupérer le maximum de modifications concurrentes tout en garantissant une collaboration réelle (c'est-à-dire, sans prévenir le comportement des utilisateurs). Le résultat obtenu est enregistré dans un fichier. Nous avons extrait les informations du fichier et effectué des calculs à partir des informations extraites. Dans la section 4.1, nous présenterons plus de détails sur le résultat et l'expérience.

Ce corpus sera utilisé par un générateur de traces pour reproduire des traces simulées plus larges, basées sur les caractéristiques obtenues à partir de cette expérience.

Structure des logs

Lors des sessions d'édition collaborative, nous avons récupéré le journal des modifications. Par exemple, la figure 3.1 présente un journal de quatre modifications. L'utilisateur a inséré deux caractères puis il les a supprimés. On a donc deux modifications *insert* et deux modifications *delete*.

```
12  2011-02-14 09:29:00,734 INFO  name - Ins("j",80,[1-1,0-9,2-2],1297672140730,1,0)
13  2011-02-14 09:29:00,734 INFO  name - Ins("h",81,[1-1,0-10,2-2],1297672140734,1,0)
14  2011-02-14 09:29:03,265 INFO  name - Del(81,1,[1-1,0-11,2-2],1297672143265,1,0)
15  2011-02-14 09:29:03,468 INFO  name - Del(80,1,[1-1,0-12,2-2],1297672143468,1,0)
```

FIGURE 3.1 – Traces réelles dans le fichier Log.

Pour les modifications *insert*, nous enregistrons les caractères insérés, leurs positions dans le texte, un vecteur d'horloge, un timestamp Unix, le numéro de document et l'identifiant unique de l'utilisateur. Pour les modifications *delete*, nous enregistrons la position texte à supprimer, le nombre de caractère à supprimer, un vecteur d'horloge, un timestamp Unix, le numéro de document et l'identifiant unique de l'utilisateur.

Mécanisme de génération

Pour réaliser une évaluation significative et comprendre le comportement des algorithmes de réplication, il est impérativement nécessaire d'exécuter les algorithmes sur différents scénarii d'édition. Pour cela, il faut mener une expérience sur des traces d'édition collaborative plus large que celle récupérées de TeamEdit. Pour générer d'autres traces avec des paramètres contrôlés, nous avons implémenté un mécanisme de génération de modifications synchrones. Ce mécanisme est illustré dans l'algorithme 1.

Le *générateur* utilise un ensemble de paramètres contrôlés par l'utilisateur tels que le nombre de modifications à générer, le nombre de répliques, un profil de modification et de réplique, une probabilité de génération et des variables (delay et sdv) pour la génération de la concurrence. Le profil de modification (modifProfil) décrit les caractéristiques de la modification telles que le pourcentage d'insertion, le pourcentage et la taille des copier/coller, etc. Ces paramètres sont contrôlés par l'utilisateur. Le profil de la réplique (repProfil) est une classe qui représente le comportement d'une réplique. Cette classe utilise une probabilité dans une méthode appelée WILLGENERATE pour décider si la réplique génère une modification ou pas.

Dans une boucle sur le nombre de modifications, le *générateur* demande dans la ligne 13 à une réplique de produire une modification. Selon une probabilité de génération définie par l'utilisateur, la réplique décide de produire ou pas une modification cette itération. À partir de

la ligne 14 le générateur commence la production de la modification. Le générateur attribue un ordre d'exécution à la modification en incrémentant le vecteur d'horloge de la réplique qui va générer la modification dans l'entrée correspondante. Ce vecteur représente l'ordre d'exécution des modifications dans l'édition. Le générateur génère ensuite une nouvelle modification suivant un profil de modification entré. Pour générer la concurrence comme dans les traces réelles, le générateur calcule pour chaque réplique le moment (en terme de nombre d'itérations) pour lui délivrer la modification. Cette valeur rt est calculée suivant une loi normale sur des paramètres delay et sdv. Dans la ligne 17, le générateur récupère le vecteur d'horloge de chaque réplique à l'instant rt. Si la réplique n'a aucune modification à récupérer à cet instant, le générateur lui attribut le vecteur d'horloge de la modification. Si la réplique à déjà des modifications à récupérer à l'instant rt, le générateur met à jour le vecteur d'horloge de la réplique. Si la réplique courante n'a pas pu générer une modification cette itération, le génération demande à une autre réplique de ré-exécuter ce processus. Ce processus est exécuté jusqu'à ce qu'une réplique réussit la génération d'une nouvelle modification.

Algorithm 1 Génération des modifications

```
1:  nbModif : nombre de modifications à générer,
2:  nbRep : nombre de répliques,
3:  modifProfil : le profil des modifications,
4:  repProfil : le profil des répliques,
5:  delay : délai moyen de propagation,
6:  sdv : écart type des retards entre les opérations,
7:  nbop=0 : nombre d'opération générée,
8:  replicaGen : le numéro de réplique qui tente la génération,
9:  proba : probabilité de génération,
10: function GENERERMODIFICATION( )
11:     newModif = null ;                                   ▷ la modification à générer
12:     while (nbop < nbModif && newModif == null) do
13:         if ( repProfil.willGenerate(replicaGen, proba)) then   ▷ tenter la génération
14:             VC.inc(replicaGen) ;                        ▷ incrémenter le VC de la réplique
15:             newModif = modifProfil.randomModif(VC, replicaGen) ;   ▷ génération
16:             for (int i = 0 ; i < nbRep ; i++) do
17:                 long rt = nbop + nextLongGaussian(delay, sdv) ;   ▷ temps de délivrance
18:                 VectorClock x = delivery[i].get(rt) ;   ▷ x : le VC de la réplique i à l'instant rt
19:                 if (x == null) then
20:                     delivery[i].put(rt, VC) ;
21:                 else
22:                     x.update(VC) ;                      ▷ mettre à jour le vecteur d'horloge x
23:                     delivery[i].put(rt, x) ;
24:             nbop++ ;                                    ▷ incrémenter le nombre d'opérations générées
25:         replicaGen = (replicaGen + 1) % nbRep ;         ▷ Choisir une autre réplique
        return newModif ;
```

3.2.3 Générateur du corpus asynchrone

Des traces massives d'édition collaborative asynchrone sont disponibles publiquement. Ces traces constituent un corpus très précieux pour évaluer les algorithmes et comprendre le comportement des développeurs pendant une session d'édition collaborative asynchrone.

Plusieurs services de gestion de développement logiciel hébergent un grand nombre de ligne de code des logiciels open-source. Par exemple, GitHub a 3.4 millions de développeurs et 6.5 millions de répertoires [8], Assembla a 800,000 développeurs et plus que 100,000 projets [9], ou SourceForge avec 3.4 millions de développeurs et 324,000 projets [10]. Lorsque le code est géré par un système de gestion de versions distribué comme Git, Mercurial ou Bazaar, et que le code source est public, l'accès à l'historique de l'édition concurrente est public aussi. Ces histoires doivent être traitées pour être supportées et exécutées par d'autres algorithmes.

Remarque. *Dans cette thèse, pour obtenir les traces réelles d'édition collaborative asynchrone, notre générateur de traces parcourt et analyse les dépôts du système le plus utilisé [6] : Git.*

Git est un système de gestion de versions décentralisé, très utilisé dans la gestion de développement logiciel. Il enregistre l'évolution d'un projet au cours du temps de manière à ce qu'on puisse revenir à une version antérieure d'un fichier à tout moment.

Le système Git repose sur le principe de branche pour travailler en concurrence ou sur une fonctionnalité en limitant les interactions avec le reste de l'équipe. Un utilisateur sur Git crée une branche pour diverger de la ligne principale de développement et effectuer des traitements sur la copie locale sans se préoccuper de ce que les autres développeurs modifient. Une fois le traitement fait, l'utilisateur peut fusionner les deux branches. Avant d'entamer la procédure d'extraction du corpus, nous définissons quelques termes utilisés :

— commit : enregistrement des modifications effectuées dans le répertoire local. Pour chaque commit, la description de la version courante du document est enregistrée dans l'historique avec un identifiant unique *sha1* ;

— push : mettre les modifications locales sur un dépôt Git. Après un push, les modifications sont visibles aux autres développeurs ;

— fusion : fusionner plusieurs branches dans une branche unique. Par défaut, Git utilise l'algorithme *git-Merge* [40] pour effectuer la fusion. Cependant, la procédure de la fusion peut produire des conflits si deux versions du document ont été modifiées en parallèle sur la même partie. Git ne saura pas quelle version utiliser. Dans ce cas, la trace de la fusion n'est pas directement stockée dans l'historique. Les utilisateurs sont invités à corriger manuellement le résultat de la fusion avant d'enregistrer un merge.

— merge : après la fusion et la correction du conflit, Git permet de stocker le merge des versions du document. Git considère le merge comme un commit mais généré à partir de plusieurs commits différents (c'est-à-dire il a plusieurs prédécesseurs). Ceci nous est utile pour garder une trace de tous les merges qui ont lieu dans le dépôt ;

— diff : observer la différence en terme de modifications entre deux versions. Git exécute un diff avant chaque merge pour vérifier s'il y a des conflits ou pas ;

— revert : annuler les changements d'un commit.

Les branches et la fusion

Dans le système Git, les développeurs travaillent sur des copies séparées qu'ils modifient localement. Ils éditent, compilent et testent leur propre version de code source. La transmission des modifications se fait ensuite par un push. Le développeur décide quand il commit, quand

8. `https://github.com/about/press`
9. `https://www.assembla.com/about`
10. `http://sourceforge.net/about`

il rend les modifications visibles aux autres développeurs et quand il fusionne et intègre les modifications des autres développeurs.

En absence de modifications locales, la fusion avec les commits des autres développeurs est faite en silence, car la version des autres développeurs remplace entièrement la version locale. Par contre, si un utilisateur a effectué des modifications locales, Git utilise un outil appelé *git-Merge* pour produire une meilleure fusion avec les modifications concurrentes. Cependant, si les modifications concurrentes sont faites dans le même document et dans la même zone, git-Merge génère un conflit. Les développeurs doivent résoudre ces conflits à la main avant d'enregistrer le merge. Pour aider les développeurs à résoudre ces conflits, Git peut être configuré pour appeler un outil visuel interactif tel que mergetool, emerge ou kdiff3.

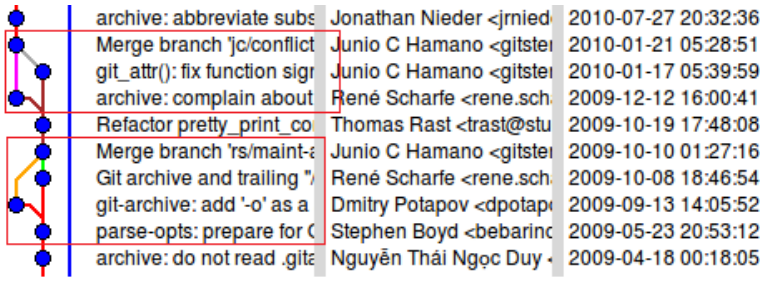


FIGURE 3.2 – Historique d'un corpus Git

Git stocke toutes les informations de branchement dans l'historique sous forme de graphe, comme présenté dans la figure 3.2. Les modifications concurrentes apportées par deux développeurs différents peuvent apparaître comme une liste si elles sont produites consécutivement, ou comme des branches si elles sont produites en concurrence. Un merge peut être produit par l'un des développeurs qui a effectué le commit concurrent ou par une personne tierce. Un merge peut impliquer deux ou plusieurs branches.

Les conflits dans la pratique

Nous supposons que le résultat du merge disponible dans l'historique de Git est le résultat attendu par l'utilisateur, même si la procédure de fusion peut générer des problèmes. Par exemple, un code source qui ne compile pas. On peut différencier les conflits directs et indirects [15]. Les conflits directs sont détectés lorsque les utilisateurs modifient en parallèle la même partie du document. Par exemple, un utilisateur qui modifie un paragraphe, alors qu'un autre le supprime en concurrence. Les conflits *indirects* sont produits à cause d'une erreur de compilation. Par exemple, un utilisateur fait appel à une méthode Java, alors qu'un autre utilisateur l'a renommé.

Après avoir analysé trois projets open-source, Brun et al. [15] ont trouvé que 76% de résultats de fusion potentielles ont été faites correctement, 16% sont des conflits textuels, 1% des erreurs de compilation et 6% de tests échoués. Les conflits textuels nécessitent une intervention des développeurs pour valider les modifications et les rendre visibles aux autres développeurs. Alors que les conflits de compilation et de test ne nécessitent aucune intervention de l'utilisateur au niveau du DVCS. En effet, les modifications sont mergées automatiquement et sans conflit mais le résultat ne compile pas et/ou les tests ne s'exécutent pas correctement.

Git propose des commandes pour revenir manuellement dans l'historique et corriger les problèmes. En raison de la structure du graphe, revenir sur un commit n'apparaît pas dans l'historique.

Pour évaluer le nombre de merge problématique réellement présent dans l'historique, nous

avons mené une étude sur le dépôt du logiciel Git [11]. Nous avons mesuré le nombre de merges qui ne compilent pas et le nombre de merges annulé (ou revert).

— 30 sur 10 000 commits les plus récents de la branche principale *master* ne compilent pas. Parmi ces 10 000 , 3 085 sont des merges et seulement un merge ne compile pas.

— Sur toute l'histoire (30 614 commits), seulement 4 commits portent le message de revenir sur un merge [12], contre 7 231 messages de merge.

Ces mesures montrent que les résultats de merge présents dans l'historique sont dans l'immense majorité effectués correctement.

Extraction du corpus

Le système Git gère et stocke les informations relatives à la collaboration comme un instantané d'un mini système de fichier. Pour chaque commit, Git enregistre *l'état* des fichiers. L'état est défini par le contenu des fichiers au moment du commit et une référence à ces fichiers. Pour être efficace, si le contenu du fichier n'est pas changé, Git ne stocke pas le fichier à nouveau, juste une référence vers le fichier original qui n'a pas été modifié.

Pour générer des modifications à partir de l'historique Git et les préparer à être exécuté correctement par les algorithmes de réplication basés sur les opérations, nous devons transformer l'*état* des fichiers en une suite de *modifications* supportables par les algorithmes de réplication.

Dans la section 2.3, nous avons expliqué que certains algorithmes de réplication utilisent les identifiants des répliques pour identifier les éléments. Le système Git ne garde pas d'informations suffisantes à propos des répliques. En effet, le système stocke seulement l'adresse mail de l'utilisateur qui a effectué un commit/merge. Cependant, l'adresse mail n'est pas fiable car le même utilisateur peut travailler sur plusieurs répliques, ou changer de mail tout en travaillant sur la même réplique.

L'idée est de concevoir un mécanisme qui parcourt l'historique de Git et crée un graphe de noeuds basé sur les merge/commit. Ensuite, le mécanisme traverse ce graphe et attribue des identifiants de réplique pour chaque noeud. Nous considérons que dans l'historique de Git chaque branche représente le travail d'une réplique différente. Après un merge, les différents parents du nœud sont donc des répliques différentes. De cette façon, on attribue des identifiants différents pour les nœuds parents. Les identifiants sont attribués arbitrairement et sans priorité pour les répliques.

Pour des raisons de performances, on réduit de manière heuristique le nombre de répliques. Le principe de base est :

1. les branches concurrentes sont des répliques différentes,
2. utiliser les répliques précédentes tant qu'elles sont libres, sinon créer une nouvelle réplique,
3. n'avantager aucune réplique par rapport à une autre.

Nous présentons dans la figure 3.3 un exemple d'extraction de répliques à partir de l'historique de Git. Au début de l'édition, deux branches concurrentes (a et b) sont créées à partir d'une seule branche. Puisque chaque branche est une réplique différente, nous avons généré donc deux répliques, 1 et 2. Ensuite, une opération de merge est effectuée. Puisque nous réutilisons les répliques précédentes, ce merge est produit par la réplique 1 ou par la réplique 2. Soit la réplique 1 qui a effectuée le merge. La réplique 2 devient associée à cette branche.

11. `https://github.com/git/git`, à partir du commit `0da7a53a`
12. Revert "Merge ..."

Dans l'étape B, deux branches concurrentes (x et z) sont crées à partir d'une seule branche (y). Puisque les répliques 1 et 2 sont associées à y, donc elles sont utilisées de nouveau. Lorsque d'autres branches sont créées à partir des branches x et z, on aura quatre branches en concurrence. Les répliques 1 et 2 ne suffisent pas et donc de nouvelles répliques sont créées.

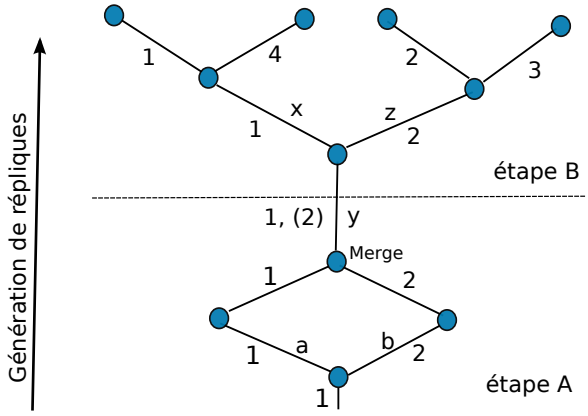


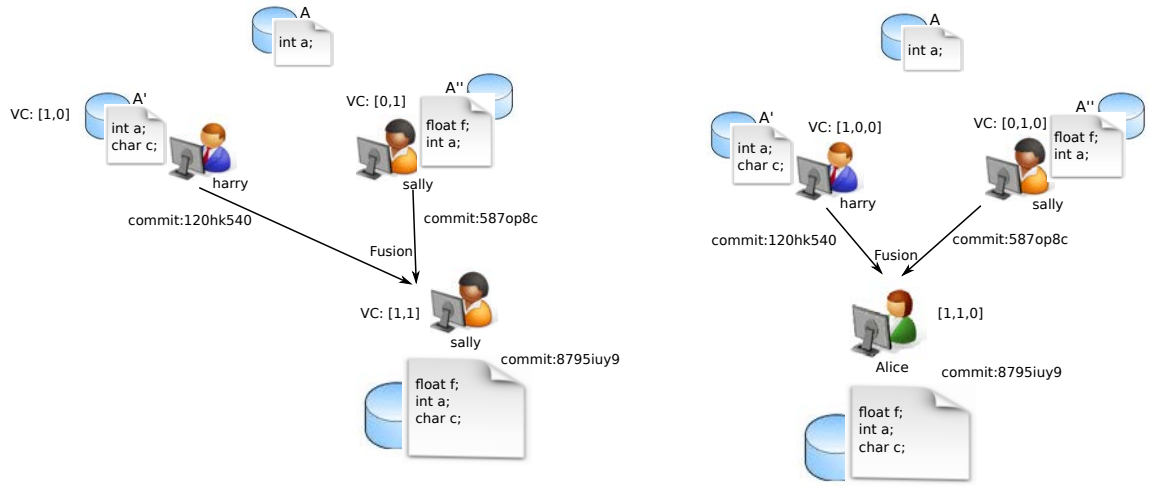
FIGURE 3.3 – Extraction de répliques

Même si le nombre de répliques extrait est inférieur à la réalité, le résultat du merge ne change pas. En effet, le résultat du merge est indépendant des identifiants des répliques. Par exemple, le scénario exécuté dans les figures 3.4a et 3.4b obtient le même résultat alors que le nombre de réplique est différent. En effet, dans la figure 3.4a, il n'existe que deux participants sur deux répliques différentes (Harry et Sally). Harry insère `"char c";` dans la première ligne et en concurrence Sally insère `"float f";` dans la troisième ligne. Sally décide ensuite de fusionner ses modifications avec celle d'Harry. Les documents sont mergés correctement. Contrairement à la figure 3.4a, dans la figure 3.4b il existe trois participants, chacun sur une réplique différente. Le nouvel utilisateur (Alice) qui a effectué l'opération de fusion n'a pas participé à la collaboration, alors que le résultat obtenu est le même que celui obtenu dans la figure 3.4a. Notre méthodologie n'a aucun effet sur le résultat final puisque les merges sont rejouées de la même manière quelque soit le nombre de répliques.

Pour rejouer l'édition collaborative à partir de l'historique de Git en utilisant les algorithmes de réplication, nous devons préparer les modifications correspondantes. Ces modifications sont préparées en deux étapes :

1. Production : dans cette phase, les modifications sont récupérées avant l'exécution. Nous traversons l'historique de Git dans le sens inverse. Ceci nous permet de produire les modifications effectuées par l'utilisateur durant l'édition collaborative.

   Lorsqu'on détecte un *commit*, on identifie la réplique (si elle ne l'est pas déjà) et on calcule la différence ou le diff [77] entre l'état courant et le parent. Le résultat de ce diff est un ensemble de *modifications* que nous stockons pour les rejouer dans la phase suivante. Lorsqu'on détecte un *merge*, on identifie la réplique et on stocke l'*état* du document. Cet état va nous servir dans l'étape suivante pour corriger le résultat du merge produit par les algorithmes de réplication. Pour une question de performances, on stocke l'état du document dans une base de données *Apache CouchDB*. Ceci nous garantit la reproductibilité de l'expérience sans pour autant re-parcourir l'historique pour chaque exécution.
2. Exécution : dans cette étape nous rejouons l'historique de Git en utilisant les algorithmes de réplication. Pour chaque commit, nous utilisons les modifications correspondantes récu-



(a) Modification avec deux utilisateurs (b) Modification avec trois utilisateurs

FIGURE 3.4 – L'influence de nombre de réplique sur le résultat du merge

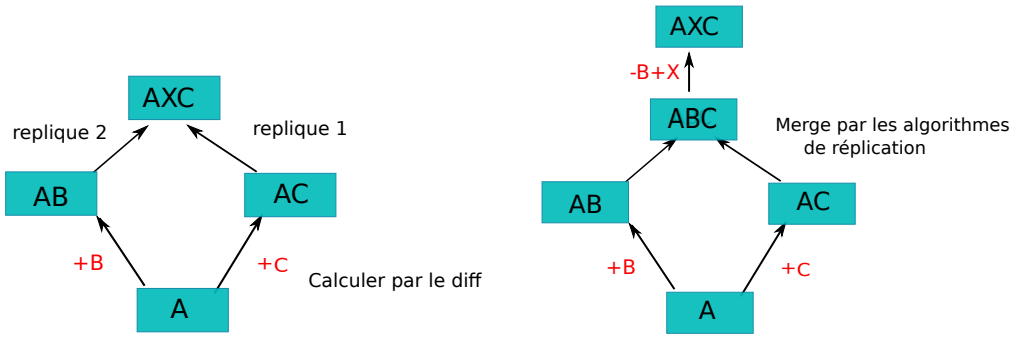
pérées dans la phase de production. En fonction de l'algorithme utilisé, les modifications sont exécutées différemment. Par conséquent, le résultat produit après une fusion diffère d'un algorithme à un autre.

Lorsqu'on détecte un merge, on fusionne les modifications avec les algorithmes de réplication. Le résultat est comparé ensuite avec l'état du merge que nous avons stocké dans l'étape précédente. La différence entre ces états est un ensemble de modifications produites durant l'exécution.

Pour bien comprendre le déroulement d'extraction de modifications, nous présentons un exemple. La figure 3.5 présente un aperçu d'un historique de Git rejoué par les algorithmes de réplication. Dans la phase de production présentée dans la figure 3.5, le mécanisme d'extraction détecte un merge avec un contenu `"AXC"`. Le mécanisme assigne donc deux identifiants différents aux parents, soit la réplique 1 et la réplique 2 ; et stocke ensuite l'état du document dans une base de données. Le mécanisme d'extraction continue le parcours de l'historique. Il détecte que les répliques partagent initialement le même document `"A"`. La réplique 1 a commité `"AC"` et la réplique 2 a commité `"AB"`. Pour produire les modifications, le mécanisme calcule le diff entre les deux états. Ainsi, deux modifications sont produites : une insertion de "C" à la position 2 pour la réplique 1 et une insertion de "B" à la position 2 pour la réplique 2.

Durant la phase d'exécution présentée dans la figure 3.5b, les algorithmes de réplication exécutent les modifications récupérées dans la phase de production, en suivant l'historique de Git. Le résultat de la fusion dépend de l'algorithme utilisé. Le résultat peut être `"ABC"`, `"ACB"` ou autre. Supposons que l'algorithme utilisé pour la fusion obtient le résultat `"ABC"`. On compare donc ce résultat avec l'état du document stocké dans la base de données récupéré dans la première phase. Dans cet état, la version fusionnée est `"AXC"`. Pour corriger le document, on demande à la réplique à qui est affecté la fusion de générer l'opération `remplacer (2, 1, "X")` pour remplacer `"B"` en `"X"`. Cette opération est générée durant la phase d'exécution.

Ce processus d'exécution imite la correction humaine. De ce fait, nous pouvons mesurer la qualité du merge produite par les algorithmes de réplication. En effet, plus l'algorithme engendre de modifications pour corriger la fusion et moins l'algorithme devrait satisfaire l'utilisateur.



(a) Production (b) Exécution

FIGURE 3.5 – Imiter une correction humaine d'un merge

Format des opérations

Après un commit, des opérations Git sont générées. Ces dernières sont définies par un quintuplé d'élément : $< replicaID, VC, Fi, Ed, commit >$ où le $replicaID$ est l'identifiant de la réplique qui a effectué le commit, VC est le vecteur d'horloge associé à l'opération effectuée, Fi est le nom du fichier, Ed est une suite d'éditions et le $commit$ est le numéro sha1 correspondant.

Après un merge, une opération de merge est générée. Cette dernière est un quadruplet de format $< replicaID, VC, commit, Etat >$ où le $replicaID$ est l'identifiant de la réplique qui a effectué le merge, VC est le vecteur d'horloge de la $replicaID$ au moment du merge, $commit$ est le sha1 attribué au merge et $Etat$ est l'état du document à l'instant du merge. Le mécanisme d'extraction calcule ensuite la différence entre le document de la $replicaID$ et le document qui se trouve dans l'historique avec le numéro $commit$. Le résultat de diff est une suite d'*Édition* que la $replicaID$ doit exécuter pour arriver à l'état $commit$. Une édition est définie par un quintuplé $< type, deb, fin, lig_{aj}, lig_{sup} >$ où le type est soit une insertion, suppression ou mise à jour d'une ligne, deb est le numéro de ligne de début de la modification, fin est le numéro de ligne de fin de la modification, lig_{aj} est une liste de lignes à ajouter et lig_{sup} est une liste des lignes à supprimer.

Dans la figure 3.6, deux répliques partagent initialement le même document A. La réplique 1 avec le vecteur d'horloge ([1,20][2,3]) remplace le nom de la méthode "method" par "sum" dans la première ligne. Son vecteur d'horloge est incrémenté dans l'entrée correspondante et devient ([1,21][2,3]). Après le commit, la réplique 1 génère l'opération < 1, [1,21][2,3], calcul.java, Ed_1, 17854 > où Ed_1 est [<remplace, 1, 1, "public int method(int a, int b)", "sum(int a, int b)">].

En parallèle, la réplique 2 remplace les deux lignes "c=a+b ;" et "return c ;" par une seule "return a+b ;" à la position 3. Après le commit, la réplique 2 génère l'opération < 2, [1,20][2,4], calcul.java, Ed_2, 47781 > où Ed_2 est <remplace, 3, 4, "c=a+b ; return c ;", "return a+b ;" >.

Ensuite, les deux répliques mergent leurs modifications. Supposons que la réplique 1 qui a effectuée le merge, l'opération produite est sous la forme < 1, [(1,21), (2,4)], 854996, RESULT>. Le mécanisme d'extraction calcule ensuite la différence entre l'état courant de la réplique 1 avec l'état du document au moment du merge numéro 854996. Le résultat est Ed_1 et Ed_2. L'algorithme utilisé produit le même document que celui dans l'historique. Donc, aucune nouvelle modification n'est générée.

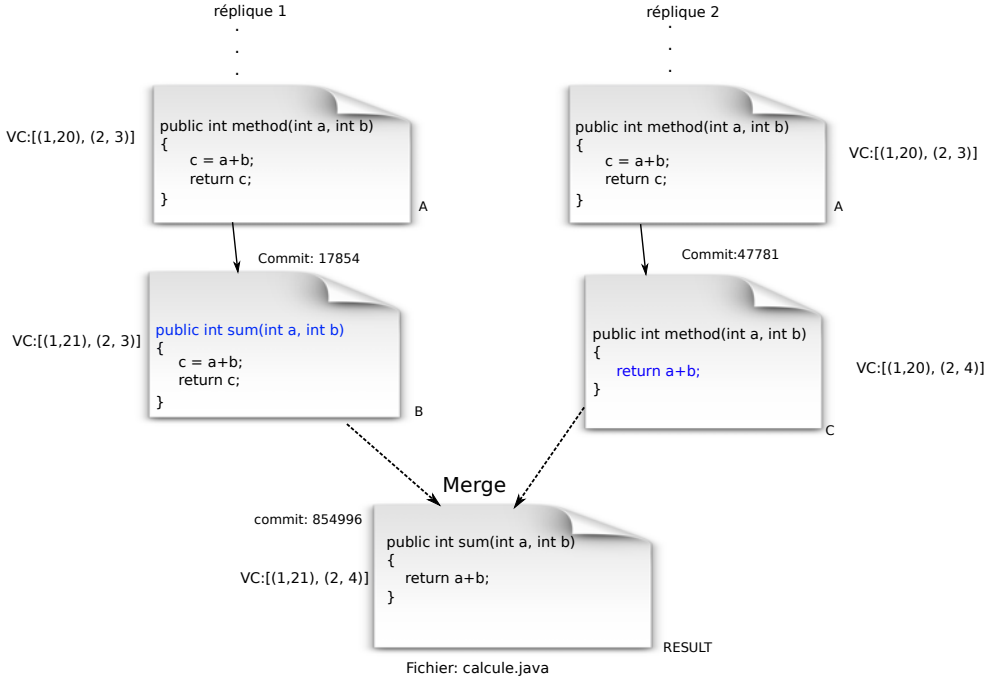


FIGURE 3.6 – Modification sur Git

3.2.4 Exécution

Maintenant que les modifications sont prêtes et que les types de données sont implémentés, nous pouvons rejouer une édition collaborative en utilisant les algorithmes de réplications. Ces derniers rejouent les modifications générées précédemment. Durant l'exécution, nous mesurons les performances des algorithmes de réplication. Comme nous l'avons indiqué dans la section 2.4, nous implémentons notre propose simulateur appelé *CausalDispatcher* sur lequel nous injectons les modifications pour les rejouer avec les algorithmes de réplication et les évaluer.

Quel que soit le type du corpus (synchrone ou asynchrone), chaque algorithme de réplication prend en entrée des modifications. Cette dernière représente l'action effectuée par les utilisateurs. Elle peut concerner un seul élément ou une suite d'éléments. Selon le type de données, les modifications prennent des formats différents. Ces modifications sont exécutées par les algorithmes de réplication directement dans la copie locale. Des opérations correspondantes aux modifications sont ainsi générées. Ces opérations diffèrent d'un algorithme à un autre. Les opérations sont ensuite propagées entre les répliques.

Le *CausalDispatcher* demande à chaque réplique d'exécuter les *modifications* et de produire les *opérations* correspondantes. Les opérations sont ensuite diffusées en mode *broadcast* entre les répliques. Avant de diffuser ou d'intégrer une opération, le CausalDispatcher fait appel à une méthode spécifique pour assurer l'ordre causal.

Le fonctionnement de *CausalDispatcher* est illustré dans figure 3.7. Dans cette figure, le CausalDispatcher demande à la réplique 1 d'exécuter la modification $Modif - 1$ en utilisant la méthode *applyLocal*. La réplique 1 exécute en local la modification et génère l'opération $OP1$ correspondante dans le format de l'algorithme utilisé. De la même façon, la réplique 2 génère à son tour sa première opération $OP2$.

La réplique 1 exécute une nouvelle modification $Modif - 3$ et génère l'opération correspondante $OP3$, avant de recevoir l'opération $OP2$. Le vecteur d'horloge dans $Modif - 3$ est $[2, 0]$,

Algorithm 2 Le Causal Dispatcher

```
1: function CausalDispatcher(trace)
2:     Map<Integer, List<Operation» history : historique des opérations produites par chaque
   réplique,
3:     Map<Integer, VectorClock> clocks : un vecteur d'horloge pour une réplique,
4:     VectorClock globalClock : vecteur d'horloge global,
5:     while (trace.hasMoreElements()) do
6:         Modification opt = trace.nextElement() ;  ▷ récupérer modification par modification
7:         int r = opt.getReplica() ;                ▷ récupérer l'identifiant de la réplique
8:         VectorClock vc = clocks.get(r) ;          ▷ récupérer le VC de la réplique
9:         Replica localReplica = replicas.get(r) ;  ▷ type : set, text ou arbre
10:        if ( !vc.readyFor(r, opt.getVectorClock())) then   ▷ la réplique est elle prête ?
11:            List<Operation> concurenteOps ;
12:            Iterator<Integer> i = opt.getVectorClock().keySet().iterator() ;
13:            while (i.hasNext()) do
14:                int e = i.next() ;                ▷ récupérer les identifiants des répliques
15:                if (e != r) then                  ▷ vérifier l'ordre causal
16:                    for (int j = opt.getVectorClock().get(e) ; j > vc.get(e) ;j–) do
17:                        insertCausalOrder(concurenteOps, history.get(e).get(j - 1)) ;
18:            for (Operation opConcur : concurenteOps) do   ▷ intégrer les op concurrentes
19:                localReplica.applyRemote(opConcur) ;
20:                int e = opConcur.getReplica() ;
21:                vc.inc(e) ;                       ▷ mis à jour du VC
22:        Operation op = localReplica.applyLocal(opt) ;   ▷ exécution locale
23:        history.get(r).add(op) ;
24:        globalClock.inc(r) ;                      ▷ mis à jour de l'horloge globale
25:    for (replica : replicas.values() ) do         ▷ récupérer les identifiants des répliques
26:        int n = replica.getReplicaNumber() ;
27:        concurenteOps.clear() ;
28:        VectorClock vc = clocks.get(n) ;          ▷ récupérer les VC des répliques
29:        for (Entry<Integer, Integer> e : globalClock.entrySet())  do
30:            for (int j = vc.get(e.getKey()) ; j < e.getValue() ; j++) do
31:                insertCausalOrder(concurenteOps, history.get(e.getKey()).get(j)) ;
32:        for (Operation opConcur : concurenteOps) do
33:            localReplica.applyRemote(opConcur) ;  ▷ intégrer les opérations manquantes
34:            int e = opConcur.getReplica() ;
35:            vc.update(e) ;
```

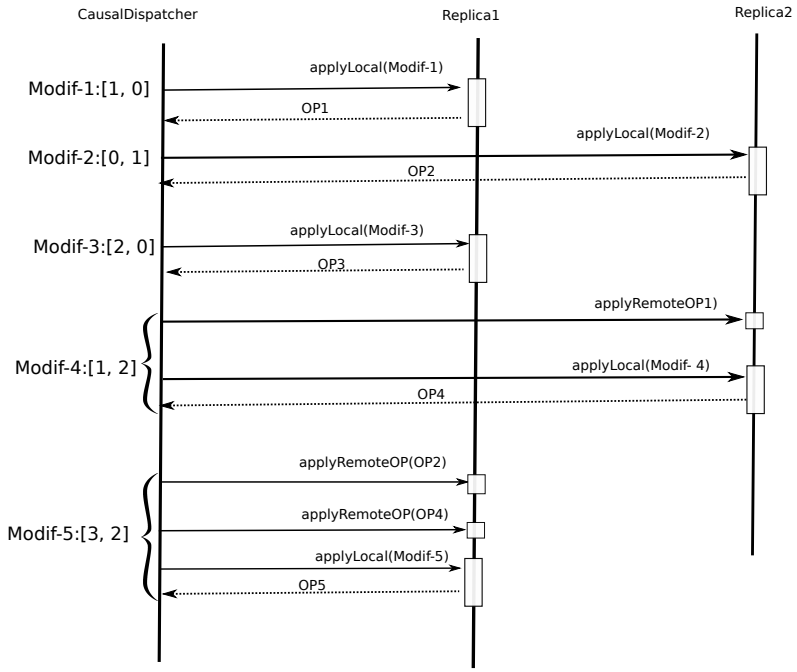


FIGURE 3.7 – Fonctionnement de CausalDispatcher

donc la modification est indépendante de la modification générée par la réplique 2.

Avant que la réplique 2 exécute la modification $Modif-4$ et génère l'opération $OP4$ correspondante, le CausalDispatcher vérifie dans le vecteur d'horloge de la modification si elle dépend d'autres modifications. Le vecteur d'horloge dans $Modif-4$ est [1, 2], donc $OP4$ dépend de la première modification effectuée par la réplique 1 ($Modif-1$). Le CausalDispatcher intègre l'opération $OP1$ dans la réplique 2 avant de générer $OP4$. De même, avant que le CausalDispatcher demande à la réplique 1 d'exécuter et de générer l'opération correspondante à $Modif-5$, le CausalDispatcher vérifie dans le vecteur d'horloge de la modification ($Modif-5$) si elle dépend d'autres modifications. Le vecteur d'horloge dans $Modif-5$ est [3, 2], donc elle dépend de $Modif-2$ et $Modif-4$. Le CausalDispatcher commence par intégrer $OP2$ et $OP4$ et seulement dans cet ordre, avant de générer $OP5$.

L'algorithme 2 présente l'implémentation de CausalDispatcher. Ce dernier prend en paramètre les caractéristiques des modifications dans une *trace*, et stocke pour chaque réplique les opérations générées dans *history*. À partir de la ligne 5, le CausalDispatcher commence le traitement des modifications. Il récupère pour chaque modification de la trace, le numéro de la réplique qui a générée la modification, le vecteur d'horloge de la réplique et le type de données. Avant d'exécuter la modification localement, le CausalDispatcher vérifie le respect de la causalité à partir de la ligne 10. Ceci est effectué en comparant le vecteur d'horloge de la réplique avec le vecteur d'horloge de la modification. S'il existe des modifications qui précèdent causalement la modification courante, le CausalDispatcher fait appel à la méthode *insertCausalOrder* pour récupérer ces modifications. La méthode *insertCausalOrder* compare les vecteurs d'horloges des répliques à partir de l'historique avec le vecteur d'horloge de la modification courante. *insertCausalOrder* stocke les modifications causalement dépendantes dans une liste.

Une fois la liste des modifications causalement dépendantes est récupérée, le CausalDispatcher les intègre dans la ligne 18 et met à jour le vecteur d'horloge de la réplique correspondante.

Ensuite, la modification locale est exécutée et une opération correspondante est générée. Après avoir parcouru la trace, Le CausalDispatcher vérifie pour chaque réplique à partir de la ligne 25 si elle a des modifications en attente d'exécution. Dans ce cas, le CausalDispatcher récupère les opérations (en attente) dans la ligne 31 et les intègre ensuite dans la réplique correspondante dans la ligne 33.

Lors de l'exécution, nous mesurons les performances des algorithmes de réplication. Dans ce qui suit, nous discutons sur les mesures.

3.2.5 Mesures

Puisque tous les algorithmes de réplication suivent le même protocole d'exécution à savoir, exécution locale, envoi et intégration, nous demandons à chaque algorithme de réplication d'implémenter une interface avec trois méthodes fondamentales décrites précédemment : *a*) *applyLocal*, *b*) *applyRemote*, *c*) *lookup*

De ce fait, tous les algorithmes suivront la même conception et les méthodes d'évaluation seront identiques pour tous.

Durant l'exécution, nous utilisons les opérations du système pour estimer le temps d'exécution et la sérialisation pour calculer l'espace mémoire occupé. Dans les chapitres 4 et 5 nous allons mesurer pour chaque opération de la trace :

- le temps d'exécution en local des modifications et la génération des opérations correspondantes, c'est-à-dire le temps d'exécution de la méthode *applyLocal* ;
- le temps d'exécution des opérations provenant des autres répliques ; c'est-à-dire le temps d'exécution de la méthode *applyRemote* ;
- la taille de la mémoire occupée par chaque algorithme de réplication. L'empreinte mémoire inclut l'espace mémoire des éléments qui sont dans le document plus la taille des métadonnées.
- la taille des messages échangées entre les répliques, i.e. la taille des opérations produite par l'algorithme de réplication utilisé.

Le temps d'exécution en local est le temps mis par une réplique pour exécuter une modification locale. Alors que le temps d'intégration est le temps mis par une réplique pour intégrer une opération distante. Plus le temps d'exécution en local est rapide et plus l'utilisateur observe ses propres changements rapidement. Donc, le temps d'exécution en local peut être considéré comme plus important que le temps d'intégration. En effet, l'utilisateur cherche à observer ses modifications rapidement. Si un algorithme de réplication n'est pas performant lors de l'intégration d'une opération distante, ceci peut ne pas être remarqué par l'utilisateur si les modifications ne concerne qu'un seul caractère. Par contre, l'intégration d'un bloc de caractère peut être remarqué si l'affichage de caractères contigus se déroule par une période suffisamment longue. Si l'utilisateur observe les mauvaises performances de l'application il quitte la collaboration. Dans le cas des modifications asynchrones, la taille et le nombre de modifications générées par l'algorithme influencent négativement le temps de traitement avec le dépôt distant, par exemple, le temps pour faire un push, un clone, etc. De plus, l'intégration des modifications concurrentes peut générer des conflits si les utilisateurs ont modifié le document dans la même zone du document. Comme nous l'avons décrit précédemment, le système ne peut pas choisir les changements d'un utilisateur et ignorer les modifications des autres. Dans ce cas, le système retourne le résultat à l'utilisateur pour résoudre les conflits. L'utilisateur doit exercer des efforts pour corriger le document. De ce fait, il faut comparer les algorithmes de réplication en terme de qualité de résultat.

L'algorithme qui produit un résultat de bonne qualité est celui qui produit le moins de conflit et par conséquent demande moins de correction à l'utilisateur.

Pour comparer les algorithmes de réplication sur la qualité du résultat, on compare le résultat de la fusion produit par les algorithmes de réplication, avec la version commité par les développeurs qui se trouve dans l'historique. La différence entre ces deux versions correspond à l'effort que le même développeur aurait mis lors de l'utilisation d'un DVCS basé sur l'algorithme de réplication évalué.

Nous proposons deux métriques :

— **Merge blocs** : le nombre de blocs différents dans le document, fusionné,

— **Merge lignes** : le nombre de lignes dans ces blocs.

Ensuite, on compare la qualité du résultat obtenu par les algorithmes de réplication, avec le résultat obtenu par l'outil utilisé par défaut par le système Git qui est *git-Merge*.

3.3 Adéquation de la méthodologie

Dans les chapitres 4 et 5, nous allons mener des expérimentations en se basant sur notre méthodologie. Il est donc important de valider l'outil d'évaluation. Dans cette section, nous validons notre outil, nous présentons notre logiciel et enfin nous discutons des limites qu'il présente.

3.3.1 Validation

Pour mesurer les performances des algorithmes de réplication de manière objective, il faut les évaluer suivant des critères bien définis. Dans ce qui suit, nous validons notre outil en fonction des propriétés d'évaluation vues dans la section 1.2.3.

Contrôle

Notre outil propose deux types de trace : synchrones et asynchrones. Dans le cas d'une exécution sur des traces synchrones, nous donnons à l'utilisateur le contrôle total sur les caractéristiques de génération des modifications tels que le nombre de répliques, la proportion d'insertion, la taille des blocs, le nombre d'opérations, etc. Si un utilisateur souhaite évaluer un algorithme de réplication, il suffit qu'il ajuste ces caractéristiques pour le confronter à des situations qu'il définit.

Dans le cas d'une exécution asynchrone, notre outil propose un mécanisme d'extraction de traces à partir des dépôts Git. L'utilisateur contrôle sur quel clone Git se déroule l'expérience et à partir de quel numéro de commit (sha-1) commence l'extraction de traces.

Reproductibilité

Grâce au contrôle des caractéristiques d'évaluation, notre méthodologie est reproductible, que ce soit en mode synchrone ou asynchrone. En effet, le résultat de l'évaluation des performances des algorithmes dépend des paramètres spécifiés par l'utilisateur. Pour reproduire une expérience suivant cette méthodologie, il suffit de donner les mêmes paramètres en entrée.

Dans notre étude, nous avons mené une série d'exécution en ajustant les mêmes paramètres. Nous rendons ces paramètres publiques pour assurer la reproductibilité de l'expérience [13]. Les résultats que nous avons obtenus sont les mêmes, si ce n'est une légère différence en temps

13. https ://www.dropbox.com/sh/yrux7yp0nqo00ik/LcX9Bcof9p

d'exécution et en mémoire. Cette différence est due à la génération aléatoire des traces d'exécution et à l'état du processeur au moment de l'exécution. Cependant, le comportement des algorithmes et les performances restent identiques.

Abstraction

Notre outil offre une abstraction d'une collaboration réelle. En effet, les algorithmes de réplication optimiste que nous allons étudier sont des algorithmes destinés à être implémentés sur des systèmes de réplication réels. En plus, le corpus sur lequel nous allons mener nos expérimentations sont réelles. Ils sont extraits soit à partir des traces de Git ou bien à partir d'une édition collaborative réelle. Seulement la communication entre les répliques est faite à travers un simulateur.

3.3.2 Framework

Pour appliquer notre méthodologie, nous avons conçu en langage Java un framework appelé *ReplicationBenchmark*[14]. Ce framework supporte les différents types de données répliqués et intègre les différents algorithmes de réplication décrits dans la section 2.3.

Dans ce framework, les algorithmes sont écrits dans le même langage de programmation, la validation des algorithmes est générique et le réglage des caractéristiques d'évaluation ne se fait qu'une seule fois au démarrage de l'exécution.

Ce framework est public[15], générique et extensible. Il mesure les performances des algorithmes, contrôle les modifications effectuées sur le document et assure la reproductibilité de l'expérience.

Les chercheurs peuvent intégrer leurs algorithmes, les comparer avec les approches existantes et évaluer leur performance. À cet instant, le framework intègre tous les algorithmes décrits dans la section 2.3 ainsi que d'autres. Les chercheurs sont invités à intégrer leurs algorithmes et mesurer leurs performances. Les algorithmes Treedoc, RGA et FCEdit ont été intégrées par leurs auteurs et l'étude de leurs performances ont été publiés dans [7, 68].

Le framework implémente des tests unitaires génériques pour tous les algorithmes optimistes. Un nouvel algorithme intégré dans le framework peut passer quelques tests ,par exemple, des tests de convergence et des tests pour vérifier les propriétés TP_1 et TP_2 pour les algorithmes OT. Si un test échoue, l'erreur est facilement identifiée. En effet, chaque méthode est testée indépendamment, donc la raison qui a provoqué l'erreur est directement affiché.

De plus, le framework sert lui même d'outil de test automatique, car on peut utiliser les traces simulées puis vérifier la convergence à terme.

3.3.3 Limites

Tout le framework repose sur notre méthodologie. Ce framework est le seul outil qui permet l'évaluation des performances des algorithmes de réplication sur des corpus réels, incluant la concurrence. Cependant, ce framework présente quelques limites :

— mode de diffusion : notre framework diffuse les mises à jour en mode broadcast. En effet, dans notre framework, pour chaque modification, le simulateur diffuse les opérations

14. le code source est disponible sur GitHub sous la licenceGPL. `https://github.com/score-team/replication-benchmarker`

15. https ://github.com/score-team/replication-benchmarker

générées à *toutes* les autres répliques. Certains systèmes pair à pair utilisent le protocole d'anti-entropie [30]. Ce protocole synchronise les répliques deux à deux,

— adaptation pour state-based : dans le système Git, l'utilisateur propage les modifications sous forme d'états, alors que notre framework est implémenté pour traiter les modifications comme des opérations. De plus, le mode de synchronisation dans Git se fait entre une ou plusieurs répliques alors que notre framework diffuse les opérations à tous les participants. Pour rejouer une édition collaboration réelle asynchrone avec des algorithmes basés sur des états, il faudra modifier le mécanisme de propagation et de gestion de modifications pour traiter les états,

— déconnexion : dans une édition collaborative réelle, la déconnexion des participants est très fréquente. Notre framework suppose que le réseau est fiable et les participants sont connectés tout au long de la session d'édition, ce qui n'est pas réaliste. Il est souhaitable de faire une étude sur la fréquence de déconnexion des usagers durant une session d'édition collaborative et l'ajouter dans les paramètres d'entrée,

— parallélisation : l'exécution de toutes les expérimentations sur une seule machine est limitée par les performances de la machine. Une solution serait d'exécuter chaque algorithme dans une machine différente et les relier à une machine centralisée qui ordonne les opérations. Ainsi, la charge serait distribuée,

— analyse automatique : le résultat des performances est analysé manuellement. Par exemple, pour comprendre le comportement des utilisateurs et réduire le nombre des conflits dans la collaboration asynchrone, nous suivons les modifications manuellement jusqu'à la génération du conflit. Il serait préférable d'intégrer un mécanisme d'analyse automatique.

3.4 Conclusion

Dans ce chapitre, nous avons proposé une méthodologie d'évaluation et un outil correspondant sur lequel nous allons mener nos expérimentations dans les chapitres suivants. Dans un premier temps, nous avons proposé une méthodologie d'extraction de *corpus*. Nous avons implémenté un mécanisme d'extraction de deux type de corpus : synchrone et asynchrone. Dans le cas d'exécution synchrone, le framework donne à l'utilisateur le contrôle total sur les différents paramètres de génération. Dans le cas d'exécution asynchrone, le framework intègre un mécanisme d'extraction de traces à partir de l'historique des projets développés sous Git. Ensuite, nous avons proposé un *simulateur* pour simuler une édition collaborative en utilisant le corpus récupéré dans la première étape. Enfin, nous avons proposé des *mesures* de performances.

L'outil que nous venons d'introduire est un outil d'évaluation efficace comme nous le verrons à l'occasion des résultats obtenus par l'implémentation. Notre framework est open-source, générique et extensible. En effet, la conception de l'outil est effectuée de façon à supporter de multiple structures de données répliquées. Il permet la reproduction des résultats tant que les paramètres d'entrée sont identiques.

Notre méthode et son implémentation répondent aux différents critères d'expérimentations vues dans le chapitre 1.2.3. Dans la section suivante, nous nous intéressons à l'évaluation des différentes approches optimistes en utilisant l'outil proposé.

Chapitre 4

Évaluation des performances

Sommaire

Le but de ce chapitre est de mener une étude approfondie sur la complexité et les performances des algorithmes de réplication optimiste. Cette étude a pour objectif de mieux comprendre le comportement de ces algorithmes et de pouvoir choisir, parmi ceux existant, le mieux adapté à chaque situation.

Nous allons présenter l'évaluation des algorithmes optimistes suivant la méthodologie décrite précédemment, en utilisant notre framework. Ce dernier exécute les différents algorithmes sur des traces synchrones et asynchrones. Nous commencerons par présenter les corpus récupérés dans chaque type de collaboration. Nous analyserons ensuite les performances des différents algorithmes optimistes sur ces traces d'exécution. Nous discuterons ainsi les points forts et faibles de chaque algorithme. Enfin, nous proposerons des solutions afin d'améliorer les performances des algorithmes et d'améliorer la satisfaction de l'utilisateur final suivant les critères d'évaluation vus dans le chapitre 1.2.3.

4.1 Performance mode synchrone

Dans cette partie, nous présentons la méthode qui nous a permis de récupérer les traces d'exécution *synchrones*. Sur ces traces, nous analysons les performances des algorithmes de réplication

des différents types de données répliqués : ensemble, texte et arbre.

4.1.1 Corpus

Comme nous l'avons décrit dans la section précédente (section 3.2.2), il n'existe pas à notre connaissance de trace d'édition collaborative synchrone publique. Pour récupérer des traces d'exécution d'édition collaborative synchrones, nous avons modifié le code source d'un éditeur collaboratif TeamEdit et nous avons mené deux expériences :

1. première expérience (rapport) : elle a été réalisée avec 13 étudiants, divisés en trois groupes, deux groupes composés de 4 étudiants et un groupe composé de 5 étudiants. Chaque groupe a été invité à rédiger en collaboration un rapport de projet. Chaque membre du groupe travaille sur le rapport à partir d'une station de travail, et n'était pas autorisé à utiliser d'autre moyen de communication que TeamEdit. La collaboration a duré une heure et demie. Avant l'expérience, nous avons indiqué aux étudiants qu'ils seront évalués non seulement en fonction du contenu du rapport mais aussi de la taille de leur contribution,
2. seconde expérience (série) : elle a duré environ une heure et demie, dans une séance hors cours, nous avons demandé à 18 étudiants volontaires de regarder un épisode de la série "The Big Bang Theory " et de produire une transcription de l'épisode en la regardant. Les étudiants ont été divisés en neuf groupes de deux, et chaque groupe a été invité à faire une transcription d'un certain acteur ou de décrire l'environnement et les actions qui se sont passées pendant l'épisode. Au cours de cette collaboration, les étudiants ont été autorisés à communiquer directement.

   Pour récupérer plus de traces, nous avons reproduit l'expérience mais nous avons attribué des rôles différents à chaque groupe.

Au cours de chaque expérience, certains étudiants se sont déconnectés puis reconnectés à la session sans que nous leur demandions. Cependant, les utilisateurs pouvait travailler sur leur copie locale et synchronisent leurs modifications une fois connectés. Ceci est très fréquent dans une édition collaborative réelle et peut influencer les performances de l'algorithme de réplication utilisé. En effet, lorsqu'un participant se reconnecte, toutes ses modifications doivent être exécutées par les autres utilisateurs. Cette procédure d'intégration peut diminuer la performance de l'algorithme, si le nombre de modifications est important.

Nous avons mené deux expériences différentes non seulement pour collecter de large traces mais aussi pour représenter deux types de collaboration différente. Le tableau 4.1 présente les caractéristiques des modifications qui varient selon le type de la collaboration. Cette variation nous permet de confronter les algorithmes dans des situations différentes.

Projet / Caractéristique	Rapport	SérieTV
Moy. pourcentage d'insertions	88.26%	93.23%
Moy. nombre de modifications	11993.00	9435.00
Moy. nombre de bloc d'insertions	112.66	174.00
Moy. nombre de bloc de suppressions	97.33	38.00
Moy. nombre de participants	4.33	18.00
Moy. taille des blocs	158.89	34.45

Table 4.1 – Caractéristiques des traces synchrones collectées.

Maintenant que nous avons récupéré les caractéristiques des traces réelles, nous allons produire une édition collaborative plus large en se basant sur ces caractéristiques. Comme nous avons décrit dans le chapitre précédent, notre framework reproduit une édition collaborative en utilisant un générateur de traces. Le framework prend en entrée un ensemble de paramètres qui va être utilisé par le générateur. Les paramètres contrôlés par notre framework dépendent du type des données que nous allons évaluer. Pour les documents textuels linéaires, les paramètres contrôlés sont les suivants :

a) la proportion de modifications de type insertion, *b*) la proportion de modifications de type bloc (copier/coller), *c*) le nombre de modifications à générer, *d*) le nombre de répliques, *e*) la probabilité que l'outil attribue à une réplique à chaque itération pour produire une modification, *f*) la latence réseau, *g*) la taille moyenne des copier-coller,

De plus, pour les types de données "ensemble" un paramètre est contrôlé qui est la taille du vocabulaire. Ce paramètre représente le nombre d'éléments qui peuvent être ajoutés dans un ensemble. Aussi, pour les types de données "arbre" une probabilité de génération de noeuds fils est contrôlé par l'utilisateur. Cette proportion permet à l'algorithme de décider s'il insère l'élément sous le même noeud ou créer un nouveau noeud fils.

Nous ajustons ensuite notre outil avec les caractéristiques extraites des traces réelles et nous lançons l'exécution des différents algorithmes de réplication.

4.1.2 Expérimentations & résultats

Pour évaluer les performances des algorithmes, nous avons produit à travers le framework un ensemble de traces d'exécution. Dans chaque expérience, le framework contrôle une caractéristique et garde les autres statiques, similaires aux traces réelles récupérées à l'aide de TeamEdit. Cependant, ajuster l'outil avec exactement les mêmes valeurs peut masquer l'influence de certaines caractéristiques sur les performances des algorithmes. Par exemple, la proportion de blocs et d'insertions sont respectivement aux alentours de 1% et 90%. Utiliser ces valeurs rend l'observation de leur impact sur les performances des algorithmes difficile. Pour cette raison, nous avons modifié légèrement ces valeurs tout en restant sur une édition collaborative réelle. Pour simplifier l'analyse des algorithmes, nous ne modifions pas la probabilité de génération et le délai de réception. Ces caractéristiques affectent le degré de la concurrence qui est également affecté par le nombre de répliques.

Le tableau 4.2 présente cinq expériences différentes :

1. Modifications : nous mesurons les performances des algorithmes de réplication en fonction du nombre de modifications. Nous incrémentons le nombre de modifications par 10 000 à chaque étape, allant de 20 000 jusqu'à 100 000 modifications. Et nous gardons les autres paramètres stables tels que le nombre de répliques à 10, la proportion de blocs à 15%, la proportion d'insertion à 80%, etc ;

2. Insertions : nous mesurons les performances des algorithmes de réplication en fonction de la proportion d'insertion. Nous incrémentons le pourcentage d'insertion par 5% à chaque étape, allant de 50% d'insertions jusqu'à 100%. Et nous gardons les autres paramètres stables tels que le nombre de modifications à 10 000, nombre de répliques à 10, la proportion de blocs à 15%, etc ;

3. Bloc de texte : nous mesurons les performances des algorithmes de réplication en fonction de la proportion de bloc. Nous incrémentons le pourcentage de bloc par 10% à chaque étape, allant de 0% bloc jusqu'à 100%. Et nous gardons les autres paramètres stables ;

4. Vocabulaire : le vocabulaire est le nombre d'éléments que l'utilisateur peut ajouter dans l'ensemble. Cette expérimentation est spécifique au type de données "ensemble". Nous mesurons dans cette expérimentation les performances des algorithmes en fonction de la taille du vocabulaire. Dans chaque étape nous multiplions par deux la taille du vocabulaire, allant de 10 éléments jusqu'à 10 240 éléments. Et nous gardons les autres paramètres stables ;
5. Répliques : chaque réplique représente un utilisateur. Dans cette expérience, nous mesurons les performances des algorithmes de réplication en fonction de nombre d'utilisateurs. Nous ajoutons dans chaque étape cinq utilisateurs, allant de 2 jusqu'à 50 utilisateurs. Et nous gardons les autres paramètres stables.

Expérience	Opérations	Insertions	Bloc texte	vocabulaire ensemble	Répliques
Nbr. opérations	20 000 $\xrightarrow{+10000}$ 100 000	10 000	10 000	10 000	10 000
Répliques	10	10	10	10	2 $\xrightarrow{+5}$ 50
Bloc %	15%	15%	0% $\xrightarrow{+10\%}$ 100%	-	15%
Insertions %	80%	50% $\xrightarrow{+5\%}$ 100%	80%	80	80%
taille de vocab.	-	-	-	10 $\xrightarrow{\times 2}$ 10240	-
Moy taille bloc	100	100	100	-	100
Prob. Génération	0.1	0.1	0.1	0.1	0.1
délai	5	5	5	5	5

Table 4.2 – Expérience pour les types de données : Texte/Ensemble

Dans ce qui suit, nous distinguons le temps d'exécution *local* –le traitement d'une modification effectuée par l'utilisateur sur la copie locale– et le temps d'exécution en *remote* –le temps d'exécution sur la copie distante–. La latence réseau n'est pas prise en considération car elle est indépendante du mécanisme de traitement.

Les expériences ont été réalisées sur des machines virtuelles de Amazon –Elastic Compute Cloud (EC2)–. Nous exécutons chaque expérience sur une machine virtuelle Intel (R) Xeon (R) 5160 de processeur d'un noyau (4096K cache, 2.27GHz, 2266,746 MHz FSB, 3,75 Gio de mémoire), sur GNU/Linux ubuntu 12.04.

4.1.3 Type de données : Ensemble

Dans les sections 2.3.4 et 3.2.1, nous avons décrit certains algorithmes de réplication pour les "ensembles". Nous avons vu qu'à l'exception de OR-set et OptORSet, les différents algorithmes ont des sémantiques différentes. C'est à dire que les différents algorithmes possèdent une manière différente de représenter les modifications. Par exemple LWW-Set incrémente une valeur d'une horloge logique à chaque insertion d'un élément alors que OR-Set génère un identifiant (tag) pour chaque élément. De ce fait, pour pouvoir comparer les différentes approches, nous exécutons chaque algorithme cent fois sur des données simulées (Sauf pour OR-set et OptORSet sur les mêmes données puisqu'ils ont la même sémantique). Dans ce qui suit, nous présentons les résultats de deux expériences : en changeant la taille de l'ensemble de données et la proportion des insertions. Nous mesurons ensuite le temps d'exécution *local*, le temps d'intégration des opérations (*remote*), ainsi que l'occupation mémoire.

Nous avons remarqué dans nos expériences que la proportion d'insertions, le nombre d'opérations et de répliques n'ont pas une influence directe sur le temps d'exécution des algorithmes, car on obtient des comportements chaotiques. Par contre, la taille du vocabulaire influence plus

clairement les algorithmes. Les figures 4.1 et 4.2 présentent respectivement le temps d'exécution moyen pour chaque algorithme en local et en remote en fonction de la taille du vocabulaire. Les résultats de temps d'exécution indiquent que :

— Les algorithmes OptORSet et ORSet obtiennent les pires performances en temps d'exécution en local et en remote. En local, les deux algorithmes consomment du temps pour générer les *tags* et identifier les éléments. Lors de l'intégration, les deux algorithmes nécessitent la récupération des *tags* de l'élément à ajouter (s'il existe) ou à supprimer. Si l'opération est une suppression, tous les *tags* de l'opération sont supprimés. Si l'opération est une insertion, le *tag* du nouvel élément est ajouté dans l'ensemble des *tags*,

— en local, OptORSet est moins performant que ORSet en temps d'exécution. En effet, OptORSet effectue un traitement de plus car il met à jour son vecteur d'horloge, alors que ORSet n'en a pas,

— En remote, le comportement des algorithmes semble stable, et contrairement à l'exécution locale, OptORSet est plus performant que ORSet. En effet, l'algorithme OptORSet effectue moins de traitement, car le nombre de *tags* dans OptORSet est inférieur au nombre de *tags* dans ORSet,

— les algorithmes LWWSet et CounterSet surpassent ORSet et OptORSet en local et en remote. En effet, LWWSet et CounterSet ne génèrent pas de *tag* en locale, et ne manipule pas des *tags* en remote. Les deux algorithmes incrémentent seulement une valeur d'une horloge logique ou un compteur,

— que ce soit en remote ou en local, LWWSet consomme un peu plus de temps que CounterSet. Cela est due à la gestion des ensembles des éléments. En effet, LWWSet manipule deux ensembles A pour les ajouts et R pour les suppressions. Alors que l'algorithme CouterSet ne manipule qu'un seul ensemble. Plus la taille du vocabulaire est grande et plus la taille des ensembles devient grande.

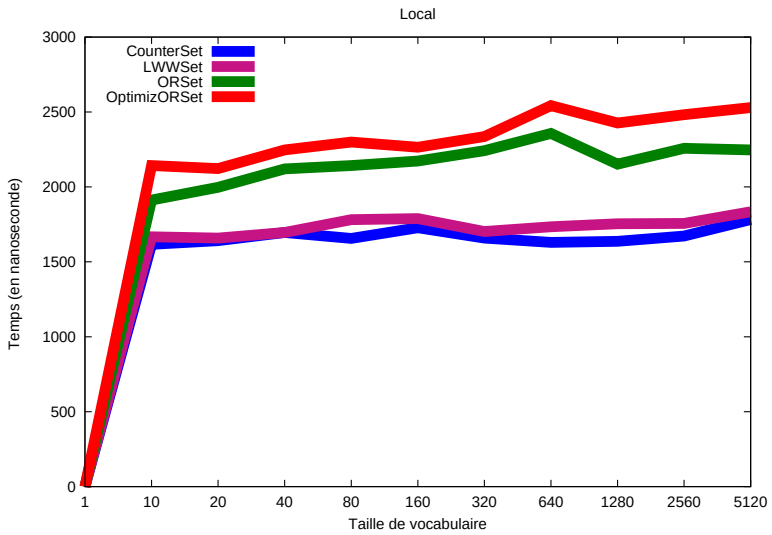


FIGURE 4.1 – Temps d'exécution local en fonction de la taille du vocabulaire

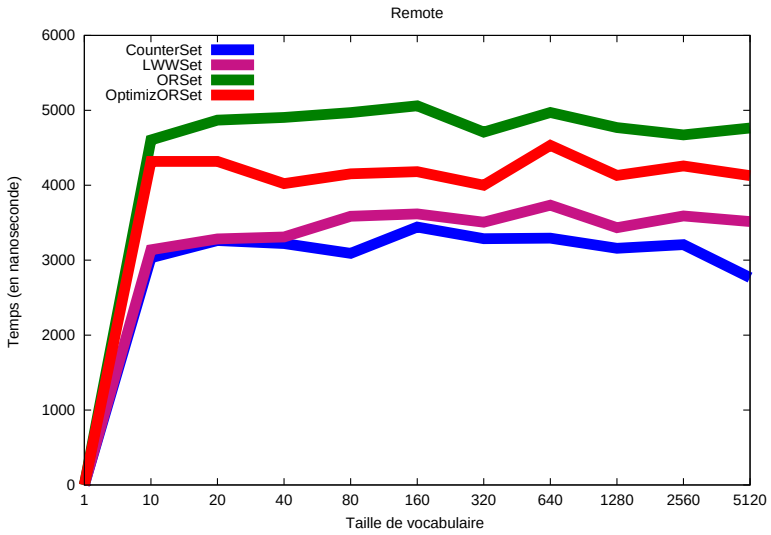


FIGURE 4.2 – Temps d'intégration en fonction de la taille du vocabulaire

Dans les figures 4.3 et 4.4 nous présentons respectivement l'espace mémoire occupé pour chaque algorithme en fonction de l'insertion et de la taille de l'ensemble de données. Les performances obtenues indiquent que :

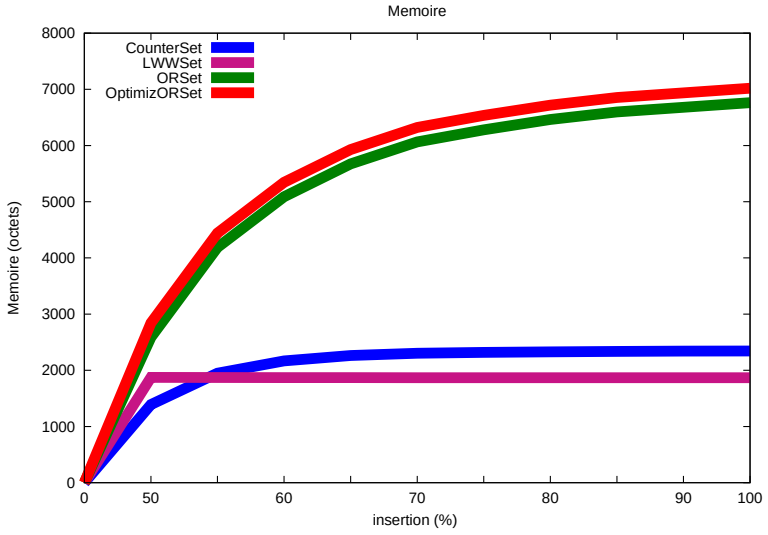


FIGURE 4.3 – Mémoire occupée par % insertion

— OptORSet ne réduit pas la taille de la mémoire. En effet, la taille du vecteur utilisé pour détecter l'ordre partiel des éléments remplace l'espace mémoire réduit par l'algorithme,

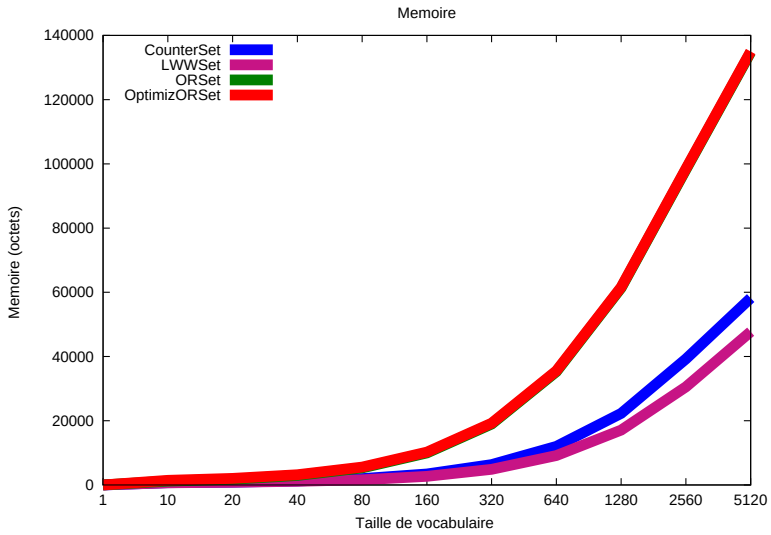


FIGURE 4.4 – Mémoire occupée par taille du vocabulaire

— La mémoire utilisée par les algorithmes LWWSet et CounterSet reste stable. Quel que soit le pourcentage d'insertion, ces algorithmes gardent tous les éléments (insérés et supprimés) dans leur modèle,

— Plus le pourcentage d'insertion augmente et plus les répliques occupent de mémoire. Progressivement, les algorithmes deviennent stables. Ceci est expliqué par le fait que l'élément généré est déjà présent dans l'ensemble des éléments. Lorsque le pourcentage d'insertion atteint les 100%, tous les éléments sont déjà présents dans l'ensemble. Par conséquent, les algorithmes atteignent le maximum en terme d'espace mémoire occupé,

— L'occupation mémoire est proportionnelle à la taille de l'ensemble des données. En effet, le nombre d'éléments dans l'ensemble de la réplique dépend de la taille des données utilisées.

Pour valider notre analyse et s'assurer que cette différence est significative, nous avons effectué une étude statistique. Le résultat de cette étude est présenté dans le tableau 4.3. Nous avons utilisé la technique ANOVA (ANalysis Of VAriance) [16]. Après avoir exécuté dix fois les quatre algorithmes, nous avons récupéré leur temps d'exécution moyen ainsi que l'écart type. Á partir de ces données, nous avons calculé la valeur p – valeur de signification –. Les expériences reposent sur l'hypothèse que chaque exécution est indépendante des autres exécutions, et que la distribution des éléments suit une loi normale. La valeur p obtenue démontre clairement que l'impact de la taille du vocabulaire sur le temps d'exécution et l'espace mémoire est très significatif. En effet, pour le temps d'exécution, $p = 0,00$ et pour la mémoire $p = 0,03$. Dans les deux expériences, $p < 0,05$.

4.1.4 Type de données : Texte

Nous mesurons dans ce qui suit les performances des algorithmes optimistes destinés à l'édition collaborative. Selon le type du corpus, nous distinguons deux types d'évaluation :

16. le résultat est significatif si $p - value$ <.05

Algorithme	local			remote		
	moy	médiane	écart-type	moy	médiane	écart-type
Counter	1671,96	1657,77	50,84	3176,62	3214,90	183,56
LWW	1737,22	1743,99	56,43	3471,79	3511,25	180,93
OR	2160,10	2162,21	130,18	4829,82	4819,92	148,44
OptimizOR	2339,40	2317,85	151,64	4204,28	4166,87	158,41
p-value	0.00			0.00		

Algorithme	mémoire		
	moy	médiane	écart-type
Counter	14627,68	4876,62	19678,27
LWW	11636,89	3778.01	15874,52
OR	36972,58	14468,78	46711,29
OptimizOR	37228,71	14725,03	46711,43
p-value	0.038		

Table 4.3 – Moyenne, médiane et écart-type pour le temps d'exécution et mémoire.

1. mesurer les performances des algorithmes sur des données simulées inspirées des traces *synchrones* réelles ;
2. mesurer les performances des algorithmes sur des traces *asynchrones* réelles obtenues à partir de GitHub [17].

Nous mesurons les performances de six algorithmes : SOCT2/TTF, WOOTO, RGA, TreeDoc, Logoot, and LogootSplit.

Certains algorithmes tels que Logoot et LogootSplit se basent sur des méthodes aléatoires pour générer des opérations. En plus, l'état du processeur au moment de l'exécution peut être influencé par d'autres applications. Pour ces raisons, nous exécutons chaque algorithme cinq fois et nous calculons la moyenne.

Pour savoir quel paramètre influence le plus les algorithmes, nous effectuons quatre expériences, et nous contrôlons dans chaque expérience une caractéristique.

Cependant, pour bien comprendre le comportement des algorithmes durant l'expérience et interpréter correctement le résultat, nous menons une étude théorique avant de présenter les résultats.

Complexité temporelle La complexité temporelle au pire cas des algorithmes est présentée dans le tableau 4.4. La complexité temporelle moyenne est présentée dans le tableau 4.5.

Pendant l'exécution locale, les algorithmes traitent soit les opérations d'insertion (*ins*) ou de suppression (*del*). Ces opérations peuvent être une séquence continue d'éléments (par exemple, couper/coller un bloc de texte). Les algorithmes doivent trouver dans leur modèle la position correspondante. Lors de l'exécution locale, les opérations qui seront envoyées à d'autres répliques sont générés. Au cours de l'exécution *remote*, le traitement des opérations est spécifique à chaque algorithme. Tout dépend de l'algorithme, l'opération *remote* peut être une opération avec un seul élément ou une suite d'éléments. On note par :

— R nombre de répliques qui ont participé à la modification du document texte,

— H le nombre total de modifications,

— N le nombre total d'éléments insérés (y compris les pierres tombales)), au pire cas $N = H$,

17. web-hosting : https ://github.com/

— b le nombre de *buckets* utilisés dans la table de hachage,

— c le nombre moyen d'opérations concurrentes,[18] au pire cas $c = H$,

— n la taille du document observé (pas de pierre tombale),

— d le nombre moyen d'éléments (pierre tombale et insertion concurrente) qui se trouvent entre deux éléments successifs dans le document – $d = N/n + c$,

— k la taille moyenne des identifiants Logoot. Dans le meilleur cas $k = log(N)$ et dans le pire cas $k = N$,

— p la longueur moyenne du chemin TreeDoc. Dans le meilleur cas $p = log(N)$ et dans le pire cas $p = N$,

— l le nombre moyen d'éléments qui se trouve dans une opération (bloc), au pire cas $l = 1$ ou $l = N$ selon l'algorithme s'il gère les blocs ou pas.

Algorithme	local		remote	
	ins	del	ins	del
SOCT2/TTF	$\mathcal{O}(H.R)$	$\mathcal{O}(H.R)$	$\mathcal{O}(H^3)$	$\mathcal{O}(H^3)$
WOOT	$\mathcal{O}(H^4)$	$\mathcal{O}(H)$	$\mathcal{O}(H^4)$	$\mathcal{O}(H)$
WOOTO	$\mathcal{O}(H^3)$	$\mathcal{O}(H)$	$\mathcal{O}(H^3)$	$\mathcal{O}(H^2)$
RGA	$\mathcal{O}(H)$	$\mathcal{O}(H)$	$\mathcal{O}(H^2)$	$\mathcal{O}(H^2)$
Logoot	$\mathcal{O}(H^2)$	$\mathcal{O}(H)$	$\mathcal{O}(H^2.log(H))$	$\mathcal{O}(H^2.log(H))$
LogootSplit	$\mathcal{O}(H)$	$\mathcal{O}(H)$	$\mathcal{O}(H.log(H))$	$\mathcal{O}(H.log(H))$
TreeDoc	$\mathcal{O}(H)$	$\mathcal{O}(H^2)$	$\mathcal{O}(H)$	$\mathcal{O}(H^2)$

Table 4.4 – Complexité temporelle dans le pire cas

Algorithme	moy. locale		moy. remote	
	ins	del	ins	del
SOCT2/TTF	$\mathcal{O}(N + l.R)$	$\mathcal{O}(N + l.R)$	$\mathcal{O}(l.H.c)$	$\mathcal{O}(l.H.c)$
WOOT	$\mathcal{O}(l.N.d^2)$	$\mathcal{O}(N + l)$	$\mathcal{O}(l.N.d^2)$	$\mathcal{O}(l.N)$
WOOTO	$\mathcal{O}(N + l.d^2)$	$\mathcal{O}(N + l)$	$\mathcal{O}(l.(N + d^2))$	$\mathcal{O}(l.N)$
RGA	$\mathcal{O}(N + l)$	$\mathcal{O}(N + l)$	$\mathcal{O}(l.(N/b + d))$	$\mathcal{O}(l.(1 + N/b))$
Logoot	$\mathcal{O}(k.l)$	$\mathcal{O}(l)$	$\mathcal{O}(l.k.log(n))$	$\mathcal{O}(l.k.log(n))$
LogootSplit	$\mathcal{O}(n/l + k)$	$\mathcal{O}(n/l + k)$	$\mathcal{O}(k.log(n/l))$	$\mathcal{O}(k.log(n/l))$
TreeDoc	$\mathcal{O}(c.p + l)$	$\mathcal{O}(l.p)$	$\mathcal{O}(c.p + l)$	$\mathcal{O}(l.p)$

Table 4.5 – Complexité temporelle en moyenne

En moyenne la complexité de chaque algorithme dépend de l, le nombre moyen d'éléments présents dans une modification effectuée par l'utilisateur. Toutefois, si l'opération *remote* est traitée indépendamment, l'exécution d'une opération *locale* nécessite un traitement spécifique avant sa diffusion.

— L'approche SOCT2/TTF nécessite lors de l'exécution locale de compter le nombre de pierres tombales $\mathcal{O}(N)$ et de générer l vecteurs d'horloge $\mathcal{O}(R)$. L'exécution des l opérations reçues nécessite des transformations de $\mathcal{O}(c)$ opérations concurrentes sur tout l'historique $\mathcal{O}(H)$,

— L'insertion dans les algorithmes WOOTs est influencée par l'algorithme de linéarisation appliqué sur des éléments concurrents et les pierres tombales $\mathcal{O}(d^2)$, et au pire cas $\mathcal{O}(H^3)$ [115]. L'exécution locale des opérations *del* est influencée seulement par le calcul de la position

18. c dépend de la latence réseau et la fréquence de génération d'opérations.

dans le modèle (parcourir le document avec les pierres tombales) $\mathcal{O}(N)$, plus le temps de génération des l opérations à envoyer aux autres répliques. L'exécution des l opérations *del* en remote nécessite l recherches linéaires dans $\mathcal{O}(N)$,

— L'algorithme RGA nécessite le parcours de son modèle pour calculer les pierres tombales $\mathcal{O}(N)$ durant l'exécution locale, et produire l opérations remote. À la réception de ces l opérations, RGA parcourt la table de hachage pour chaque élément qui est $\mathcal{O}(H)$ dans le pire cas et $\mathcal{O}(1 + N/b)$ en moyenne,

— les algorithmes Logoots utilisent des identifiants de taille $\mathcal{O}(k)$ en moyenne et $\mathcal{O}(H)$ au pire cas. En local, le temps d'exécution d'une opération est constant pour Logoot. Pour une insertion, il génère l nouveaux identifiants de taille k en moyenne. Une suppression nécessite le parcours de l identifiants en moyenne pour arriver à la position de la suppression. L'exécution des opérations en remote est influencée par la recherche dichotomique, $\mathcal{O}(k.log(n))$ en moyenne [120]. LogootSplit identifie chaque bloc avec un seul identifiant, en local il génère n/l nouveau identifiants. Le modèle de LogootSplit contient n/l blocs, tandis que la complexité en remote est influencée par la recherche dichotomique sur des identifiants blocs, $\mathcal{O}(k.log(n/l))$ en moyenne,

— L'algorithme Treedoc parcourt un chemin de taille p pour trouver la position correcte de l'identifiant lors de l'intégration. Si le noeud au bout de ce chemin est un noeud-majeur, une recherche linéaire est effectuée. Donc, il existe $c.p$ noeuds en moyenne traversés. Une opération *ins* d'un bloc d'élément produit une seule opération remote, alors qu'une suppression d'un bloc d'éléments produit l opérations remote.

Complexité spatiale Nous étudions dans ce qui suit, l'espace mémoire occupé par les différentes approches et la taille des messages générés et diffusés dans le réseau.

Algorithme	modèle		message	
	worst	avg.	worst	avg.
SOCT2/TTF	$\mathcal{O}(H.R)$	$\mathcal{O}(H.R)$	$\mathcal{O}(H.R)$	$\mathcal{O}(l.R)$
WOOTx	$\mathcal{O}(H)$	$\mathcal{O}(N)$	$\mathcal{O}(H)$	$\mathcal{O}(l)$
RGA	$\mathcal{O}(H)$	$\mathcal{O}(N)$	$\mathcal{O}(H)$	$\mathcal{O}(l)$
Logoot	$\mathcal{O}(H^2)$	$\mathcal{O}(k.n)$	$\mathcal{O}(H^2)$	$\mathcal{O}(l.k)$
LogootSplit	$\mathcal{O}(H^2)$	$\mathcal{O}(k.n/l)$	$\mathcal{O}(H)$	$\mathcal{O}(k+l)\|O(k.n/l^2)$
TreeDoc	$\mathcal{O}(H)$	$\mathcal{O}(n)$	$\mathcal{O}(H^2)$	$\mathcal{O}(p+l)\|O(p.l)$

Table 4.6 – complexité spatiale

— SOCT2/TTF sauvegarde tout l'historique des opérations $\mathcal{O}(H)$, chacune avec son vecteur d'horloge $\mathcal{O}(R)$. Les l opérations envoyées contiennent un vecteur d'horloge,

— Les approches WOOTs et RGA stockent $\mathcal{O}(N)$ éléments avec leurs identifiants de taille fixe. Une modification effectuée par l'utilisateur produit l opérations remote,

— Logoot stocke $\mathcal{O}(n)$ éléments avec leurs identifiants de taille $\mathcal{O}(k)$. Un utilisateur produit l opérations. Chaque opération avec un identifiant transmis dans le réseau,

— LogootSplit stocke seulement $\mathcal{O}(n/l)$ éléments. Une modification *insert* effectuée par l'utilisateur produit une opération de taille $\mathcal{O}(k+l)$ de taille k. Une modification de suppression produit en moyenne n/l^2 opérations remote. Ces opérations sont diffusées avec leurs identifiants dans le réseau,

— Treedoc stocke un arbre de $\mathcal{O}(n)$ éléments. Une modification d'insertion produit une seule opération de taille $\mathcal{O}(p+l)$. Alors qu'une modification de suppression produit l opérations remote diffusées dans le réseau avec leurs identifiants de taille in $\mathcal{O}(p)$.

Discussion L'étude théorique montre qu'il n'y a pas un algorithme qui surpasse tous les autres en performance, que ce soit dans le pire des cas ou en moyenne. Cependant, RGA et TreeDoc semblent de bons candidats, car la table de hachage utilisée par RGA accélère le traitement des opérations remote et la structure d'arbre de Treedoc rend la recherche des éléments plus performante. Puisque le processus de génération et d'intégration des opérations ainsi que la structure des données diffèrent d'un algorithme à un autre, les différentes approches doivent être comparées les unes avec les autres.

Face au comportement des utilisateurs, les paramètres réseaux, l'architecture déployée, etc, l'analyse théorique n'est pas suffisante pour classer les différents algorithmes et prédire leur comportement, d'où la nécessité d'effectuer une évaluation expérimentale.

Dans ce qui suit, nous présentons le résultat d'évaluation de performance des algorithmes sur des traces synchrones.

Temps d'intégration

Les figures 4.5 et 4.6 présentent respectivement le temps d'intégration des différentes approches en fonction de nombre d'opérations et de nombre de répliques.

L'axe vertical dans la figure présente le temps moyen nécessaire pour l'exécution d'une opération remote correspondant à une opération effectuée par l'utilisateur. Cet axe utilise une échelle logarithmique. L'axe horizontal présente une valeur d'un paramètre expérimental.

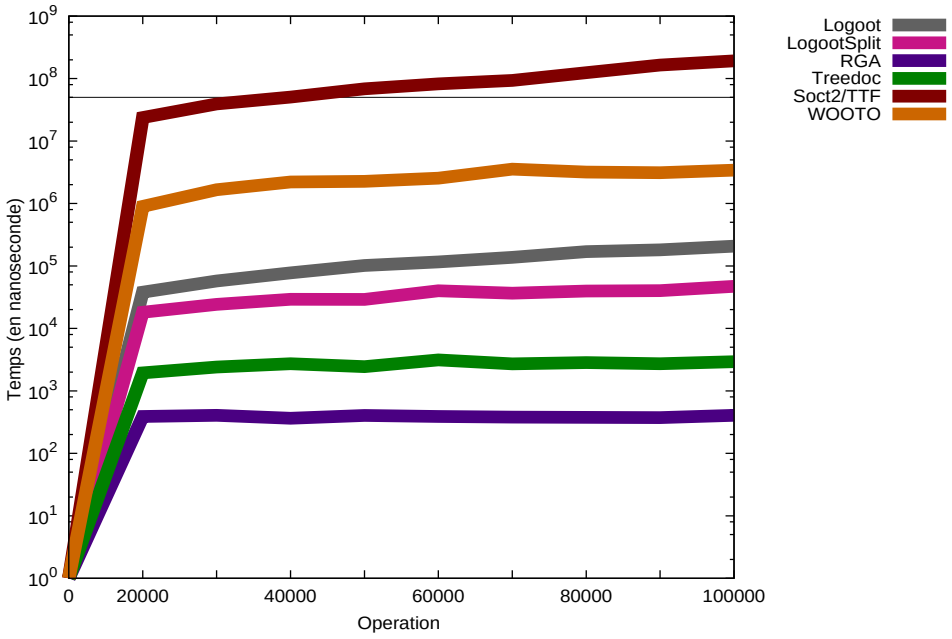


FIGURE 4.5 – Temps d'intégration par nombre de d'opérations

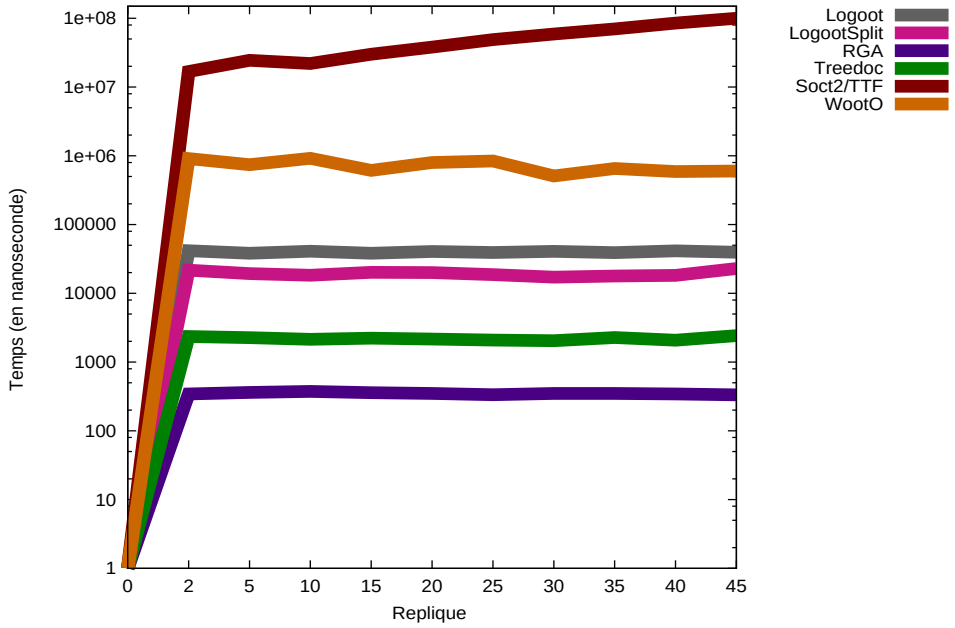


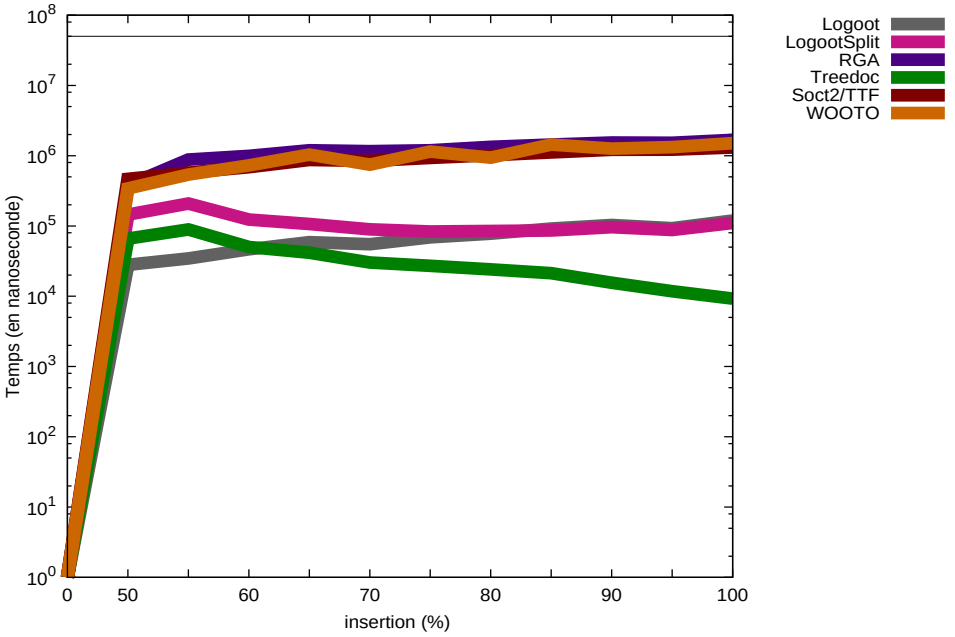
FIGURE 4.6 – Temps d'intégration par nombre de répliques

Temps par opérations L'algorithme SOCT2/TTF semble le moins performant. Plus le nombre d'opérations augmente et plus l'algorithme SOCT2/TTF se dégrade. Cette dégradation revient au nombre de transformations qui croît avec le nombre d'opérations concurrentes. L'algorithme WOOTO perd aussi en performance à cause de la taille de son modèle. WOOTO stocke tous les éléments y compris les éléments supprimés. Par conséquent, le temps de parcours du modèle pour trouver la position correcte de l'élément consomme du temps. Les performances de Logoot se dégradent aussi mais moins que SOCT2/TTF et WOOTO. Cela est dû à la croissance de la taille des identifiants Logoot. Comme indiqué dans l'étude théorique, la complexité temporelle en moyenne de Logoot est influencée par la taille des identifiants. Cette taille est proportionnelle au nombre d'insertion. Ce dernier dépend du nombre d'opérations. Les identifiants affectent légèrement Treedoc et LogootSplit, puisque les deux approches se basent sur les blocs et par conséquent la taille du document est moins longue pour trouver la position correspondante à l'opération reçue.

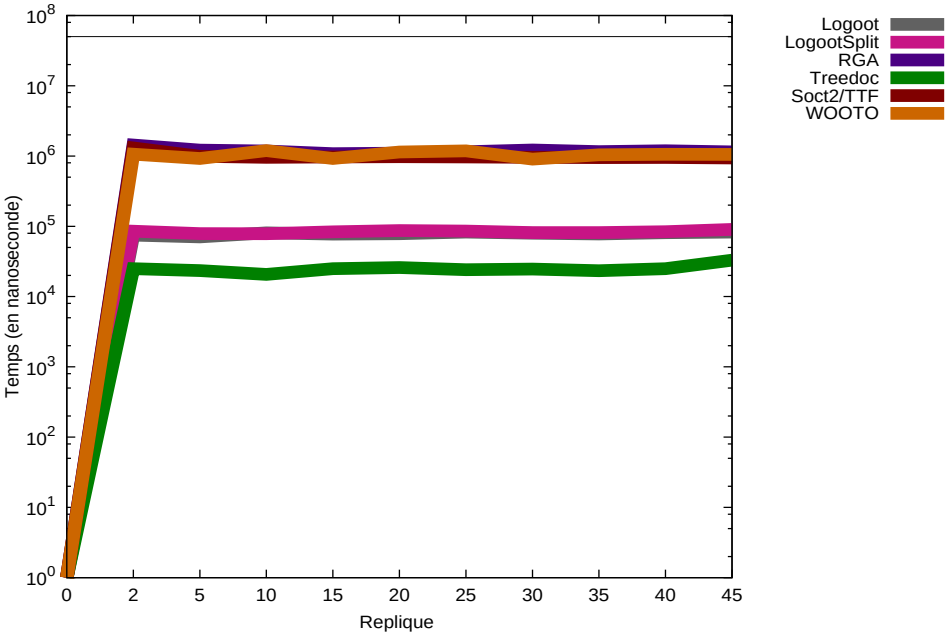
Temps par répliques Nous pouvons observer que les algorithmes CRDT ont un comportement très stable lorsque le nombre de répliques augmente. Les performances de SOCT2/TTF se dégradent mais moins vite que le nombre de répliques. La complexité temporelle de SOCT2/TTF ne dépend pas de R, car l'algorithme ne traverse pas les vecteurs d'horloge durant l'intégration des opérations ; Mais elle dépend de c qui est proportionnelle à R. Donc, Cette perte est due à l'augmentation de la concurrence c et le nombre de transformations.

Temps de génération

Les figures 4.7a et 4.7b présentent respectivement le temps de génération des algorithmes



(a) Temps de génération par % d'insertion



(b) Temps de génération par nombre de répliques

FIGURE 4.7 – Temps de génération

optimistes en fonction de la proportion d'insertions et du nombre de répliques.

Temps en fonction de % d'insertion Les algorithmes utilisant les pierres tombales (RGA, WOOTO et TTF) perdent en performances lorsque le pourcentage d'insertion augmente. En effet, les algorithmes doivent traverser leur modèle – l'équivalent de tout l'historique, qui devient assez long– pour chercher la position correspondante à une opération. Logoot obtient une mauvaise performance. Ceci est dû à la taille des identifiants qui ne s'arrête pas de grandir. Treedoc et LogootSplit gagnent en performances puisque dans les deux algorithmes, une opération d'insertion de bloc est traitée en tant qu'une seule opération contrairement à une suppression.

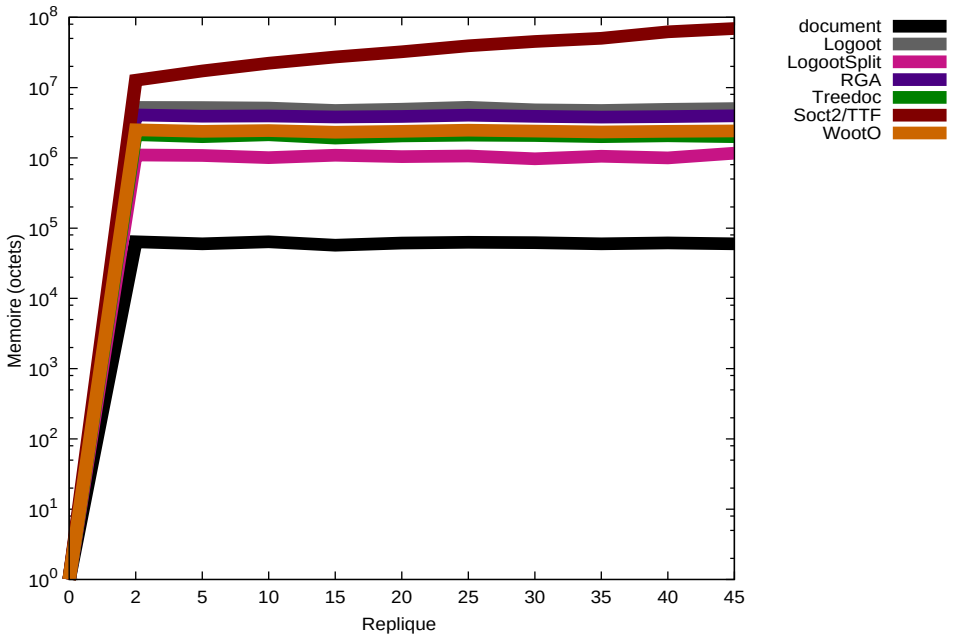


FIGURE 4.8 – Occupation mémoire en fonction de nombre de répliques

Temps en fonction des répliques Comme pour l'intégration, le temps de génération pour TreeDoc, Logoot, LogootSplit, RGA et WOOTO restent stable en fonction de nombre de répliques. Localement, la génération des opérations pour ces algorithmes est indépendante du nombre des utilisateurs. Contrairement à l'intégration, SOCT2 reste stable car il n'effectue aucune transformation pour générer une opération.

L'empreinte mémoire Nous présentons dans les figure 4.8 et 4.9 la mémoire occupée requis par une réplique lors de l'expérience. La taille de la mémoire est calculée en utilisant la sérialisation Java. Nous calculons la taille du document observé par les utilisateurs sans les métadonnées et la taille de la réplique qui inclut les métadonnées utilisées par l'algorithme de réplication. La différence entre la taille du document et la taille de la réplique indique la surcharge d'un algorithme.

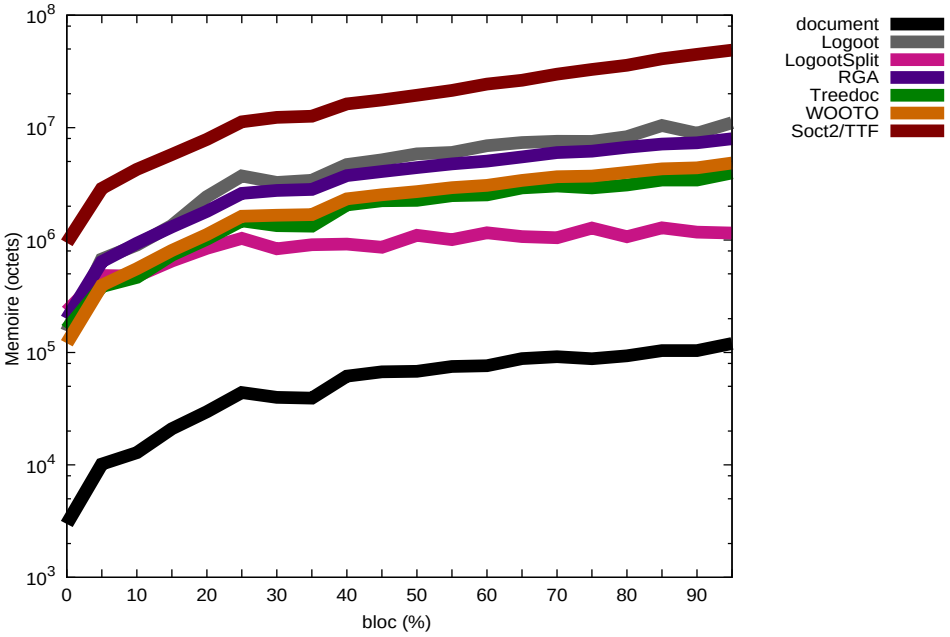


FIGURE 4.9 – Occupation mémoire par % de bloc

À l'exception de SOCT2/TTF qui utilise les vecteurs d'horloge pour détecter la concurrence, le nombre de répliques n'influence pas la mémoire occupée des autres algorithmes. Leur comportement reste très stable durant toute l'expérience. Le pire algorithme est SOCT2/TTF qui nécessite des pierres tombales et l'historique des opérations. Plus surprenant, et contrairement aux expérimentations menées dans [119] utilisant une granularité de ligne, le deuxième pire algorithme est Logoot. Ce dernier ne nécessite pas de pierres tombales, mais des identifiants de taille importante sont générés pour chaque caractère. Logoot nécessite une surcharge cent fois plus grande que la taille du document. La meilleure performance est obtenue par LogootSplit, suivi par TreeDoc, WOOTO et RGA. Malgré la meilleure performance, LogootSplit nécessite une surcharge de vingt fois la taille du document.

La figure 4.9, présente la taille de la mémoire occupée par une réplique en fonction de la taille des blocs. À l'exception de LogootSplit, la mémoire occupée grossit pour tous les algorithmes ainsi que pour le document, puisque plus de bloc implique plus de caractères. Le classement entre les algorithmes reste le même que dans l'expérience précédente, sauf pour LogootSplit qui a un comportement différent. En effet, LogootSplit a été conçu pour la bonne gestion des blocs. Avec une proportion inférieure à 60%, il obtient le même comportement que les autres. Il est stable même si la taille du document augmente. Avec plus de 60% de blocs, sa surcharge occupe seulement trois fois la taille du document.

Taille des messages Pour optimiser les ressources du réseau, les algorithmes doivent générer et diffuser des messages de petite taille. La figure 4.10 présente la taille moyenne des messages sérialisés en fonction de la proportion d'opérations bloc. Sans surprise, le comportement et le classement des algorithmes sont les mêmes obtenus sur la taille de la mémoire occupée. Au

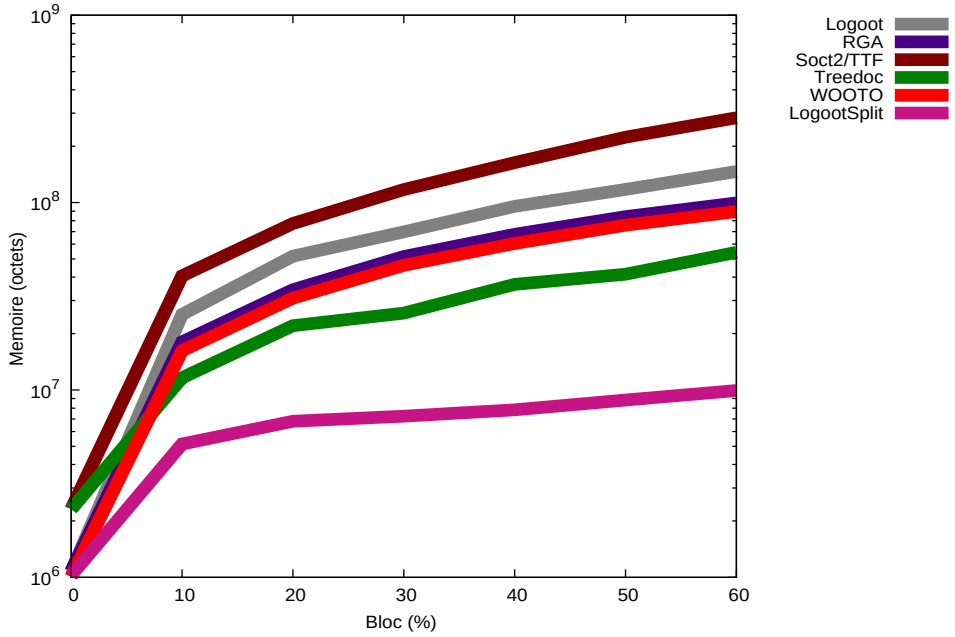


FIGURE 4.10 – Taille des messages

fil du temps, SOCT2/TTF produit un flux important dans le réseau. L'algorithme envoie le vecteur d'horloge dans le message, c'est pourquoi, SOCT2/TTF ne passe pas à l'échelle. Pour les algorithmes CRDT, les messages contiennent l'identifiant de l'élément. Donc la taille des messages est proportionnelle à la taille des identifiants.

4.1.5 Nouvel algorithme : WOOTH

À partir de l'analyse du résultat, nous avons remarqué que l'utilisation de la table de hachage pour la recherche de la position durant l'intégration des opérations améliore les performances de l'algorithme. En effet, RGA obtient les meilleures performances en temps d'intégration. D'un autre coté, l'algorithme WOOTO obtient de bonnes performances en temps d'exécution locale et en mémoire, par contre il est l'algorithme CRDT le moins performant en temps d'intégration. L'idée est d'améliorer l'algorithme WOOTO par l'utilisation des tables de hachage.

ALGORITHME	LOCAL		REMOTE	
	INS	DEL	INS	DEL
WOOTO	$\mathcal{O}(H^3)$	$\mathcal{O}(H)$	$\mathcal{O}(H^3)$	$\mathcal{O}(H^2)$
WOOTH	$\mathcal{O}(H^3)$	$\mathcal{O}(H)$	$\mathcal{O}(H^3)$	$\mathcal{O}(H^2)$
RGA	$\mathcal{O}(H)$	$\mathcal{O}(H)$	$\mathcal{O}(H^2)$	$\mathcal{O}(H^2)$

TABLE 4.7 – Complexité temporelle dans le pire cas

WOOTH [7] est une nouvelle version de WOOTO inspirée par l'approche RGA. Les performances lors de la génération des opérations et la diffusion des mises à jour sont identiques à WOOTO. Cependant, lors de l'intégration des opérations, l'utilisation des tables de hachage et

les listes chaînées peuvent améliorer les performances de WOOTO.

Algorithme	moy. locale		moy. remote	
	ins	del	ins	del
WOOTO	$\mathcal{O}(N + l.d^2)$	$\mathcal{O}(N + l)$	$\mathcal{O}(l.(N + d^2))$	$\mathcal{O}(l.N)$
WOOTH	$\mathcal{O}(N + l.d^2)$	$\mathcal{O}(N + l)$	$\mathcal{O}(l.(N/b + d^2))$	$\mathcal{O}(l.(1 + N/b))$
RGA	$\mathcal{O}(N + l)$	$\mathcal{O}(N + l)$	$\mathcal{O}(l.(N/b + d))$	$\mathcal{O}(l.(1 + N/b))$

Table 4.8 – Complexité temporelle en moyenne

La recherche du caractère précédent et suivant dans le document WOOTO pour intégrer une opération nécessite le parcours de toute la chaîne au pire cas. Le tableau 4.7 et 4.8, présente respectivement la complexité temporelle dans le pire cas et en moyenne, pour les algorithmes WOOTO, RGA et WOOTH lors de la génération et d'intégration des opérations.

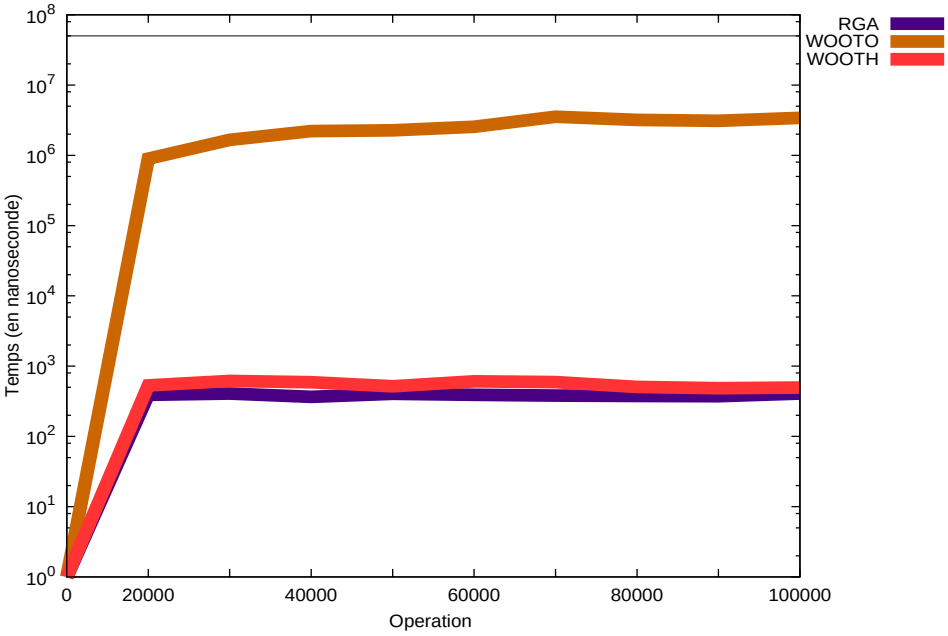


Figure 4.11 – Temps d'intégration en fonction de nombre de modifications

Lors de la génération, la complexité de WOOTH et la même que pour WOOTO. L'algorithme de linéarisation appliqué sur les éléments concurrents et les pierres tombales rendent la complexité moyenne $\mathcal{O}(l.d^2)$, et au pire cas $\mathcal{O}(H^2)$. Cependant, avec l'utilisation des tables de hachage, la recherche des élément lors de l'intégration est optimisée. En effet, la complexité temporelle de WOOTH est en fonction de N/b au lieu de N en moyenne.

Les figures 4.11 et 4.12 présentent respectivement le temps d'intégration en fonction de nombre d'opérations et de répliques pour les trois algorithmes WOOTO, RGA et WOOTH. On remarque clairement que l'utilisation des tables de hachage pour trouver les éléments précédents et suivant améliore les performances de WOOT. Même si RGA et WOOTH sont des algorithmes différents, on peut observer un comportement commun entre les deux approches. En

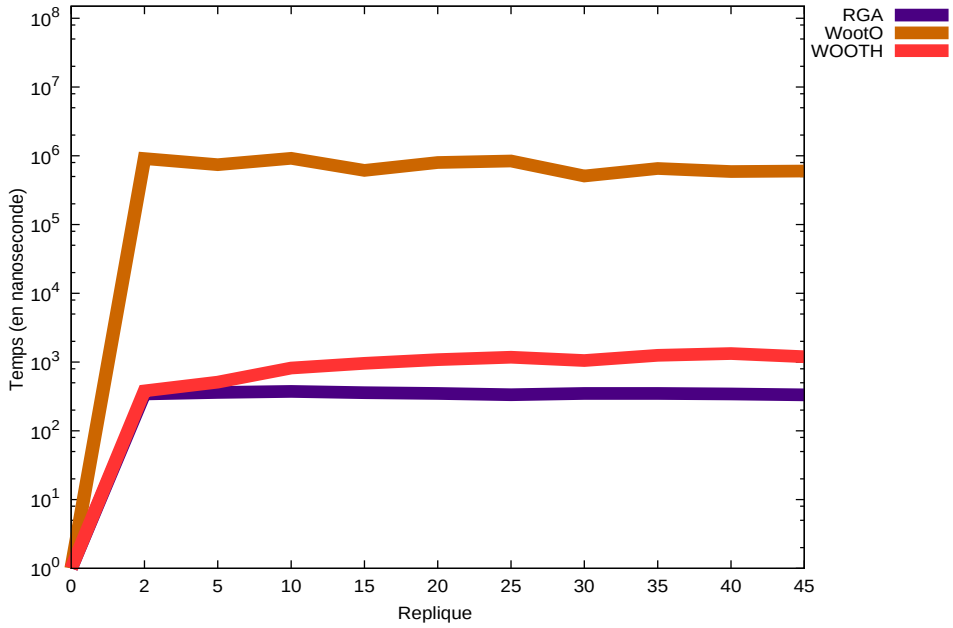


FIGURE 4.12 – Temps d'intégration en fonction de nombre de répliques

effet, les deux algorithmes se basent sur l'utilisation des tables de hachage et les listes chaînées.

L'utilisation des tables de hachage n'occupe pas beaucoup d'espace mémoire. Comme les figures 4.13 et 4.14 démontrent, il n'existe pas une grande différence en mémoire entre WOOTO qui n'utilise pas de table de hachage et WOOTH qui les utilise. De plus, WOOTH occupe légèrement moins de mémoire que RGA. Ce dernier stocke dans son modèle une structure de donnée dite s4vector pour trouver la position des éléments dans la table de hachage et pour résoudre les conflits entre les insertions concurrentes à la même position.

Discussion Les résultats obtenus à partir de la simulation sont compatibles avec l'étude théorique. Cette cohérence valide l'implémentation des algorithmes. De plus, bien que le jeu de données soit généré aléatoirement dans les différentes expériences, les résultats sont cohérents les uns avec les autres. Par exemple, en prenant les valeurs expérimentales de base (10 000 opérations, 10 répliques, 15 % blocs, 80 % des insertions), on obtient le même classement et à peu près les mêmes performances dans chaque série d'expériences.

Comme les résultats le démontrent, le choix d'un algorithme dépend fortement du contexte de l'application et de la collaboration. TreeDoc est un algorithme générique, il obtient de bons résultats pour la quasi-totalité des expériences, toujours en dessous de $50\mu s$ en moyenne. Cependant, nous ne pouvons pas le recommander pour chaque utilisation, par exemple dans une collaboration avec un taux élevé de copier/coller, LogootSplit est le mieux adapté. Pour une collaboration avec un très grand nombre de participants, RGA et WOOTH sont les plus appropriés. En effet, ils obtiennent le meilleur temps d'intégration des opérations en remote.

Pour les périphériques limités en mémoire, l'utilisation de LogootSplit est le bon choix ; car il consomme peu de mémoire, génère moins de flux dans le réseau et obtient de bonne performances

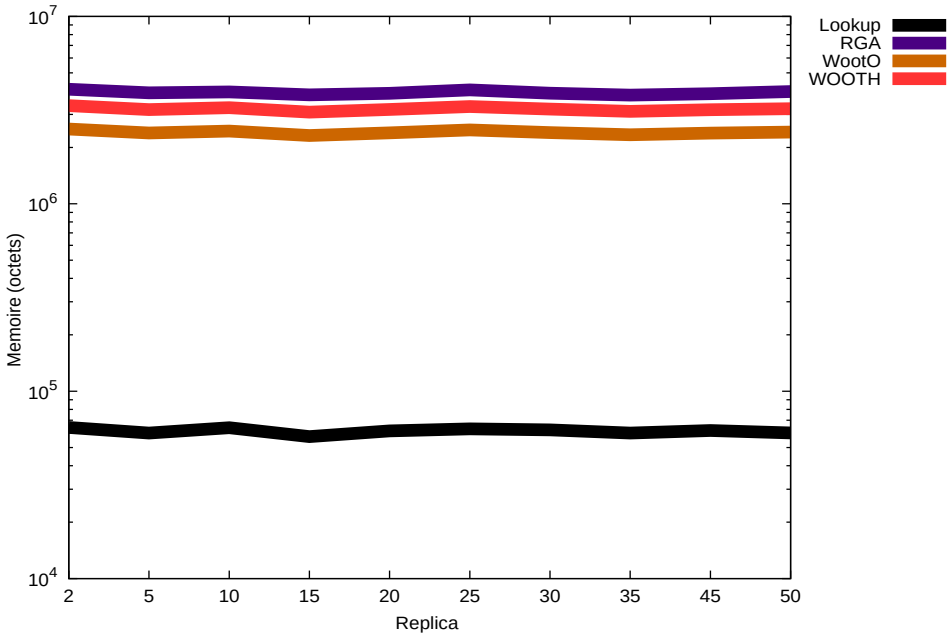


FIGURE 4.13 – Occupation mémoire par nombre de répliques

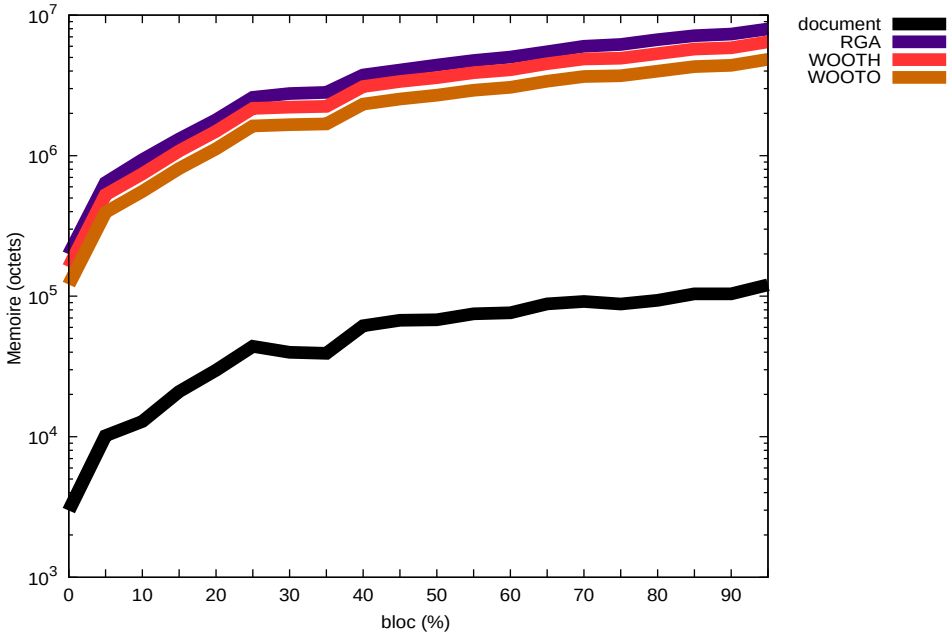


FIGURE 4.14 – Occupation mémoire par % de bloc

pour le temps d'exécution ($< 500\mu s$ en moyenne).

4.1.6 Type de données : Arbre

La convergence inéluctable est difficile à garantir face aux conflits lorsque la structure de données est complexe. En effet, l'utilisation des structures de données récursive pour représenter les graphes, arbres, etc, génère des conflits qui n'apparaissaient pas dans les structures précédentes (ensemble et texte). Pour contrôler ces conflits, le framework propose des *politiques de gestion de conflits* décrite dans la section 2.3.6. Certains systèmes de collaboration, tels que les systèmes de contrôle de versions (Git, SVN,, etc), ou les systèmes de fichiers distribués [44] nécessitent l'intervention des développeurs durant la phase de fusion pour corriger les conflits, alors que d'autres les corrigent automatiquement. Par exemple, SVN crée un "arbre de conflit" lorsque le fichier *these.tex* est créé sous le répertoire */Bureau/Rapport* qui est supprimé en concurrence. Notre approche cherche automatiquement la meilleure fusion possible, en se basant sur les politiques suivantes :

1. Skip : supprimer le noeud orphelin, donc supprimer le fichier *these.tex*,
2. Reappear : recréer le chemin qui mène vers le noeud orphelin. Suivant notre exemple, le chemin à recréer est */Bureau/Rapport/these.tex*,
3. Root : placer le noeud orphelin sous noeud root. Donc placer le fichier *these.tex* sous le répertoire /,
4. Compact : placer le noeud orphelin sous le dernier noeud prédécesseur. Dans notre exemple, le dernier noeud prédécesseur est */Bureau/*. Donc placer le fichier *these.tex* sous le répertoire */Bureau/*.

Dans le contexte d'une édition collaborative d'un document structuré en paragraphes, sections et chapitres, un noeud orphelin peut être une phrase insérée dans un paragraphe supprimé en concurrence, un paragraphe ajouté dans une section supprimée en concurrence, etc. Pour ordonner les éléments dans l'arbre d'un fichier structuré et garantir un ordre total entre les éléments, une *position unique* est attribuée pour chaque élément. Ces positions permettent une insertion entre deux autres. L'ordre total assure que tout les éléments apparaissent dans le même ordre sur chaque réplique. Les algorithmes décrits précédemment tels que Logoot, Treedoc, WOOTs et FCEdit génèrent ces positions uniques pour assurer la cohérence éventuelle.

Pour générer des traces d'exécution, le framework simule une session d'édition collaborative sur un document structuré. Comme décrit dans la section 4.1.1, le générateur de traces intégré dans le framework produit des traces sur un document structuré suivant des paramètres contrôlés par l'utilisateur. Ces paramètres sont similaires à ceux utilisés pour la génération des traces linéaires avec un paramètre de plus dit *perChild*. Ce paramètre est une probabilité utilisée par le générateur de traces pour créer un chemin à une modification. Le générateur de traces traverse l'arbre noeud par noeud. Sur chaque noeud, le générateur de traces génère une variable aléatoire et la compare avec *perChild*. Si la probabilité est satisfaisante et si le noeud courant possède des noeuds fils, le générateur descend vers un des noeud fils et ré-exécute le processus. Sinon, un nouveau noeud fils est créé sous le dernier noeud traversé. Le générateur de traces sauvegarde le chemin parcouru, qui sera attribué à la modification.

Une fois les modifications préparées, le framework fait appel au simulateur pour rejouer les modifications en utilisant les algorithmes de réplication. Le simulateur crée un arbre suivant deux paramètres :

1. typeNode : un noeud peut être de type FCTree ou TreeOPT. De plus, dans le cas d'un arbre ordonné, les algorithmes de réplication tels que Logoot, WOOTH, RGA ou Treedoc peuvent être utilisés pour ordonner les noeuds dans l'arbre.
2. politique : une politique de gestion de conflit, skip, reappear, root ou compact ;

Dans ce qui suit, nous évaluons les performances des algorithmes de réplication dédiés aux documents structurés à savoir FCTree et TreeOPT. Et nous évaluons aussi les algorithmes utilisés pour ordonner les noeuds tels que Logoot, WOOTH sur les différentes politiques de gestion de conflits. Les traces que nous avons récupérées contiennent 30 000 opérations avec 88% d'insertions, générés par quatre répliques. Une opération générée est soit une insertion d'un élément ou une suppression d'un sous-arbre/noeud. Une opération locale est divisée en plusieurs opérations remote qui sont diffusées dans le réseau et envoyées aux autres répliques. Nous évaluons le temps d'exécution (en local et en remote) ainsi que l'occupation mémoire pour chaque algorithme.

Temps d'exécution selon la politique skip

Les algorithmes de réplication pour les documents structurés FCTree et TreeOPT ont été proposés pour traiter les conflits avec la politique skip. Pour cette raison, nous les évaluons seulement avec la politique skip.

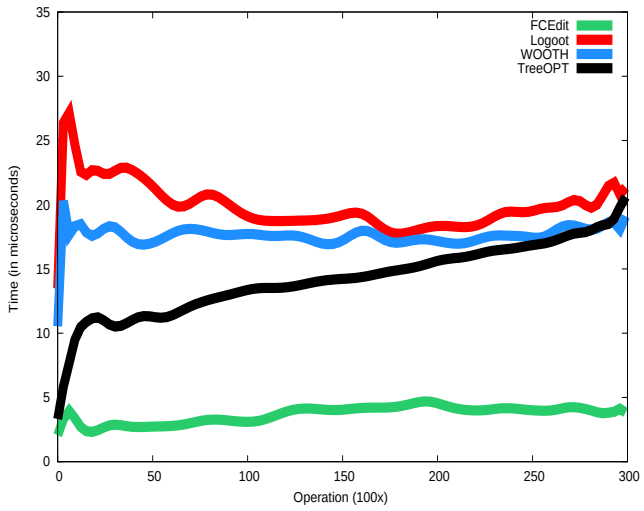


FIGURE 4.15 – Temps d'exécution en local avec Skip.

Local Le temps d'exécution moyen des opérations en local est présenté dans la figure 4.15.

Les performances des algorithmes dédiés pour les documents linéaires (Logoot et WOOTH) sont moins bonnes que celles obtenues avec les algorithmes dédiés pour les documents structurés (FCEdit et TreeOPT), mais ils restent stables tout au long de l'expérience. Ils ne dépassent pas $30\mu s$, ce qui les rend acceptable pour les utilisateurs. Les performances de TreeOPT basées sur SOCT2 se dégradent au début de l'expérience, étant donné que le taux d'insertion est supérieur aux suppressions, la taille de l'arbre devient rapidement grande. Après 100 000 opérations, la majorité des algorithmes deviennent stables. FCEdit est le meilleur algorithme puisque chaque

noeud est identifié par un identifiant unique, en utilisant une table de hachage pour relier identifiant et noeud, avec une complexité d'environ $O(1 + n/k)$ en moyenne, il obtient la meilleure performance.

Remote Dans la figure 4.16, nous présentons le temps d'exécution remote des algorithmes utilisant la politique *skip* sur une échelle logarithmique.

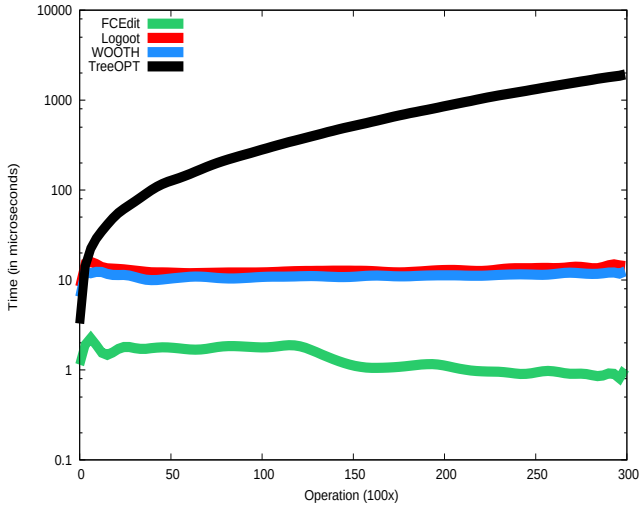


FIGURE 4.16 – Temps d'exécution en remote avec Skip

Pour simuler une expérience réelle, le mécanisme de ramasse de miettes [52, 105] de SOCT2 est désactivé. Puisque les utilisateurs se déconnectent et se connectent à tout moment, le mécanisme de ramasse de miettes ne peut pas purger l'histoire. De ce fait, les performances TreeOPT se dégradent au fil du temps. L'ensemble des opérations reçues sont stockées dans l'historique. Par conséquent il prend du temps pour séparer les opérations concurrentes des opérations transformées, ce qui rend l'algorithme moins efficace.

Comme en local, les performances de Logoot et WOOTH restent stables. Avec $10\mu s$, il surpasse l'algorithme TreeOPT. La performance de FCEdit reste bonne et stable durant toute l'expérience, avec seulement 3 μs, il devient le meilleur algorithme dans notre expérience.

Comparaison entre les différentes politiques

Dans ce qui suit, nous allons présenter les comportements de Logoot avec les différentes politiques qui existent et WOOTH également avec la politique *reappear*. Pour les arbres ordonnés basés sur WOOTH, les politiques *root* et *compact* ne sont pas autorisées. En effet, nous ne pouvons pas fusionner des noeuds qui dépendent de l'élément suivant et précédent avec d'autres noeuds situés sous une origine différente.

Local Dans la figure 4.17 les performances obtenues sont globalement les mêmes sauf pour la politique *root*. Ceci est dû au nombre de fils sous le noeud *root*. Dans les politiques *root* et *compact*, l'algorithme doit déplacer un sous arbre supprimé. Dans la politique *root*, il le déplace sous la racine de l'arbre alors que dans la politique *compact* il le déplace sous le dernier père

observable. Dans le cas où le noeud déplacé se trouve avec un autre noeud avec le même label, les deux noeuds sont fusionnés. Le nombre de noeud sous la racine dans la politique *root* est plus grand que le nombre de fils sous un noeud dans la politique *compact*. En effet, dans le cas de la politique *root* la racine contient tous les noeuds supprimés, alors que dans la politique *compact* un noeud contient ses noeuds fils et des noeuds supprimés de ses fils. Par conséquent, le temps de recherche d'un noeud avec le même label dans la politique *root* est plus grand que dans la politique *compact*.

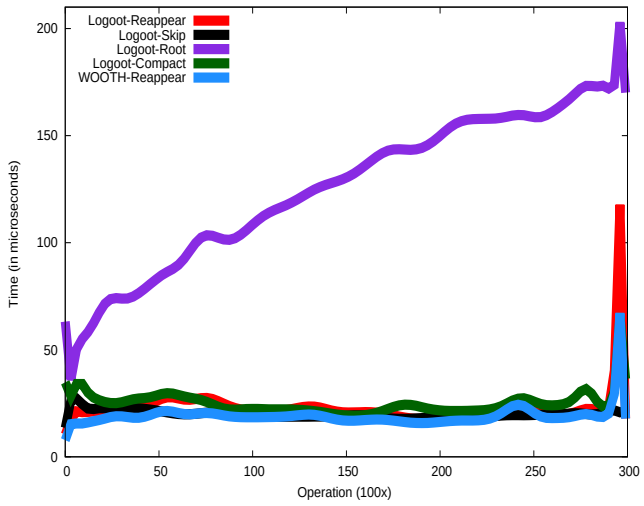


FIGURE 4.17 – Temps d'exécution local avec les différentes politiques.

Remote Le comportement des différents algorithmes durant l'intégration des opérations présenté dans la figure 4.18, est légèrement différent du comportement des algorithmes en temps de génération présenté dans la figure 4.17.

Le comportement de Logoot avec la politique *skip* est le plus stable. Le temps moyen d'exécution reste autour de $10\mu s$. Comme précédemment, le comportement des algorithmes sous la politique *root* est le moins performant et le plus chaotique. Il s'améliore lorsque la réplique supprime un chemin de l'arbre qui se trouve sous le noeud *root* (le cas de 6000 et 23000 opérations). Les deux algorithmes Logoot et WOOTH avec la politique *reappear* et également Logoot avec la politique *compact* restent stables globalement. Bien que certains algorithmes soient moins efficaces que d'autres, le temps d'exécution ne dépasse jamais $1ms$.

Occupation mémoire

La taille de la mémoire occupée de chaque algorithme étudié peut augmenter au fil du temps en raison de l'historique, des pierres tombales ou de la taille des identifiants. Nous présentons dans ce qui suit, le comportement des algorithmes concernant la consommation mémoire pour la politique *skip* sur une échelle logarithmique, présenté dans la figure 4.19.

TreeOpt occupe plus de mémoire que n'importe quel autre algorithme. Cet algorithme utilise une instance de SOCT2 sur chaque noeud de l'arbre. De ce fait, chaque noeud stocke l'ensemble des opérations exécutées et un vecteur d'horloge dans l'historique. L'algorithme WOOTH et

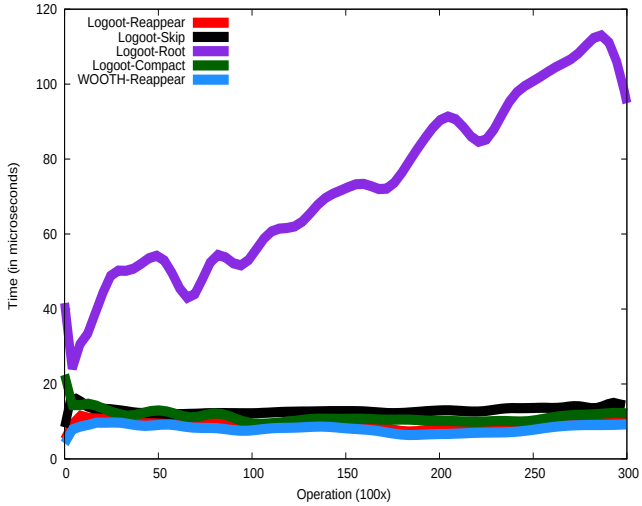


FIGURE 4.18 – Temps d'exécution remote avec les différentes politiques

Logoot ont presque le même comportement. WOOTH stocke une table de hachage et ne supprime jamais les identifiants, mais seulement les masquer à l'utilisateur, alors que la taille mémoire occupée par Logoot dépend de la taille des identifiants. Même si Logoot supprime réellement les éléments supprimés, le gain de mémoire est caché par la taille de ses identifiants. FCEdit reste le meilleur algorithme en ce qui concerne la mémoire occupée. Les identifiants FCEdit sont moins coûteux que les identifiants Logoot et les noeuds supprimés sont réellement supprimés de la mémoire contrairement à WOOTH et TreeOpt.

4.1.7 Conclusion

Nous avons présenté dans cette section les performances des algorithmes de réplication de différents types de données sur des traces synchrones. Nous avons mené plusieurs expériences et dans chacune nous avons évalué un algorithme sur un critère, dans le but de savoir quel est le facteur qui influence le plus les performances des algorithmes.

Pour les types de données "ensemble", nous avons trouvé que l'algorithme CounterSet et LWWSet sont les plus performants en terme de temps d'exécution et de mémoire occupée, car ils ne requièrent aucune identification contrairement à ORset et OptORSet.

Pour les algorithmes de réplication textuels, nous avons trouvé que Treedoc est un algorithme générique souhaitable pour n'importe quelle application, car dans la quasi-totalité des expériences, il ne dépasse pas les $50\mu s$ en moyenne. Cependant, dans une collaboration avec un taux élevé de copier/coller, LogootSplit est le mieux adapté. RGA et WOOTH sont plus performants en temps d'intégration. Dans une session d'édition collaborative massive où seulement quelques participants éditent, ces deux algorithmes sont les plus appropriés. Pour les périphériques limités en mémoire, l'utilisation de LogootSplit est la plus appropriée, car il consomme peu de mémoire et génère moins de flux dans le réseau.

Les algorithmes destinés à l'édition collaborative des documents structurés dépendent de la politique utilisée pour la gestion des conflits. Cependant, les algorithmes OT restent les moins performants car ils stockent toutes les opérations dans leurs modèles et nécessitent des transfor-

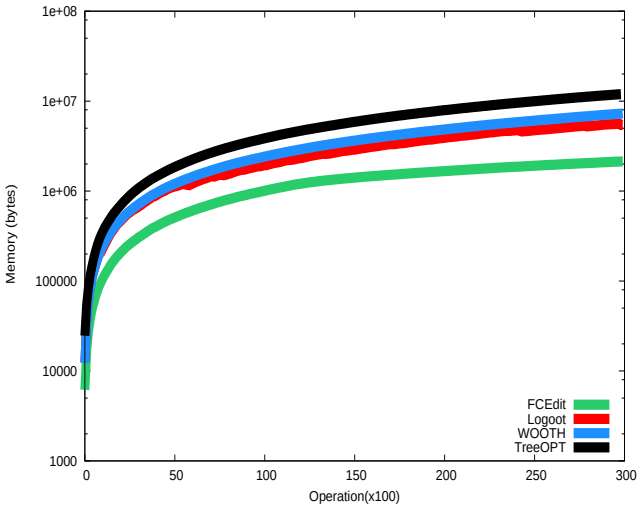


FIGURE 4.19 – L'occupation mémoire.

mations. FCEdit obtient les meilleurs performances dans notre expérience.

Pour valider notre expérience et comprendre mieux le comportement des algorithmes, nous allons exécuter dans la section suivante, les mêmes algorithmes sur des traces asynchrones.

4.2 Performance en mode asynchrone

Dans une édition collaborative asynchrone, les modifications sont regroupées dans des états. Les utilisateurs contrôlent le moment de rendre visible leurs modifications ou le moment d'intégrer les modifications des autres développeurs, introduisant ainsi plus de concurrence.

Comme décrit dans la section 3.2.3, notre framework intègre un mécanisme qui est capable d'extraire des traces d'édition collaborative à partir des dépôts Git, et d'exécuter sur ces traces les différents algorithmes optimistes. Dans cette section, nous évaluons les algorithmes d'édition collaborative pour les documents linéaires dans le cadre d'une édition collaborative asynchrone. Pour comparer les deux expérimentations (synchrone et asynchrone), les expériences sont exécutées sur des machines virtuelles Amazon EC2, avec les mêmes configurations. Nous validons les résultats obtenus par une étude statistique. Nous étudions la corrélation entre les performances des algorithmes et les caractéristiques obtenus à partir de l'édition d'un fichier.

4.2.1 Corpus

Pour produire des corpus asynchrones, nous avons récupéré [19] les projets les plus populaires disponibles sur GitHub [20]. Dans le tableau 5.1, nous présentons les caractéristiques de ces projets. Pour chaque fichier, nous avons extrait le nombre de commits, de lignes modifiées, la proportion d'insertions, la taille moyenne des blocs et le nombre de merge. Pour garantir la reproductibilité de l'expérience, nous présentons aussi le numéro du dernier commit (sha1) effectué sur le répertoire

19. par la commande : git clone.
20. `https://github.com/popular/starred`

au moment de notre expérience. Ainsi, le résultat peut être reproduit en récupérant le dépôts Git à partir de cet identifiant [21].

Projet	git/git			twitter/bootstrap			joyent/node		
Head sha1	8c7a786b6c8			cd42f56178			9f29785783		
Fichiers	2496			250			2588		
Caractéristiques	max	min	moy	max	min	moy	max	min	moy
Commits	1742	2	75.75	633	12	84.21	631	2	21.75
Merge	451	0	12.92	55	0	7.44	37	0	0.88
Ligne modif	25304	149	1953.49	119594	964	18104.65	123340	276	7400.23
Moy taille blocs	28.13	1.84	6.78	80.93	12.35	32.06	761.25	7.65	97.45

Table 4.9 – Caractéristiques des projets Git

Performance générale Le tableau 4.10 présente le temps d'exécution des différents algorithmes sur trois répertoires (git/git, joyent/node and twitter/bootstrap). La valeur *commit* est le temps nécessaire pour exécuter toutes les opérations locales sur les fichiers, divisé par le nombre de commits. La valeur *merge* est le temps nécessaire pour intégrer toutes les opérations distantes dans les fichiers, divisé par le nombre de merge. Ainsi, ces mesures correspondent au temps observé par l'utilisateur s'il utilise un DVCS basé sur un de ces algorithmes.

Algorithme	git		node		bootstrap	
	Commit	Merge	Commit	Merge	Commit	Merge
SOCT2/TTF	337	6723	1816	18153	1373.0	33485.8
WOOTH	306	15	2232	133	1936.9	612.4
RGA	365	13	1662.2	49.3	1709.1	64.1
Logoot	159	24	1138.2	64.7	1337.0	76.0
LogootSplit	99	63	135.2	90.1	188.5	97.7
TreeDoc	96	18	200.4	41.9	595.0	76.1

Table 4.10 – Temps moyen pour commit et merge (in μs)

Le temps d'exécution locale des algorithmes est presque le même que celui obtenu dans le mode synchrone. En effet, on peut classer les algorithmes en trois catégories : les algorithmes basés sur les blocs –LogootSplit et Treedoc– qui obtiennent les meilleurs performances, les algorithmes basés sur les pierres tombales qui obtiennent les pires performances - WOOTH, RGA et SOCT2/TTF - et Logoot entre les deux. Cependant, le classement au sein de ces ensembles peut varier en raison des caractéristiques de la collaboration et la structure des données utilisées. Par exemple, LogootSplit est légèrement plus performant que TreeDoc car il n'utilise pas les pierres tombales, et RGA plus stable que WOOTH car il n'utilise aucun système de linéarisation.

Le temps d'intégration des opérations en mode asynchrone est plus surprenant par rapport aux expériences synchrones. L'algorithme SOCT2/TTF demeure le pire. Les performances de RGA, TreeDoc, Logoot et LogootSplit restent très bonnes. Contrairement à l'expérience syn-

21. revenir vers une version ultérieur avec la commande : git reset

chrone, la performance de Logoot est légèrement meilleure que LogootSplit. De plus, le WOOTH obtient les pires performances sur les projets *bootstrap* et *node*.

Ce comportement est dû à quelques fichiers les plus édités dans ces projets, qui causent des problèmes de linéarisation à WOOT. En effet, le tableau 4.9 montre que ces deux projets contiennent des fichiers qui sont beaucoup plus édités que celui de git.

Discussion Pour détecter quelle caractéristique affecte le plus les performances des algorithmes, nous avons effectué une étude sur la corrélation entre les performances et les caractéristiques. Nous considérons que les fichiers peu édités sont peu informatifs. L'expérience est effectuée seulement sur les fichiers les plus éditées.

Nous avons analysé la *corrélation bivariée* entre le temps d'exécution moyen par ligne éditée et la taille moyenne des blocs, la proportion d'insertions, le nombre de lignes et le nombre de répliques simulées. Nous avons remarqué que les temps d'exécution sont fortement corrélés avec la taille moyenne des blocs. Donc, nous avons calculé la *corrélation partielle* entre le temps d'exécution et les autres caractéristiques en contrôlant la taille des blocs.

4.2.2 Intégration

Le tableau 4.11 présente la corrélation entre le temps d'intégration et les caractéristiques d'éditions. La figure 4.20 et 4.21 présentent respectivement, un nuage de points pour le temps d'intégration en fonction de nombre de lignes éditées et de répliques simulées.

Algorithm	Lines		Replicas	
	Bivariate	Partial	Bivariate	Partial
SOCT2/TTF	0.99	0.99	0.45	0.3
WOOTH	0.07	0	0.07	0.03
RGA	-0.06	-0.12	0.02	0.03
Logoot	0.18	-0.01	0.07	-0.06
LogootSplit	0.34	-0.25	0.075	-0.02
TreeDoc	-0.01	-0.08	0.02	0.02

Table 4.11 – Corrélation pour le temps d'exécution en remote.

Le tableau et les résultats obtenus par le framework démontrent que :

— les performances des CRDT ne sont pas influencées par le nombre de répliques et de lignes éditées, car la corrélation partielle est aux alentours de 0. Ce résultat est compatible avec le résultat obtenu sur le corpus synchrone présenté dans les figures 4.5 et 4.6 ;

— Contrairement aux algorithmes CRDT, le temps d'intégration des opérations pour l'algorithme SOCT2 est influencé par le nombre de lignes éditées et le nombre de répliques, car la corrélation partielle est 0.99 en fonction des lignes et de 0.3 en fonction des répliques. Ces deux paramètres ont un impact sur le taux de concurrence et la taille de l'historique. Cependant, SOCT2 perd en performance lors de l'intégration car il nécessite le parcours de l'historique pour détecter les opérations concurrentes et effectuer les transformations nécessaires. Ce résultat est compatible avec l'expérience d'éditions synchrones présentée dans les figures 4.5 et 4.6 ;

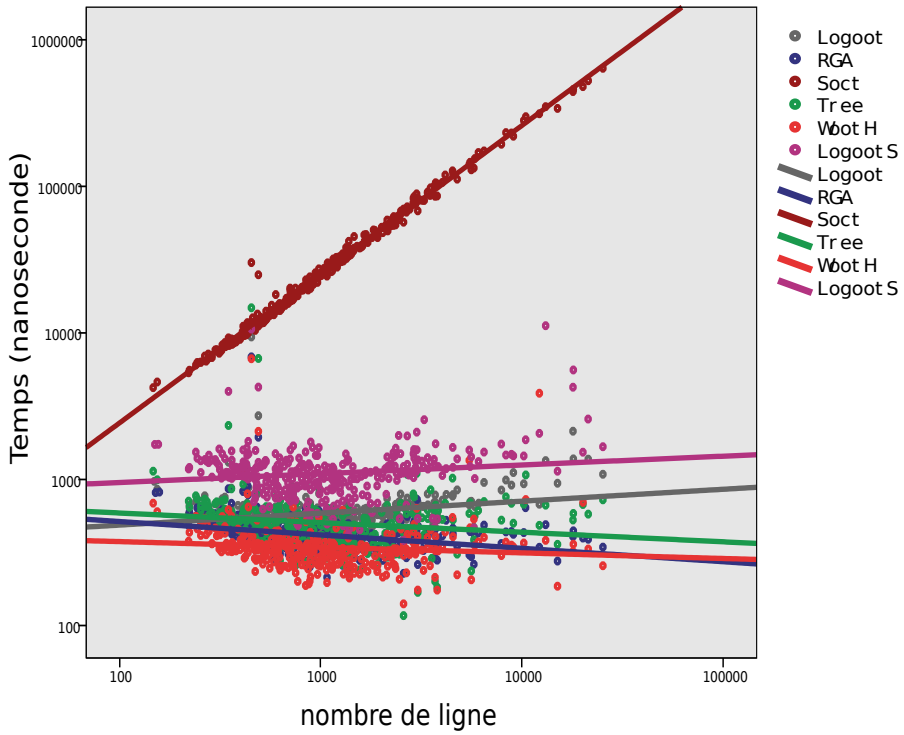


FIGURE 4.20 – Temps d'intégration par nombre de lignes

— LogootSplit s'améliore en temps d'intégration lorsque le nombre de modifications augmente. En effet, La corrélation partielle est -0.25 entre le temps d'intégration et le nombre de lignes éditées. Contrairement à l'expérience sur le corpus synchrone, dans cette expérience les modifications concernent des lignes et LogootSplit est dédié pour ce type de collaboration. Cependant, l'amélioration est due à la position des modifications. Au début de l'édition, un document est peu édité. Les développeurs insèrent au milieu du document (pour le corriger et le mettre en forme) ce qui nécessite l'intégration des opérations de type *split* qui casse la ligne en deux blocs et les identifiants correspondants. Mais avec le temps, le document devient plus long, les développeurs éditent à la fin du document et intègre moins d'opérations de split.

4.2.3 Exécution locale

L'analyse visuelle de la dispersion des données est moins évidente que pour l'intégration. Donc, nous avons mené une analyse statistique des données pour vérifier la cohérence avec les autres résultats expérimentaux. L'analyse du temps d'exécution locale est présenté dans le tableau 4.12.

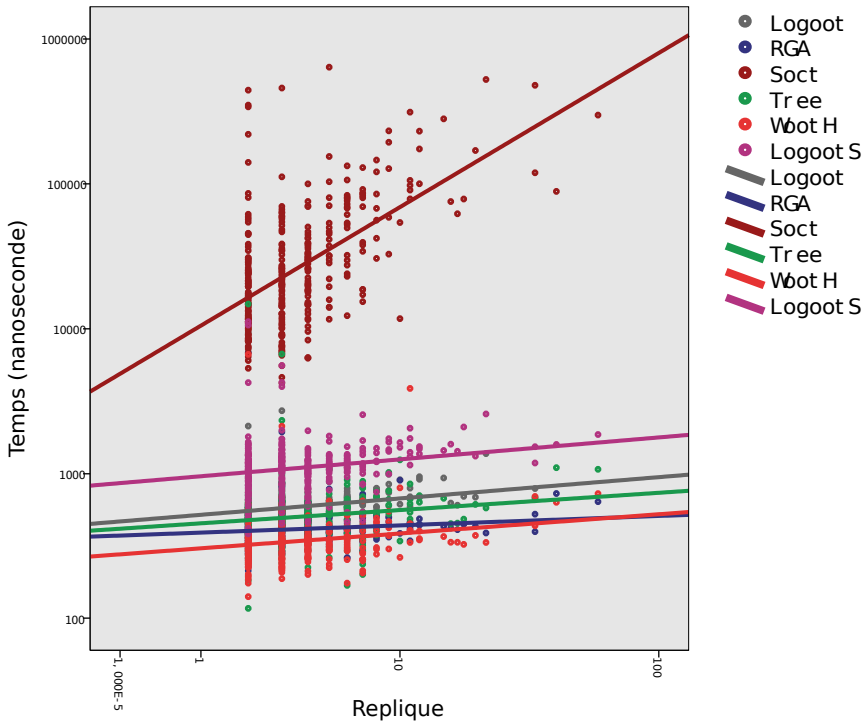


FIGURE 4.21 – Temps d'intégration par nombre de répliques

ALGORITHME	BLOCS	INSERTION		RÉPLIQUES	
		BIVARIATE	PARTIAL	BIVARIATE	PARTIAL
SOCT2/TTF	0.69	0.16	0.32	0.11	-0.02
WOOTH	0.83	0.02	0.29	0.34	-0.01
RGA	0.77	0.09	0.32	0.21	-0.11
Logoot	0.51	0.24	0.33	-0.01	-0.01
LogootSplit	0.87	-0.09	-0.1	0.15	-0.09
TreeDoc	0.68	-0.15	-0.15	0.13	0.03

TABLE 4.12 – Corrélation pour le temps d'exécution local

Les valeurs de corrélation indiquent que :

— les performances locales et le classement entre les algorithmes sont conformes aux expériences d'édition synchrones,

— la corrélation partielle montre que le pourcentage d'insertion influence les algorithmes qui utilisent les pierres tombales et l'algorithme Logoot. Pour SOCT2/TTF, WOOTH et RGA cette influence est due à la taille de l'historique qui devient assez grand. Ce qui ralenti la génération de l'opération puisque les algorithmes doivent trouver la position correspondante à la ligne insérée. Les performances de Logoot se dégradent à cause de la taille des identifiants. Ce résultat est similaire à celui obtenu dans l'expérience sur le corpus synchrone présenté dans la figure 4.7a ;

— en local, le nombre de réplique n'influence aucun algorithme de réplication. La corrélation

partielle dans le tableau 4.12 est proche de 0 pour tous les algorithmes. Le comportement des algorithmes en local est le même comportement obtenu sur les traces synchrones et présenté dans la figure 4.7b ;

— le temps d'exécution local pour LogootSplit et TreeDoc et la proportion d'insertions sont légèrement corrélés avec un coefficient négatif. Comme la complexité théorique l'indique, Treedoc et LogootSplit génère une seule opération lors d'insertion et plusieurs opérations lors de la suppression. Plus le taux d'insertion augmente et plus le nombre de suppression diminue et donc moins d'opérations sont générées, ce qui explique le gain de temps en local. Le résultat obtenu suit le comportement des algorithmes en local présenté dans la figure 4.7a.

4.2.4 L'occupation mémoire

le tableau 4.13 présente l'occupation mémoire total de chaque algorithme ainsi que la taille moyenne des messages. Nous présentons également l'espace mémoire occupé par le système Git dans son répertoire .git.

Les résultats obtenus sont compatibles avec l'expérience synchrone. Le classement entre les algorithmes est respecté, sauf que Logoot fonctionne relativement mieux en mode asynchrone. Les algorithmes basés sur les pierres tombales – SOCT2/TTF, RGA et WOOTH dans cet ordre – obtiennent les plus mauvaises performances car ils ne suppriment jamais les éléments. TreeDoc et LogootSplit obtiennent les meilleures performances.

Malgré les mauvaises performances de sérialisation Java et la compression utilisée par l'outil Git, l'espace mémoire occupée par les algorithmes CRDT est inférieur à celui occupé par le système Git [22]. Cependant, Git permet de revenir sur un état ultérieur. Cette fonctionnalité est réalisable par les algorithmes OT et CRDT en utilisant les pierres tombales, ainsi, sans coût supplémentaire de mémoire pour TTF, RGA et WOOTH.

Algorithme	git/git	twitter/bootstrap	joyent/node	messages
.git	60	26	171	-
SOCT2/TTF	125.36	39.50	223.51	147.40
WootH	55.84	18.38	88.14	107.85
RGA	63.94	21.41	101.24	116.87
Logoot	45.21	8.10	67.30	113.58
LogootSplit	29.22	6.32	44.98	66.105
Treedoc	38.69	7.61	22.85	73.40

Table 4.13 – mémoire totale occupée (par mo)

4.2.5 Taille des messages

La taille des messages des algorithmes sur les traces asynchrones est présenté dans le tableau 4.13. Le résultat obtenu suit le même classement par rapport à celui obtenu avec l'expérience synchrone. En effet, le pire algorithme est SOCT2. Il produit des messages de grande taille, car il envoie le vecteur d'horloge avec l'opération. Par conséquent il ne passe pas à l'échelle. RGA et

22. Git stocke des informations supplémentaires concernant la gestion des projets. Ces informations sont plus petite que l'état des fichiers

WOOTH spécifient l'élément précédent et suivant dans leurs messages, tandis que Logoot ajoute son identifiant qui grossit au fil du temps. Treedoc et LogootSplit qui sont basés sur des blocs sont les meilleurs, car leurs identifiants ne sont pas volumineux.

4.3 Conclusion

Dans ce chapitre, nous avons évalué les différents algorithmes de réplication optimistes sur des traces synchrones et asynchrones. Dans la première section, nous avons donné un aperçu sur l'expérience que nous avons menée pour récupérer des traces synchrones réalistes. Nous avons ensuite étudié la complexité théorique des différentes approches et nous les avons exécutées sur des traces synchrones simulées inspirées des traces réalistes.

Dans la deuxième section, nous avons mené une autre expérience pour valider le résultat de la première l'expérience. Nous avons donc exécuté les approches dédiées au document textuels linéaires sur des traces asynchrones récupérées à partir des répertoires Git. Nous avons trouvé que la hiérarchie entre les algorithmes en temps d'exécution, mémoire occupée et la taille des messages est compatible avec celle obtenue sur les traces synchrones.

Après avoir analysé le résultat, nous avons proposé un nouvel algorithme WOOTH qui prend les avantages de deux autres algorithmes RGA et WOOTO. Nous avons montré que WOOTH obtient de bonnes performances en temps de génération, d'intégration et en mémoire occupée. Cet algorithme est souhaitable pour les applications d'édition collaborative temps réel.

Les résultats obtenus sont significatifs et cohérents avec l'analyse statistique dans les deux expériences : synchrone et asynchrone. De plus, ils suivent l'analyse théorique. Cette cohérence valide notre méthodologie décrite dans le chapitre 3.

Chapitre 5

Évaluation la Qualité des Résultats

Sommaire

La collaboration pour le développement logiciel nécessite une gestion efficace des contributions des différents développeurs. Dans ce but, les systèmes de gestion de versions tels que Git, SVN, CVS, Darcs et Mercurial ont été développés. Ces systèmes offrent à chaque développeur une copie du projet sur laquelle ils effectuent des modifications. Lorsque le développeur le souhaite, il partage et/ou fusionne ses modifications avec celles des autres. Les systèmes de gestion de versions gèrent et contrôlent les modifications apportées sur le projet. Face aux modifications parallèles, ces systèmes peuvent produire des résultats conflictuels ou erronés durant le processus de la fusion. En effet, dans certains cas, les systèmes ne peuvent pas fusionner correctement les modifications concurrentes et génèrent des conflits. Dans ce cas, le système retourne le résultat erroné aux développeurs pour résoudre les conflits.

Pour réduire le nombre de conflits et créer un environnement favorable à la collaboration, nous étudions dans ce chapitre, le comportement des utilisateurs et la gestion de la concurrence du système *Git* [110]. Ce système permet à chaque développeur de travailler sur une réplique locale du projet. Les développeurs décident quand est ce qu'ils rendent leurs modifications visibles aux

autres développeurs et quand est ce qu'ils fusionnent leurs modifications avec les modifications distantes.

Comme nous l'avons décrit dans la section 3.2.3, Git gère les modifications comme un ensemble d'états. Lorsqu'une réplique reçoit un état distant, le système Git fait appel à l'algorithme git-Merge [40] pour calculer la différence ou le *diff3* [77] entre l'état local et l'état distant avant de les fusionner. À cet étape, le système Git peut détecter des conflits si les développeurs ont modifié en concurrence le même fichier dans la même zone du texte. Le système ne peut pas choisir une version aléatoire et annuler l'autre. Il retourne un conflit à l'utilisateur et lui demande de le résoudre manuellement. Une fois que les conflits sont corrigés, l'utilisateur fusionne les modifications.

Afin de réduire l'effort des développeurs et augmenter la productivité de l'équipe, un système d'édition collaborative doit réduire le nombre de conflits. Aujourd'hui, de nombreuses solutions ont été proposées pour améliorer la fusion automatique. Une distinction peut être faite entre les fusions [74] :

1. textuels [84] : le système traite les modifications ligne par ligne. Si les modifications concurrentes ont été effectuées dans la même zone de texte, une seule modification est sélectionnée. Il est impossible de combiner les deux modifications,
2. syntaxiques [16] : ce type de fusion est plus performant que la fusion textuelle. En effet, certains types de collaboration sont fusionnés sans conflit. Par exemple, si un développeur ajoute des commentaires au milieu d'un code qui est modifié en concurrence par un autre développeur, la fusion textuelle génère un conflit alors que la fusion syntaxique la fusionne correctement,
3. sémantiques [49] : le système fusionne sans conflit les modifications sémantiquement dépendantes.

Dans cette thèse, nous traitons les conflits les plus fréquents dans un système de gestion de version : les conflits textuels [91].

Les approches operation-based que nous avons étudiées dans la section 2.3, ont été proposées pour le contrôle des modifications concurrentes dans l'édition collaborative [34, 113, 7]. Contrairement à git-Merge, ces approches représentent les modifications par une séquence d'opérations qui sont intégrées automatiquement dans le document. Étudier dans quelle mesure leurs résultats satisfait les utilisateurs est important.

Nous proposons dans ce chapitre une méthode pour calculer le nombre de conflits en analysant le résultat de la fusion produit par les différents algorithmes. Nous observons à travers notre framework les différents comportements de collaboration dans l'historique de Git. Ensuite, nous analysons les cas les plus fréquents qui créent les conflits. Enfin, nous adaptons des solutions pour réduire le nombre de conflits, l'effort des utilisateurs, et le temps consommé pour la correction, et donc améliorer la qualité du résultat.

5.1 Description du problème

Comme nous avons décrit dans la section 3.2.3, nous distinguons une opération de *merge* d'une opération de $fusion$. Nous considérons un *merge* une opération de $fusion$ suivie de résolution de conflits.

Le résultat de la fusion dépend fortement de l'approche utilisée (state-based ou operation-based) et du type d'algorithme de réplication utilisé. En effet, la manière de traiter les modifications dans l'approche state-based est différente de l'approche operation-based, et chaque algorithme de réplication operation-based gère les modifications différemment.

Toutes les approches que nous allons évaluer considèrent le texte comme une séquence de lignes.

5.1.1 State-based

L'algorithme git-Merge utilisé par défaut dans le système Git permet de fusionner deux branches de l'historique. Après une fusion, si un ou plusieurs fichiers sont en conflit, le système fait appel à un outil tel que diff3[41] pour gérer les différences sur chaque fichier.

Lors d'une fusion, Git parcourt les deux documents. Si les deux parties ont apporté des changements à la même zone, Git ne peut pas choisir au hasard une version et ignorer l'autre. Il demande aux utilisateurs de résoudre le conflit en affichant les deux versions.

Dans la figure 5.1, supposons que le document original est O. Les utilisateurs 1 et 2 modifient O et produisent respectivement les documents A et B. Lorsque Git fusionne les documents, git-Merge cherche les parties identiques entre O et A et entre O et B. Ensuite, git-Merge examine les parties où O diffère de A et de B à la fois. Enfin, il détecte s'il y a un conflit et dans quelle partie du document.

Si un conflit est détecté, le système renvoie les résultats à l'utilisateur avec des marqueurs, comme nous l'avons décrit dans la section 2.2.1. Ces marqueurs sont utiles pour localiser rapidement le conflit dans le document. Ils précisent la position du conflit dans le document ainsi que les modifications concurrentes effectuées par les autres utilisateurs sur le document original. L'utilisateur concerné est invité à apporter des corrections sur leur document pour résoudre le conflit et ajouter des modifications si nécessaire.

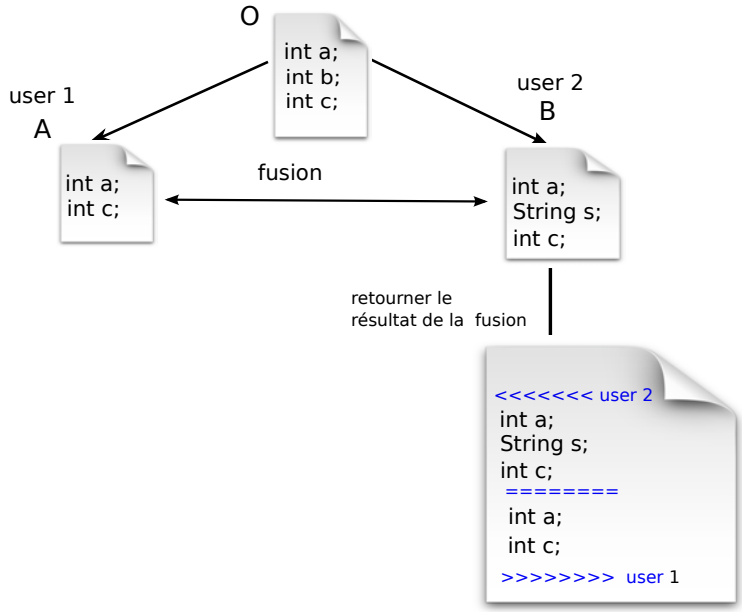


FIGURE 5.1 – Conflit sur les systèmes d'édition state-based

Après la fusion et la correction des conflits, les modifications sont mergées avec succès. Cependant, une fusion sans conflit ne signifie pas que le document ne contient pas d'erreurs. On distingue deux types de conflit :

1. direct : ce type de conflit est généré lorsque des utilisateurs modifient le même document en même temps dans la même zone comme présenté dans la figure 5.1,

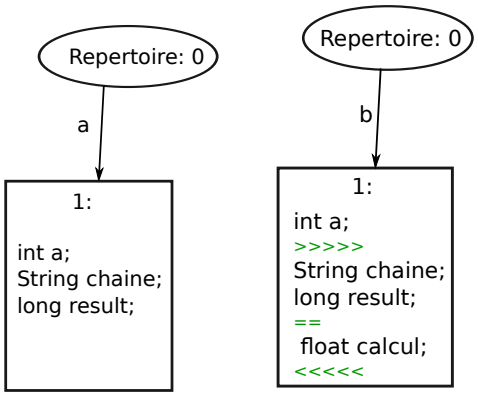
2. indirect : dans certains cas, Git fusionne les modifications avec succès. Par contre le résultat présente des erreurs de compilation ou de test. Un conflit indirect est généré lorsque des modifications concurrentes sont dépendantes. Ces modifications n'affecte pas forcément le même document. Par exemple, un utilisateur fait appel à une méthode Java dans la classe classeA.java, alors qu'un autre utilisateur a modifié en concurrence le nom de cette méthode qui se trouve dans la classe classeB.java. Git fusionne correctement les modifications puisque les deux utilisateurs n'ont pas modifié le document dans la même partie par contre le programme ne compile pas. Le merge textuel ne détecte pas ce type de conflit. Les développeurs doivent donc chercher eux même l'erreur et la corriger.

5.1.2 Operation-based

Pour les approches operation-based, nous allons évaluer un algorithme OT (TTF) et quelques CRDTs. Les algorithmes sont décrits dans la section 2.3.5.

L'application de ces algorithmes sur un système collaboratif réel ne doit pas fusionner automatiquement les opérations concurrentes sans prévenir l'utilisateur. Il est plus approprié d'informer les utilisateurs et de leur demander de vérifier les résultats.

Par exemple, So6 [75] qui est similaire à *diff3* utilise un algorithme OT. La figure 5.2 présente un scénario d'édition collaborative avec So6. Supposons que deux utilisateurs travaillent en concurrence sous le même état initial. L'algorithme So6 génère l'état initial suivant deux opérations : 1) création du fichier par l'opération $mf(1,0,a)$ où 1 est l'identifiant du fichier, 0 est l'identifiant du répertoire et a est le nom du fichier dans le répertoire ; 2) ajouter du texte par l'opération $ab(1,0,$"int a ;"," String chaine ;","long result ;") où 1 est l'identifiant du fichier à modifier et 0 est la position du texte à ajouter.



(a) État initial (b) État après fusion

FIGURE 5.2 – État initial et final du scénario

L'utilisateur 1 renomme le fichier a par b et supprime les deux dernière lignes. En concurrence, l'utilisateur 2 ajoute "float calcul ;" dans le fichier a à la position 3. Lorsque l'algorithme So6 synchronise les opérations, il procède à la transformation des opérations avant de les intégrer dans la copie locale. À ce moment, So6 détecte que les modifications affectent le document dans la même zone du document et que le fichier a été renommé. So6 ne peut pas fusionner les modifications correctement et génère donc un conflit. Des marqueurs sont ajoutés dans le document autour du conflit pour que les utilisateurs le détecte facilement comme présenté dans la figure 5.2b.

5.2 Corpus

Pour calculer l'effort fait par les utilisateurs en cas de conflit, et observer le comportement des utilisateurs, nous avons besoin de savoir ce que les utilisateurs veulent comme résultat final avant de commencer la collaboration. L'historique de Git contient ces informations.

L'idée consiste à reproduire la même collaboration que sur l'historique de Git à l'aide d'un mécanisme intégré dans notre framework. Durant l'exécution, nous observons le comportement des utilisateurs et nous calculons l'effort effectué en cas de conflit pour corriger le document.

Comme l'historique de Git est généré par des outils basés sur des états, nous les transformons en opérations afin de simuler l'exécution sur les algorithmes operation-based. Pour évaluer la qualité des algorithmes de merge, on calcule la différence entre le résultat calculé par ces algorithmes et la version fusionnée disponibles dans l'historique de Git.

Comme décrit dans la section 3.3.2, notre framework rejoue l'histoire des dépôts Git sur des algorithmes operation-based. Il compare le résultat de la fusion obtenu par les algorithmes, avec le résultat du merge produit par les développeurs et qui se trouve dans l'historique. La différence entre ces deux résultats correspond à l'effort que le même développeur aurait fait si Git avait utilisé un de ses algorithmes.

Projet	git			bootstrap			node		
Head sha1	8c7a786b6c			37d0a30589			88333f7ace		
Langage de programmation	C			CSS			JavaScript		
Fichier avec merge	557			69			44		
Nb. Lignes éditées	1 091 125			225 596			1 192 244		
Caractéristiques	max	min	avg	max	min	avg	max	min	avg
Commits	1742	4	59,4	526	5	88,4	631	9	111,1
Merge	451	1	10,1	49	1	6,4	37	1	6,3
Nb. Blocs.	3887	4	190,3	1975	1	117,3	2390	21	496,9
Nb. Lignes.	25305	24	1222,4	25216	66	3317,6	56076	1392	13782,1
Réplique	59	2	3,5	10	2	3,7	4	2	2,1
Merge blocs avec git-Merge	3184			1 614			1 138		
Merge lignes avec git-Merge	10 159			14 658			8 159		

Projet	html5-boilerplate			d3			gitorious		
Langage de programmation	CSS			JavaScript			Ruby		
Head sha1	f27c2b7372			d1d71e16			c1105ebe86		
Fichier avec merge	4			38			72		
Nb. Lignes éditées	4 463			99 093			66 517		
Caractéristiques	max	min	avg	max	min	avg	max	min	avg
Commits	208	27	140,7	891	5	63,27	437	4	58,2
Merge	20	1	11,3	184	1	8,75	10	1	2,1
Nb. Bloc.	230	25	115,7	1348	1	70,13	771	2	58,7
Nb. Lignes.	2 753	142	1 488,3	64 691	30	2 668,45	12 902	54	936,9
Replique	6	2	4,0	30	3	4,16	5	3	3,1
Merge blocs avec git-Merge	24			648			489		
Merge lignes avec git-Merge	87			4 658			2 303		

Table 5.1 – caractéristiques des projets

5.2.1 Étude empirique

Pour détecter les conflits les plus fréquents et comprendre le comportement des développeurs, nous avons besoin de récupérer de large traces. Nous avons donc analysé six projets Git (voir le tableau 5.1) à partir de GitHub et Gitorious, sur la base des critères suivants : *a*) la popularité

des projets de GitHub [23], *b*) activité des projets de Gitorious, *c*) les projets sont développés en utilisant seulement Git dans Github, et pas un miroir de système tels que SVN, puisque le framework analyse seulement les traces git.

Nous avons également sélectionné le dépôt git du logiciel Git lui-même, puisqu'il contient un grand nombre de commits et de merge. Un tel projet contient un bon résultat de merge, puisqu'ils sont faits par des spécialistes de l'outil.

Dans le tableau 5.1, nous présentons les caractéristiques de six projets sélectionnés. Le numéro de commit utilisé pour exécuter nos expériences est présenté avant chaque répertoire. Nous donnons aussi le langage de programmation et le nombre de lignes éditées pour chaque répertoire. Les caractéristiques sont calculées par fichier. Nous avons calculé le maximum, la moyenne et le minimum des commits et de merge qui ont affectés les fichiers. Nous présentons également le nombre de blocs et le nombre de développeurs. Chaque développeur travaille sur une réplique, le nombre de développeurs est calculé selon la méthodologie décrite dans la section 3.2.3. Nous utilisons dans notre étude seulement les fichiers qui contiennent au moins un merge.

Les projets récupérés contiennent beaucoup d'éditions. Par exemple, les projets Git et node contiennent plus d'un million de lignes chacun. Le nombre de fusions et de répliques est indépendant du nombre de lignes éditées. Le projet d3 contient presque 1 million de lignes éditées et 30 développeurs mais seulement 38 fichiers avec des merges, alors que dans le projet bootstrap ne dépasse pas les 30 000 lignes avec 10 développeurs mais il contient 69 fichiers avec des merges. Ceci explique que dans certains projets les utilisateurs étaient organisés et chaque utilisateur travaillait dans des fichiers différents.

5.2.2 Métrique du merge

Nous définissons l'effort de l'utilisateur comme la différence entre la fusion simulée par les algorithmes de réplication et le merge effectué par les utilisateurs réels après avoir corrigé les conflits. Nous définissons deux métriques :

— **Merge blocs** : le nombre de blocs différents dans le document fusionné,

— **Merge lignes** : le nombre de lignes dans ces blocs ou le nombre de lignes insérés par le simulateur pour corriger le document.

Par exemple, si l'utilisateur a besoin de trois insertions consécutives et deux suppressions consécutives pour corriger son document. Le nombre de blocs est égale à deux et le nombre de lignes à cinq.

Remarque. *On ne prend pas en considération dans la métrique les marqueurs ajoutés par le mécanisme du merge (ligne commençant par "`>>>>>>>>`", "`<<<<<<<<`" et "`========`").*

5.2.3 git-Merge

Pour pouvoir comparer les résultats produits par les algorithmes de réplication operation-based et l'algorithme git-merge utilisé par le système git, il faut mettre en place un outil qui rejoue l'historique des répertoires git avec git-merge. Nous avons donc implémenté l'algorithme qui rejoue l'historique des répertoires git par lui même. Cet algorithme traverse l'historique des répertoires et calcule la différence entre l'état du document après la fusion automatique de git avec l'état du document après un merge [24]. Puisque git stocke seulement l'état du document

23. `https://github.com/popular/starred`, Décembre 2012
24. Pour rappel, merge est une fusion suivie de résolution de conflits

après la fusion et la correction des conflits, donc la différence entre la fusion automatique et le merge est une liste de lignes que l'utilisateur avait corrigé.

Le nombre de merge blocs et de merge lignes des différents projets git produits par git-merge sont présentés dans le tableau 5.1. Dans ce qui suit, nous essayons de comprendre le comportement des algorithmes de réplication lors de la fusion et les cas d'édition qui génèrent les conflits. Ensuite, nous présentons quelques améliorations des algorithmes.

5.3 Étude des conflits

Durant l'exécution des algorithmes de réplication sur les traces Git, nous mesurons le nombre et la taille des conflits. Ces derniers donnent des informations sur les lignes ajoutées ou supprimées mais pas sur les comportements des utilisateurs qui ont générés ces conflits.

Pour comprendre le comportement des utilisateurs et les scénarii qui produisent les conflits, nous avons analysé manuellement les traces d'édition qui produisent les conflits. Cette tâche est difficile étant donné qu'il faut suivre les modifications conflictuelles depuis leur génération et comprendre ensuite ce que les utilisateurs voulaient comme résultat.

Dans ce qui suit, nous présentons les comportements les plus fréquents que nous avons pu observer générant des conflits. Cela nous permettra ensuite de comprendre l'origine de ces conflits, de les résoudre avec les algorithmes de réplication et enfin de réduire l'effort des utilisateurs.

5.3.1 Observation de la collaboration

Le comportement des utilisateurs au cours de la collaboration est différent d'un projet à un autre. En effet, plusieurs facteurs peuvent influencer la collaboration tels que le nombre d'utilisateurs, le type de projet, la proximité des utilisateurs, etc. Il est donc difficile de détecter et de savoir quels sont les cas les plus communs qui créent des conflits au cours de la collaboration.

Pour extraire ces scénarii, nous demandons au framework de stocker dans un fichier les modifications produites durant la simulation de l'édition collaborative. Comme nous avons décrit dans la section 3.2.3, le résultat du merge dépend de l'algorithme de réplication utilisé. Une fois la simulation terminée, nous analysons manuellement le fichier stocké. Dans ce qui suit, nous discutons sur les différents scénarios observés.

Mise à jour à la même position

Le conflit se produit lorsque deux utilisateurs modifient en concurrence un texte à la même position (pas nécessairement le même contenu), car il n'y a pas d'ordre entre les opérations.

Ce genre de conflit est mieux maîtrisé par les algorithmes operation-based que state-based. Par exemple, dans la figure 5.3, initialement deux utilisateurs partagent le même document `"int a;"`. Ensuite, l'utilisateur 1 met à jour par `"int a=0;"` lorsque l'utilisateur 2 insère en concurrence à la même position `"int x;"`. Après la procédure du merge, le système Git ne peut pas fusionner les deux documents puisque les deux utilisateurs apportent des modifications dans la même position. Un conflit est généré. Le résultat produit par les algorithmes de réplication operation-based dépend de comment l'opération *update* est exécutée. En effet, certains algorithmes gèrent l'opération *update* correctement, alors que d'autres la transforme en suppression suivie d'une insertion. Donc, les algorithmes operation-based peuvent produire deux résultats :

1. si l'algorithme supporte les opérations *update*, dans ce cas le merge produit un résultat correct et aucun conflit n'est généré ;

2. si l'algorithme operation-based transforme l'opération *update* en suppression suivie d'une insertion, deux résultats peuvent apparaître, puisqu'il n'y a pas d'ordre entre les deux insertions :
   (a) lorsque l'utilisateur 2 reçoit l'opération distante, il supprime `"int a;"` et insère `"int a=0;"` dans la bonne position. L'utilisateur 1 exécute l'insertion reçue `"int x;"` dans la bonne position. Dans ce cas le merge est correct et aucun conflit n'est généré,
   (b) puisqu'aucun ordre n'est défini entre les deux insertions, il est possible de produire le document `""int a=0;" "int x;"` au lieu de `"int x;" "int a=0;"`.

Dans le pire cas, les algorithmes operation-based nécessitent deux modifications (une suppression et une insertion). Alors qu'en utilisant une approche state-based, les utilisateurs ont besoin au minimum d'une seule modification comme pour l'utilisateur 2 dans la figure 5.3, et au pire cas trois modifications comme pour l'utilisateur 1.

Dans cet exemple, la différence entre les deux approches n'est pas très grande (deux contre quatre modifications), mais cette différence est plus importante dans une véritable édition collaborative, puisque les utilisateurs produisent parfois de nombreuses opérations de copier/coller.

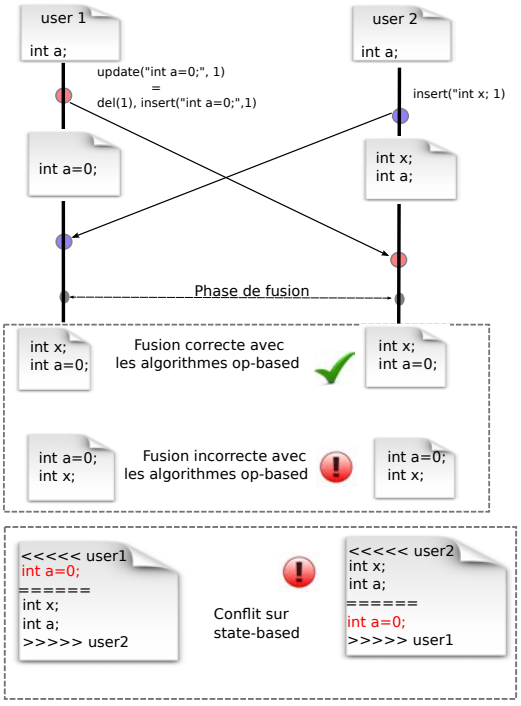


FIGURE 5.3 – Mise à jour à la même position

Accidental Clean Merge (ACM)

On appelle Accidental Clean Merge (ACM) le cas où les utilisateurs insèrent le même contenu dans la même position. git-Merge gère bien ce type de conflit comme le montre la figure 5.4. Les algorithmes de réplication operation-based tiennent compte des modifications par ligne, une nouvelle opération est générée pour chaque ligne. Après la fusion, les lignes sont dupliquées dans le document. Les utilisateurs doivent corriger ligne par ligne.

Dans la figure 5.4, deux utilisateurs insèrent en concurrence le même contenu à la même position (position 2). git-Merge détecte que les deux lignes sont identiques et fusionne les deux

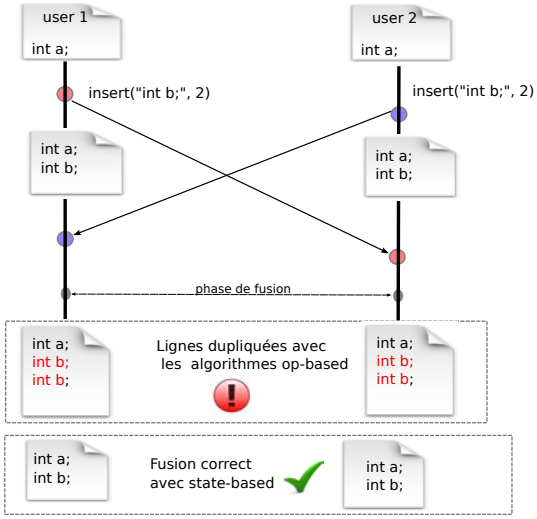


FIGURE 5.4 – Accidental clean merge

modifications correctement. Alors que les algorithmes de réplication operation-based, génèrent des lignes dupliquées `"int b;" "int b;"` puisqu'il génère une nouvelle opération pour chaque ligne.

Undo/Redo

Les opérations undo/redo sont très utiles dans les systèmes d'édition collaborative. Ils permettent aux utilisateurs de revenir sur une modification effectuée à tout moment. Dans le système Git, une opération undo/redo est générée lorsque les utilisateurs *revert* les modifications [25]. Une annulation [26] d'une modification avec les algorithmes de réplication operation-based engendre la génération d'une nouvelle modification. Le type de cette modification dépend de la modification annulée. Par exemple, si une suppression est annulée, une nouvelle insertion est générée.

Dans la figure 5.5, on présente le cas ou git-Merge gère bien le undo/redo alors que les approches operation-based créent des conflits. Initialement, les sites 1 et 2 partagent le même document `"int a;""int b;"`. Le site 1 supprime la ligne 2 dans l'intention de créer le document `"int a;"`. En concurrence, le site 2 supprime la même ligne, annule cette opération et revient sur le document initial. Durant la phase de merge, git-Merge fusionne correctement les modifications et produit le document `"int a;"`. Alors que les algorithmes operation-based créent un conflit dans le document. En effet, lorsque le site 1 reçoit les opérations du site 2, il réinsère `"int b;"` puisque le site 2 a annulé la suppression. Ainsi, les utilisateurs produisent le document `"int a;" "int b;"`. Pour avoir le même résultat que dans l'historique de Git et respecter l'intention des utilisateurs, ils doivent supprimer `"int b;"` de leur document. Cet exemple démontre que l'utilisation des algorithmes de réplication operation-based tels qu'ils sont définis n'est pas satisfaisante dans ce cas. Adapter ces algorithmes pour le undo/redo est nécessaire pour réduire les conflits.

25. commande "git revert"
26. CTRL+Z/Y

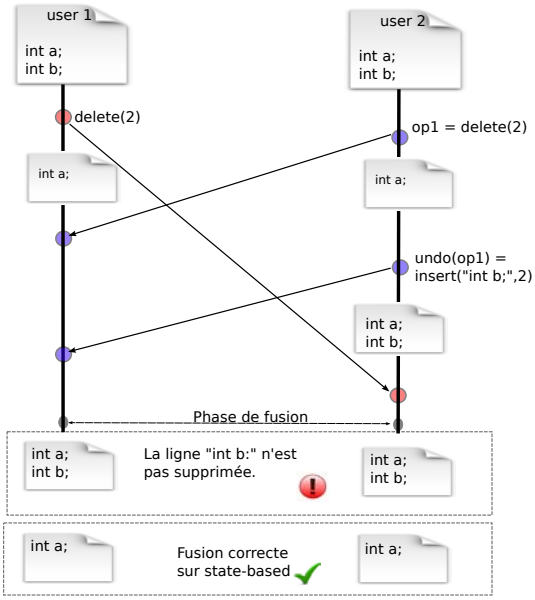


FIGURE 5.5 – opération undo/redo

5.3.2 Résultats Quantitatifs

Dans le tableau 5.1, nous avons présenté le nombre de *merge blocs* et de *merge lignes* produit par gitMerge. Pour comparer les résultats avec les approches operation-based, nous présentons dans les figures 5.6 et 5.7 les métriques *merge bloc* et *merge ligne* des différentes approches sur tous les dépôts Git.

Comme git-Merge est l'algorithme utilisé par défaut dans le système Git pour merger les modifications, nous présentons les résultats de git-Merge comme la référence (= 100 %). La figure 5.8 présente la somme des métriques dans tous les répertoires.

Analyse statistique Pour vérifier si les résultats obtenus sont significatifs, nous effectuons un test de statistique. Étant donné que les projets Git que nous avons utilisés sont indépendants, une méthode d'analyse non paramétrique serait l'approche la plus appropriée pour l'analyse. En utilisant le test *Kruskal-Wallis*[27], nous observons que tous les algorithmes de réplication operation-based surpassent git-Merge, et les résultats obtenus sont très significatifs[28].

Pour le merge bloc, le gain moyen est entre 31% et 33% et p-value est de 0.004 pour tous les algorithmes operation-based, à l'exception de Logoot. Même si le gain moyen obtenu par Logoot est de 5%, le résultat reste significatif (p-value=0,00019). De plus, la différence du merge obtenu entre les approches operation-based est très significative statistiquement (p-value $<$ 0,001).

Le gain moyen en lignes est entre 32% et 35% pour tous les algorithmes operation-based, y compris Logoot. Cette différence est très significative p-value = 0. La différence en merge lignes entre les algorithmes de réplication operation-based est de 3% mais aussi très significative (p-value = 0,00235).

Nous avons présenté une analyse statistique des résultats obtenus pour tous les répertoires, mais le framework donne des résultats par fichier. Un test par fichier nous permet de savoir dans quel fichier le résultat est le plus significatif. Ceci nous aide à chercher seulement sur des fichiers

27. Le test de Kruskal-Wallis ne suppose pas que les données sont distribuées suivant la loi normale.
28. résultat significatif si la variable de signification $p - value$ est inférieur à $0,05$

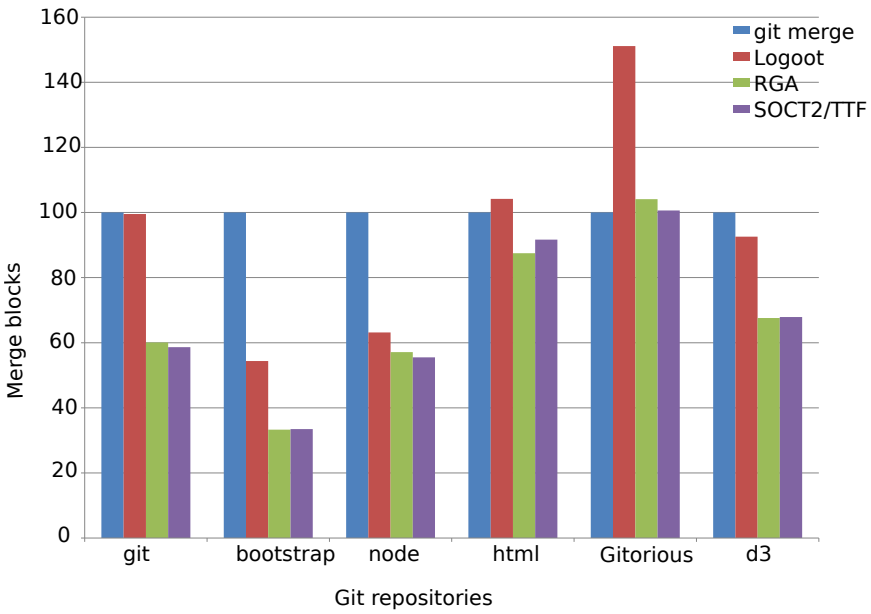


FIGURE 5.6 – merge blocs

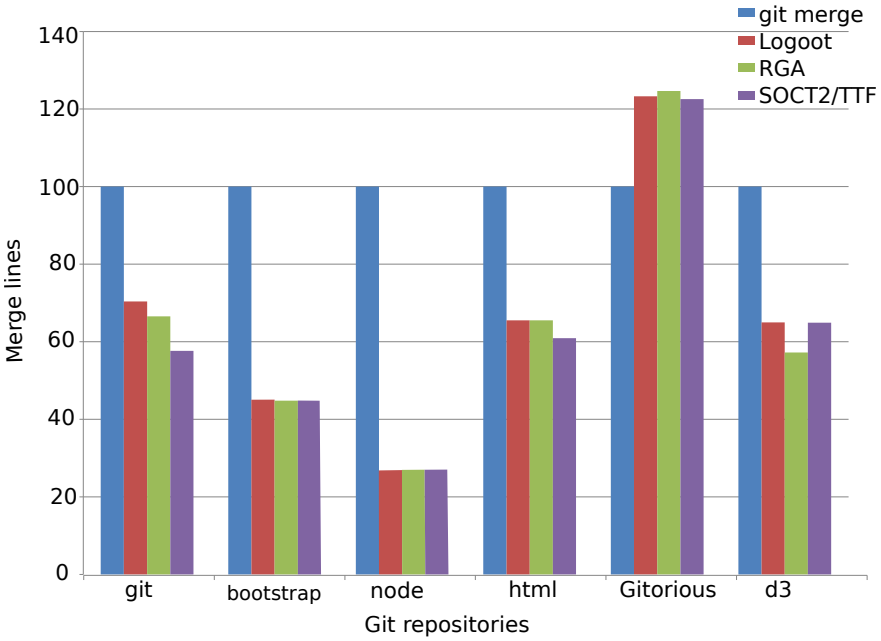


FIGURE 5.7 – merge lignes

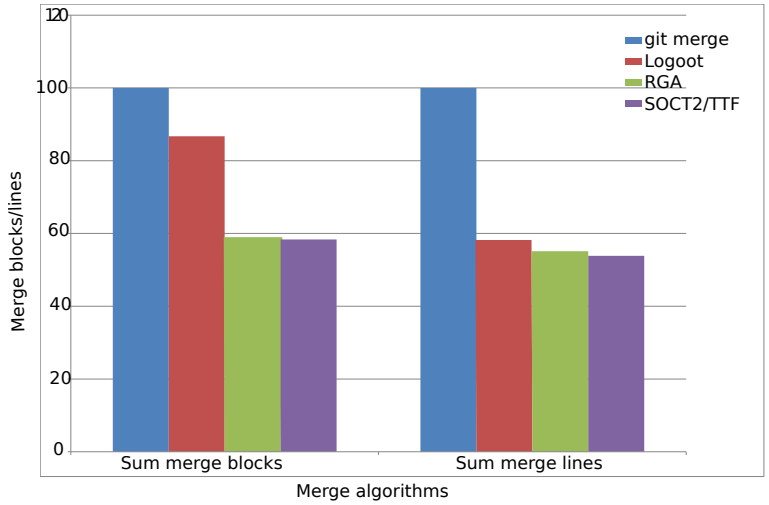


FIGURE 5.8 – La somme des merges bloc/ligne sur tous les répertoires

spécifiques pour comprendre dans quel type de collaboration les utilisateurs introduisent plus de corrections.

Dans ce qui suit, nous allons expliquer pourquoi une telle différence existe entre les algorithmes et pourquoi dans Logoot nous obtenons un résultat différent.

5.3.3 Résultats Qualitatifs

Différence entre git et operation-based La différence est due à un type de collaboration très courant : des modifications concurrentes effectuées sur deux blocs consécutifs. dans la figure 5.9, l'utilisateur 2 et l'utilisateur 3 partagent le même document `A` contenant quatre lignes. Ensuite, l'utilisateur 2 met à jour deux lignes (soit la ligne 1 et la ligne 2) et produit le document `B`. En concurrence, l'utilisateur 3 met à jour deux lignes (soit la ligne 3 et la ligne 4). l'utilisateur 3 commit ses modifications et produit le documents `C`. Au cours de la procédure du merge, git-Merge crée un conflit, même si les deux modifications sont effectuées sur des positions différentes. L'algorithme git-Merge est basé sur des états. Il traite les modifications contigues comme des modifications effectuées sur la même zone du document.

git-Merge renvoie le code `B'` à l'utilisateur 2 et `C'` à l'utilisateur 3. Pour obtenir le code final, le développeur qui exploite le merge doit modifier au moins 1 bloc et 4 lignes. Contrairement à git-Merge, les algorithmes de réplication operation-based fusionnent automatiquement et correctement les modifications. Ce scénario est connu [74] et est l'un des avantages de l'utilisation des algorithmes de réplication operation-based pour le merge. Nous avons analysé dans l'historique de Git, les fichiers où git ne merge pas correctement et au moins un algorithme de réplication operation-based merge correctement. Nous avons trouvé dans 87% des fichiers – parmi ceux choisis aléatoirement [29] – ce type de comportement. Dans ces fichiers, les utilisateurs font des corrections manuellement sur les deux modifications, au lieu de choisir une version complète. Cette analyse montre clairement et pour la première fois, l'efficacité des algorithmes de réplication operation-based pour le merge.

Différence entre les algorithmes de réplication operation-based Nous avons constaté que cette différence est due à des modifications concurrentes affectées sur le document à la même

29. Les quinze premiers fichiers dans l'ordre alphabétique dans le dépôt git.

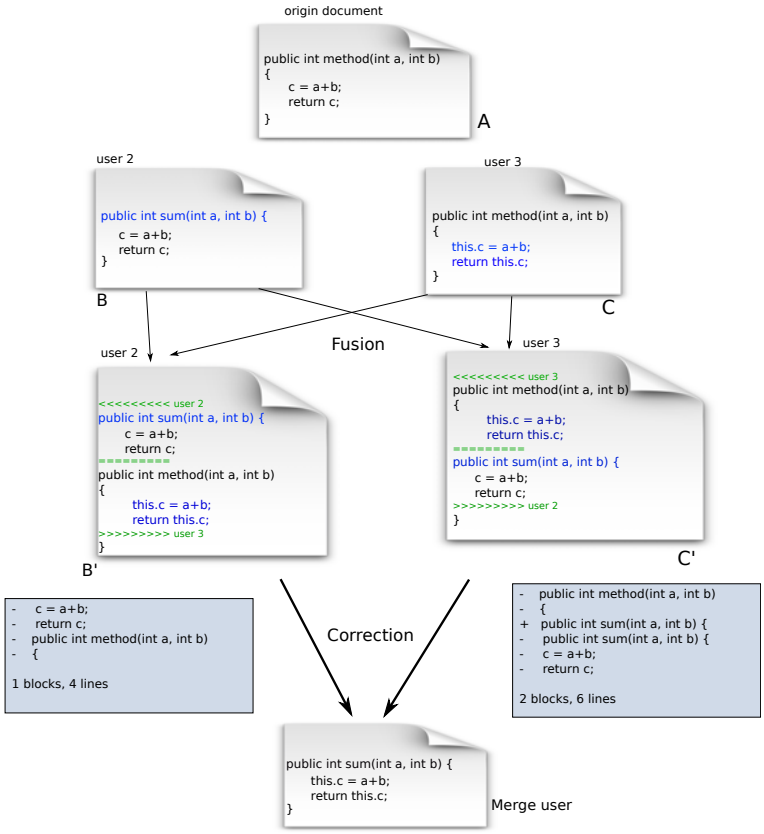


FIGURE 5.9 – Modification sur deux blocs consécutifs

position. Logoot utilise une méthode aléatoire pour générer des identifiants. Les lignes ajoutées en concurrence se chevauchent fréquemment et elles sont entremêlées dans le document. Si le développeur doit garder une seul ligne parmi les lignes dupliquées, il doit supprimer chaque ligne séparément au lieu de supprimer un seul bloc. Ainsi, Logoot obtient les pires performances. Dans le scénario illustré dans la figure 5.10, deux développeurs insèrent en concurrence et à la même position deux blocs différents. Puisque Logoot crée des identifiants aléatoires pour chaque ligne. Les différentes lignes des blocs sont mixées. Par conséquence, le développeur doit éditer 4 blocs et 4 lignes pour corriger le document. En utilisant d'autres algorithmes de réplication operation-based comme RGA et TTF, l'ordre entre les lignes concurrentes est déterminé par l'identifiant de la réplique. Dans le cas de conflit entre deux blocs contigus, l'utilisateur doit modifier seulement un bloc qui contient 4 lignes.

Nous avons également remarqué une légère différence entre RGA et SOCT2/TTF. Cette différence est due à la gestion des insertions concurrentes à la même position. SOCT2/TTF donne la priorité selon l'identifiant de la réplique alors que RGA selon la structure de s4vector du noeud généré.

Répertoire Gitorious Pour les deux métriques - blocs et des lignes -, le comportement des algorithmes de réplication operation-based sur le répertoire Gitorious est différent des autres dépôts. Tous les algorithmes de réplication operation-based sont moins efficaces que git-Merge. La moitié des merge blocs et merge lignes obtenues sont dues à un comportement de collaboration spécifique sur le fichier "`diff_browser.js`". La collaboration dans ce fichier commence

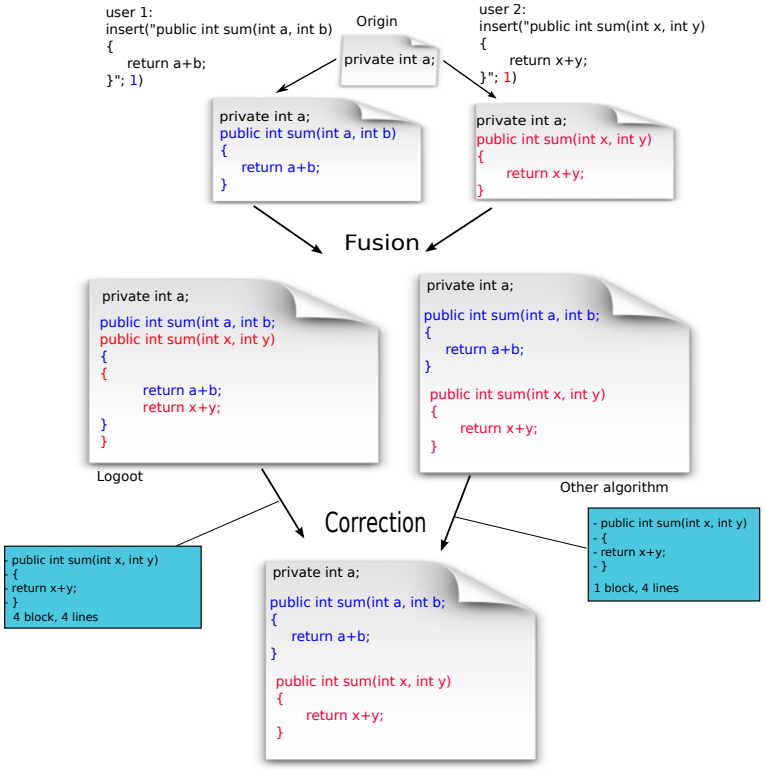


FIGURE 5.10 – Le merge sur les algorithmes de réplication operation-based

par une fusion de deux branches qui n'ont pas d'ancêtre commun. Cependant, ces deux branches contiennent du code avec de nombreuses lignes en commun (accidental clean merge). Comme nous l'avons vu précédemment, git-Merge fusionne correctement ce type d'édition.

Nous avons analysé, dans les différents projets, l'historique des fichiers où git-Merge surpasse les algorithmes de réplication operation-based. Nous avons remarqué que le nombre de commits qui annule (*revert*) d'autres commits peut être élevé dans les dépôts étudiés (parfois la moitié du nombre de commit de merge). Cela peut conduire à des problèmes bien connus comme le undo puzzles [103] présenté dans la figure 5.5. Il existe plusieurs mécanismes d'annulation pour les approches operation-based [103, 120] et elles doivent être évaluées dans notre framework.

Maintenant que nous avons analysé le comportement des algorithmes de réplication et que nous avons montré dans quel cas les conflits sont plus susceptibles d'être générés, il est important de vérifier s'il est possible d'améliorer leur comportement. Dans ce qui suit, nous proposons quelques améliorations.

5.4 Adapter le Merge

Pour améliorer les performances des algorithmes de réplication operation-based dans les systèmes asynchrones, nous devons éviter "la plupart" des conflits fréquents : mise à jour concurrente, Acciental clean merge et undo/redo. Dans ce qui suit, nous adaptons l'approche Tombstone Transformation Functions (TTF)[83] (décrit dans la section 2.3.1) pour éviter ce genre de conflits. Comme TTF est un algorithme OT, nous devons ajouter des fonctions de transformation pour chaque cas.

5.4.1 Améliorer le merge : opération de mise à jour

Lors des mises à jours concurrentes, certains algorithmes de réplication operation-based ordonnent mieux les lignes que d'autres. Cependant, nous n'avons observé aucun algorithme qui les ordonne toutes correctement. Par exemple, dans la figure 5.11, les lignes `"int x;"` et `"int a = 0;"` ont de nouveaux identifiants et entre ces identifiants il n'y a pas d'ordre puisque les opérations sont concurrentes. Par conséquent les lignes se chevauchent. La solution est de réutiliser les mêmes identifiants.

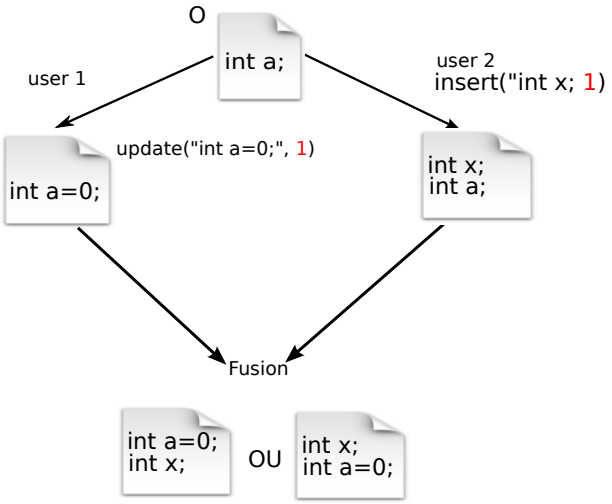


FIGURE 5.11 – Mis à jour dans les algorithmes de réplication operation-based

Dans ce qui suit, nous ajoutons une opération de mise à jour (*update*) pour TTF. L'opération de mise à jour conserve la même position et modifie seulement le contenu. Ceci garantit que les modifications concurrentes insertion/update ne se chevauchent jamais. Une modification de suppression est considérée comme une mise à jour mais avec une valeur *null*. La gestion des modifications concurrentes entre suppression et update peut se faire de deux manières différentes : donner la priorité selon l'identifiant de la réplique ou bien donner la priorité à la suppression. Dans ce qui suit, nous allons mener deux expériences sur le dépôt git :

1. Update : les mises à jours concurrentes avec les suppressions sont gérées selon l'identifiant de la réplique. La fonction de transformation est donnée dans l'algorithme 3 ;

Algorithm 3 Transformation update avec update

```
1: function TRANSFORM(upd(p1,Site_i), upd(p_2, Site_j))
2:     if ( (p_1 <> p_2) or (p_1=p_2 et Site_i > Site_j) ) then return upd(p1,Site_i) ;
3:     else return noop() ; // aucun effet
```

2. delWin : Nous donnons la priorité à la suppression en cas d'opération concurrente delete/update. Les fonctions de transformation sont données dans l'algorithme 4 ;

La figure 5.12 présente le nombre de *merge bloc* et *merge ligne* de notre TTF adapté et TTF original sur le répertoire git.

Lorsqu'un conflit se produit entre suppression et mise à jour, donner la priorité aux opérations de suppression ne réduit pas beaucoup le nombre de conflits. Par contre, garder le même identi-

Algorithm 4 Transformation update avec suppression

```
1: function TRANSFORM(op1, op2)
2:     Soit t1 et t2 respectivement le type de op1 et op2
3:     Soit p1 et p2 respectivement la position de op1 et op2
4:     if ( t1 = del et t2 = upd ) then return del(p1,Site_i) ;
5:     else if (t1 = upd et t2 = del) then
6:         if (p1 <> p2) then return upd(p1,Site_i) ;
7:         else return noop() ;
```

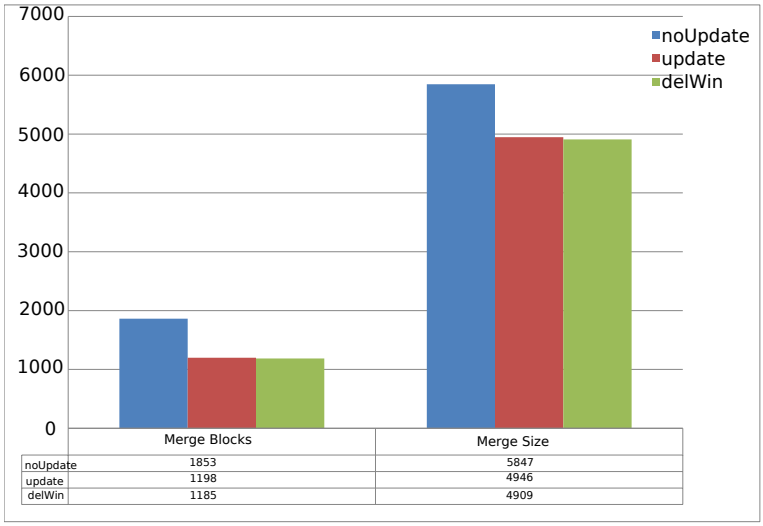


FIGURE 5.12 – TTF sur le répertoire git

fiant conduit à une amélioration remarquable. En effet, garder la même position pour l'opération de mise à jour permet de réduire 36 % de blocs et 16 % des lignes.

5.4.2 Adapter le undo/redo

Les opérations undo/redo dans l'édition collaborative sont très utiles, mais considérées comme un problème difficile [117, 120, 70, 22, 66]. Dans le système Git, la seule information qui peut être utile pour détecter une véritable opération undo/redo, est le message introduit par les utilisateurs quand ils annulent leur modification. Malheureusement, les utilisateurs ne spécifient pas dans leurs messages que c'est une opération d'annulation.

Pour simplifier l'undo, nous supposons que toutes les opérations de suppression sont considérées comme *undo* d'une opération insertion. Avant d'insérer une opération dans le document, nous vérifions si l'opération est un *redo* ou une simple insertion suivant l'algorithme 5. L'algorithme reçoit deux arguments : la position d'insertion et le contenu de l'insertion. Il renvoie l'opération à exécuter dans le document et à envoyer aux autres répliques. Dans la ligne 2, l'algorithme vérifie s'il existe à la même position le même contenu dans une pierre tombale (invisible pour les utilisateurs). Si la condition est vérifiée, cette opération est considérée comme *redo*, dans l'autre cas, elle est considérée comme une simple insertion. Il est possible que l'utilisateur supprime un élément, et qu'il le réinsère à la même position sans utiliser un *redo*. Dans une collaboration réelle, il y a peu de chance d'avoir ce cas.

Pour gérer les opérations undo/redo, l'algorithme utilise le *degré de visibilité* [120]. Lorsqu'une

Algorithm 5 Détection des redo

```
1: function LocalInsertion(pos, pos)
2:     if (getDoc(pos).visibility = false) and (getDoc(pos) == content) then
3: return redo(position, content) ;
4:     else return insert(position, content) ;
```

ligne est créée, elle a un *degré* de visibilité de 1. Chaque fois que la ligne est supprimée, l'algorithme réduit son degré de visibilité. Lorsqu'une suppression est annulée ou une réinsertion est effectuée, l'algorithme augmente son degré de visibilité. La ligne n'est visible que si son degré de visibilité est strictement supérieur à 0.

5.4.3 Adapter l'Accidental Clean Merge (ACM)

ACM se produit lorsque les utilisateurs insèrent en concurrence le même contenu à la même position. Après la phase du merge, le document fusionné peut contenir des lignes dupliquées. Les algorithmes OT (décrit dans la section 2.3.1) peuvent être utilisés pour éviter ce type de conflit. Durant la phase de transformation, les algorithmes OT détectent les ACM et peuvent les transformer en opérations *noop* (opérations sans aucun effet). Les algorithmes CRDT peuvent aussi détecter les ACM mais ils nécessitent plus de traitement. En effet, pour détecter les ACM avec les CRDT, il faut détecter les opérations concurrentes. Les algorithmes CRDT ont été proposés pour que les opérations concurrentes soient indépendantes.

Lorsque deux insertions concurrentes affectent le document à la même position, TTF et d'autres algorithmes OT utilisent l'identifiant des répliques pour donner une priorité à une des deux opérations [102, 88]. Utiliser cette solution pour éviter le cas des ACM, peut créer une divergence telle que présentée dans la figure 5.13.

Trois sites partagent initialement le même document `"ABC"`. Le site 0 et le site 2 insèrent en même temps le même élément "X" à la position 1 et produisent le document `"AXBC"`. Alors que, le site 1 insère en concurrence "Y" à la position 2 et produit le document `"AYBC"` . Pour éviter le conflit de type ACM, quand le site 0 reçoit l'opération du site 2, il n'exécute pas l'opération puisque les deux utilisateurs insèrent le même contenu à la même position. Cependant, lorsque le site 0 reçoit l'opération du site 1, il détecte que les deux opérations ont la même position. Puisque les algorithmes OT donnent la priorité en fonction de l'identifiant de la réplique. Lorsque op2 est transformée avec op1, l'insertion de op2 devient à la position 2 au lieu de la position 1. Le site 0 produit le document `"AXYBC"`. D'autre part, lorsque le site 1 reçoit op1 du site 0, elle n'est pas transformée, puisque la priorité est donnée pour le site 0. Ainsi, le site 1 produit le document `"AXYBC"`. Ensuite, lorsque le site 1 reçoit l'opération du site 2, il la transforme en une opération d'insertion "X" à la position 3 et produit le document `"AXYXBC"`. Sur le site 1, le ACM n'est pas détectée et les répliques divergent.

Pour cette raison, nous proposons une solution en donnant la priorité à l'élément inséré et non pas à l'identifiant de la réplique, par exemple, le hash code du contenu de l'opération. Durant la phase de transformation, nous ajoutons un nouveau test pour détecter les ACM. L'algorithme 6 teste dans la ligne 5 et 6 s'il existe deux éléments avec le même contenu inséré en concurrence à la même position. Dans ce cas, il renvoie une opération *noop*. Dans l'autre cas, l'opération est transformée en comparant la position et le hash code du contenu. En appliquant l'algorithme 6 sur la figure 5.13, le problème de divergence est résolu. En effet, lorsque le site 1 reçoit l'opération du site 2, l'insertion de "X" est transformée à la position 2 au lieu de position 3 puisque le hash code de "X" est inférieur à celui de "Y". Ainsi, l'algorithme détecte qu'il y a deux insertions

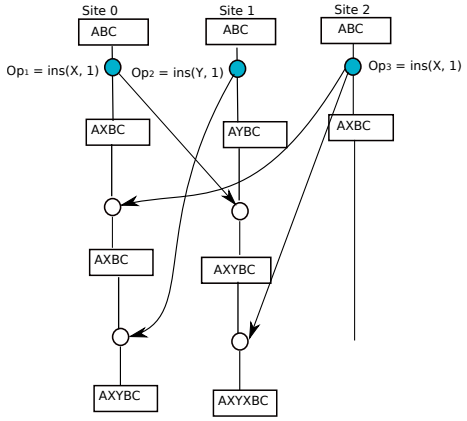


FIGURE 5.13 – OT crée une divergence

de "X" à la même position. Le site 1 détecte l'accidental clean merge et l'opération 3 n'est pas exécutée. Finalement, toutes les répliques convergent et produisent le document `"AXYBC"`.

Algorithm 6 Détection ACM

```
1: function TRANSFORM(op1, op2)
2:     Let c1 and c2 respectivement le contenu de op1 et op2
3:     Let t1 and t2 respectivement le type de op1 et op2
4:     Let p1 and p2 respectivement la position de op1 et op2
5:     if (t1 = insert) and (t2 = insert) then
6:         if (c1=c2) and (p1=p2) then return noop() ;
7:         else if (p1 > p2) or (p1=p2 and HashCode(c1) > HashCode(c2)) then
8: return insert(c1, p1+1,Site_i) ;
9:         else return insert(c1, p1,Site_i) ;
```

Dans ce qui suit, nous allons présenter une étude expérimentale pour comparer notre solution avec les approches operation-based existantes et avec git-Merge du système Git.

5.4.4 Évaluation expérimentale

Pour vérifier si la solution que nous avons proposé réduit le nombre de conflits, nous exécutons quatre algorithmes : git-Merge, TTF, WOOTH et notre solution sur huit répertoires Git.

Puisque l'expérimentation présentée dans la section 5.3 et cette expérience se sont déroulées sur des périodes différentes. Nous allons récupérer les huit répertoires Git de GitHub et Gitorious selon leur popularité au moment de cette expérience. Nous avons récupéré trois projets de l'expérience précédente : bootstrap, d3 et git, et cinq nouveaux répertoires présentés avec leur caractéristiques dans le tableau 5.2.

Pendant l'expérience, nous avons calculé le nombre d'ACM et de undo/redo. Le résultat est présenté dans le tableau 5.3.

Le nombre de *merge bloc* et *merge lignes* présentés dans les tableaux 5.1 et 5.2 sont bien corrélés avec le nombre d'ACM et les opérations undo/redo présentés dans le tableau 5.3. Plus le nombre des ACM et d'opérations undo/redo augmentent et plus notre solution est meilleure par rapport aux autres algorithmes. Par exemple, dans le dépôt git, nous avons détecté 1 272 ACM et 1 614 undo/redo, le gain en terme d'effort obtenu par notre solution est d'environ 54%

Project	cloud/backbone	gitorious/mainline	jquery/jquery	rails/rails	statusnet/mainline
Head sha1	6ac7704c	c1105eb	2f2e045	36f7732	d7880c1
Fichier avec Merge	11	72	29	352	213
Commits	2293	4136	5035	28895	12057
Merge	274	151	178	1153	1218
Nbr.Opérations	2605	3915	5386	26899	11953
Max. Répliques	13	5	12	6	11
Merge bloc avec git-Merge	155	489	458	442	1159
Merge ligne avec git-Merge	895	2303	2146	3899	4783

Table 5.2 – Caractéristiques des répertoires git

Projet	Accidental Clean Merge	undo	redo
Git	1272	42734	1614
bootstrap	563	7210	3957
backbone	271	1357	1137
d3	7	19877	218
Gitorious	750	932	513
jquery	213	1947	1432
rails	426	5329	16172
status	2297	9060	6352

Table 5.3 – ACM and Undo/Redo sur répertoire git

sur le dépôt git (figure 5.14).

Même si les algorithmes OT et CRDT fusionnent les modifications de manière complètement différente, le résultat de TTF et WOOTH sont presque les mêmes dans *merge bloc* et *merge lignes*. Changer la manière de gérer les opérations (transformation ou identifiant unique) n'est pas suffisante pour améliorer la qualité du merge et de réduire l'effort des utilisateurs.

L'algorithme CMUndo implémente plusieurs fonctions pour détecter les deux cas de conflits : ACM et undo/redo. Dans la figure 5.14 et sur les répertoires qui contiennent un grand nombre d'ACM et d'opérations undo/redo, git-Merge surpasse les algorithmes TTF et WOOTH mais il reste moins performant que l'algorithme CMUndo. En effet, TTF et WOOTH ne gère pas undo/redo et ACM, alors que git-Merge peut les fusionner correctement et récupérer quelques lignes identiques lorsque deux blocs sont insérés en concurrence. À l'exception de répertoire Gitorious, CMUndo réduit l'effort des utilisateurs sur tous les autres répertoires.

En utilisant l'algorithme git-Merge, le système asynchrone crée plus de conflits que l'algorithme CMUndo. Les modifications ne peuvent pas être fusionnées et les utilisateurs font plus de corrections. Si on compare git-Merge et l'algorithme CMUndo, le gain obtenu est 60% de blocs sur le répertoire jquery, 54% sur le répertoire git, 59 % sur le répertoire bootstrap et il perd 1 % sur le répertoire Gitorious.

Nous avons remarqué que la différence entre l'algorithme CM qui traite seulement les conflits ACM et l'algorithme CMUndo est seulement de 3%. Cela démontre que l'impact des ACM est beaucoup plus important que undo/redo.

Dans la figure 5.15, il est clair que l'algorithme CMUndo est le meilleur. Il surpasse largement tous les autres algorithmes et en particulier git-Merge. Lorsqu'un conflit est généré, il y a plus de probabilité de générer un grand bloc en utilisant des algorithmes state-based que operation-based. En effet, sur les approches operation-based, quelques lignes peuvent être insérées correctement, alors que dans state-based si une seul ligne est en conflit, toute les modifications consécutives sont en conflit. Lorsque les états sont mélangés les utilisateurs ont besoin de beaucoup de corrections. Pour cette raison, les algorithmes TTF et WOOTH sont plus performants que git-Merge sur *merge lignes*.

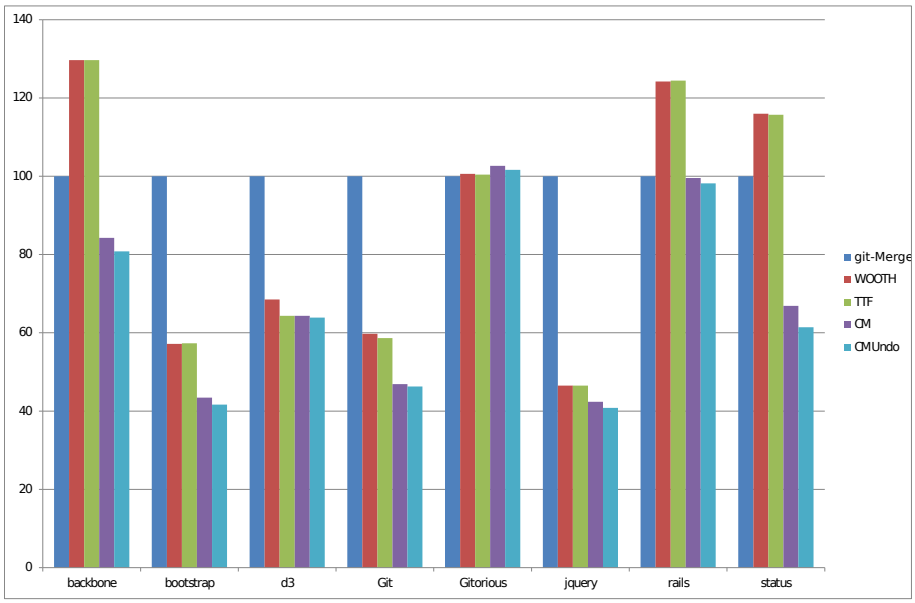


FIGURE 5.14 – Merge blocs sur le projet git

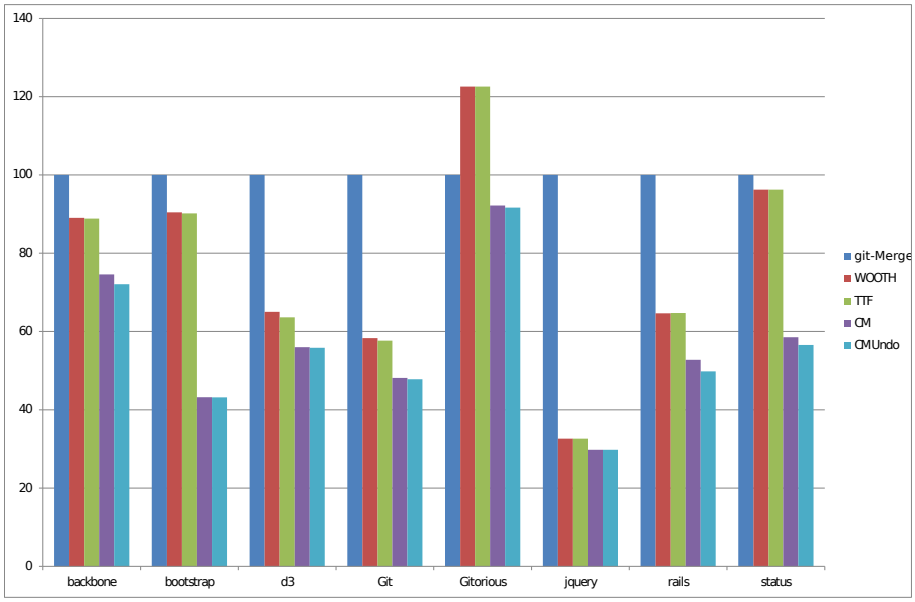


FIGURE 5.15 – Merge lignes sur le projet git

L'utilisation de l'algorithme CMUndo sur un système asynchrone,réduit largement l'effort des utilisateurs. Si on compare avec git-Merge, l'algorithme CMUndo réduit de 70% le nombre de lignes sur le répertoire jquery, 52% sur le répertoire git et 57% sur le répertoire bootstrap.

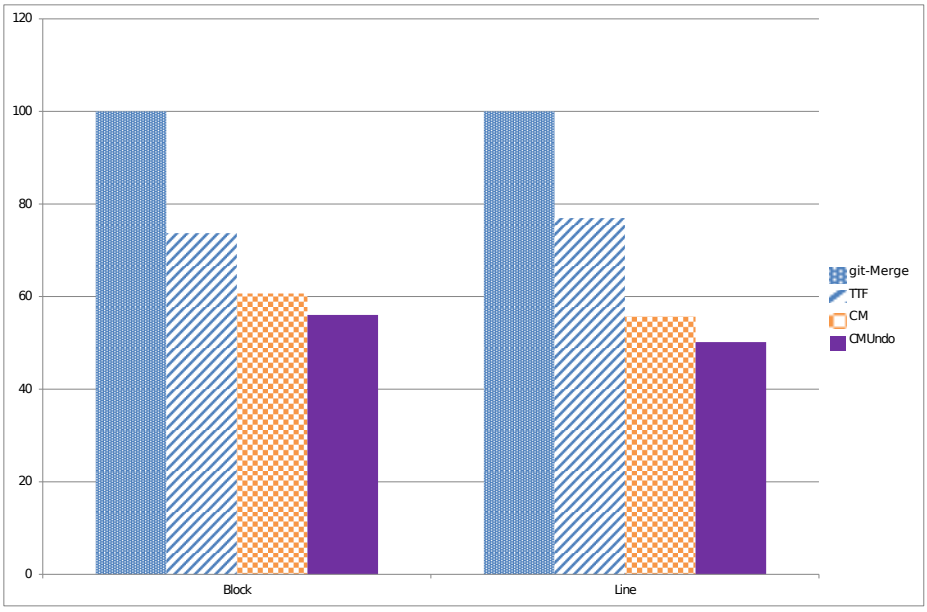


FIGURE 5.16 – Merge bloc et merge ligne total

Gitorious Dans le répertoire Gitorious, git-Merge est plus efficace que tous les algorithmes operation-based, que se soit pour *merge bloc* ou le *merge ligne*. Comme décrit précédemment, cela est dû à un comportement de collaboration spécifique dans le fichier `diff_browser.js`, les utilisateurs insèrent en concurrence presque le même document avec plus de 800 lignes. Au cours de la procédure de fusion git-Merge gère bien ce type d'édition, car il peut détecter quelques zones de texte identique.

Pour résumer notre expérience, nous avons intégré les algorithmes operation-based sur le système d'édition collaborative Git. Nous avons ensuite proposé une métrique qui représente l'effort de l'utilisateur effectué pour résoudre les conflits et corriger le document. Le *merge bloc* est le nombre de blocs de texte en conflit, alors que *merge lignes* est le nombre de lignes à modifier pour résoudre le conflit. Pour comprendre le comportement des utilisateurs et détecter les conflits, nous avons rejoué une édition collaborative sur huit projets. Nous avons récupéré 1 335 fichiers contenant des opérations merge. Sur ces fichiers, nous avons détecté 5 799 ACM et 118 409 opérations undo/redo. Nous avons ensuite adapté un algorithme operation-based appelé CMUndo pour gérer ces conflits. L'utilisation de cet algorithme obtient un gain de 3 583 blocs et 21 675 lignes. La figure 5.16 présente le nombre total des *merge bloc* et *merge lignes*. Nous avons séparé les deux algorithmes (l'algorithme (CM) qui gère seulement les ACM et celui (CMUndo) qui gère de plus les undo/redo de notre approche pour observer l'impact de chacun sur le résultat.

5.5 Conclusion

Fusionner automatiquement les modifications dans les systèmes asynchrones est essentiel dans une collaboration. Cependant, lorsque des modifications concurrentes se produisent, l'outil de fusion peut générer des conflits. Les utilisateurs font alors des efforts pour résoudre ces conflits et corriger leur document. Réduire l'effort des utilisateurs peut améliorer la productivité de l'équipe et encourage les utilisateurs à collaborer.

Dans cette section, nous avons évalué des algorithmes de réplication optimiste sur un système asynchrone, conçu pour l'édition collaborative. Dans un premier temps, nous avons proposé une méthodologie et une métrique pour mesurer la qualité du merge. Nous avons démontré que notre outil d'évaluation permet d'observer la collaboration. Cette étude nous a permis de bien comprendre le comportement des développeurs et de détecter des conflits réels. Nous avons ensuite démontré que nous pouvons réduire les conflits et les interactions humaines par l'utilisation des algorithmes de réplication operation-based sur les systèmes asynchrones. Nous avons montré que notre solution est meilleure que l'outil git-Merge utilisé comme outil de merge par défaut dans le système Git.

L'évaluation des algorithmes de réplication que nous avons menée dans les deux derniers chapitres, nous a démontré que l'application des différents algorithmes dépend du type de la collaboration et du système. Les applications d'édition collaborative actuelles sont déployées sur des architectures coûteuses et complexes. Dans le chapitre suivant, nous proposons une architecture basée sur les deux protocoles que nous avons vus : OT et CRDT.

Chapitre 6

Architecture

Sommaire

Dans les sections 4.1 et 4.2, nous avons mesuré les performances des algorithmes de réplication optimiste. Nous avons trouvé que les performances en temps de calcul des approches CRDT étaient adéquat pour les applications temps réel, mais l'empreinte mémoire occupée par tous les algorithmes destinés aux systèmes pair à pair est élevé. Améliorer et adapter un algorithme pour toutes les exécutions est impossible. Cependant, il est possible d'améliorer les performances du système en changeant la topologie du réseau.

Récemment, des systèmes d'édition collaborative ont été déployés dans les nuages (ou cloud) tels que Google Docs [1] et Microsoft SkyDrive [5]. Dans ces systèmes, les documents sont stockés dans le nuage. Les utilisateurs peuvent conserver des copies locales dans leurs dispositifs pour garantir une faible latence et une haute disponibilité. Cependant, pour garantir la consistance du document, les algorithmes de réplication optimiste sont utilisés.

Dans ce chapitre, nous discuterons des problèmes liés à l'architecture des systèmes d'édition collaborative sur les nuages. Ensuite, nous proposerons une nouvelle architecture pour résoudre certains problèmes. Cette solution est basée sur les algorithmes de réplication optimiste OT et CRDT. Enfin, nous allons évaluer certains algorithmes sur l'architecture proposée.

6.1 Système

Dans les systèmes d'édition collaborative temps-réel, chaque utilisateur doit observer immédiatement ses propres modifications et le plus rapidement possible. Aujourd'hui, un grand nombre d'applications d'édition collaborative hébergent leurs documents dans une infrastructure dite "dans le nuage", tel que Google Drive [1] ou SkyDrive de Microsoft [5].

Un système d'édition collaborative dans les nuages offre aux utilisateurs des ressources limitées. Le nombre d'ordinateurs physiquement mis à disposition aux utilisateurs peut varier en fonction du temps. Les applications déployées sur les nuages doivent donc tolérer la dynamicité de ce réseau. De même, les données étant répliquées, l'informatique dans les nuages requiert des algorithmes de maintien de la cohérence. Pour réduire la latence réseau et augmenter la disponibilité du document les systèmes d'édition collaborative adoptent généralement la réplication optimiste.

Comme nous l'avons décrit dans la section 2.3, les algorithmes OT [34, 105] sont des algorithmes de réplication optimiste. Ils assurent la consistance du document en adaptant les paramètres des opérations distantes afin de prendre en compte les effets des opérations concurrentes. Il existe différentes architectures de déploiement de systèmes OT.

Les architectures OT décentralisées sont complexes à mettre en oeuvre et nécessitent de coûteux mécanismes qui occupent un large espace de stockage et qui ne passent pas à l'échelle tels que les vecteurs d'horloges et l'historique des opérations comme nous l'avons démontré par les expérimentations présentées dans le chapitre 4.

Les architectures centralisées tolèrent un grand nombre de clients mais requièrent un mécanisme pour ordonner les opérations [113]. Cependant, maintenir un ordre total a un coût élevé. Soit cet ordre est maintenu par un site centralisé avec les risques de pannes, soit il est maintenu avec des algorithmes de consensus distribués synchrones. Dans ce cas, le système peut être tolérant aux pannes mais passe difficilement l'échelle [64]. La figure 6.1 présente un exemple d'une architecture d'un système d'édition collaborative sur les nuages. Pour réduire la latence réseau entre des utilisateurs géographiquement distribués, assurer la répartition de la charge et la tolérance aux pannes massives, plusieurs centres de données sont mis à disposition. Les utilisateurs sont connectés avec le centre de données le plus proche. À l'aide d'un consensus distribué [18], un séquenceur central peut être mis en place dans le cadre d'un centre de données pour ordonner les opérations [113].

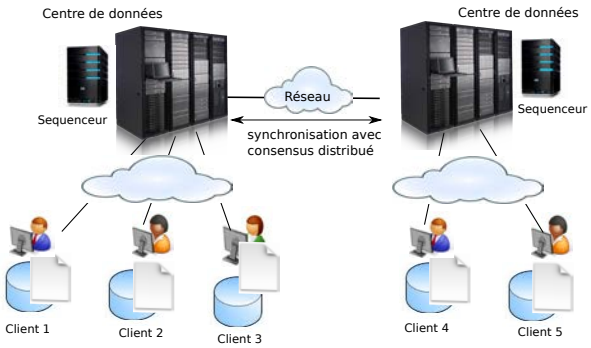


FIGURE 6.1 – architecture avec consensus distribué

Google Doc [1] adopte un algorithme OT centralisé appelé Jupiter [78]. Pour assurer un ordre total entre les opérations, Google Docs utilise un consensus [59, 18] dans chaque centre de données. Les centres de données peuvent être synchronisés en utilisant un Multi-Paxos [32]. Cette solution ne tolère pas les pannes et passe difficilement à l'échelle.

Récemment, Google a introduit une nouvelle approche dite TrueTime (TT) [26] pour assurer un ordre total sans l'utilisation de consensus inter centre de données. Cette approche utilise un GPS et des horloges atomiques pour ordonner les opérations avec une très grande précision. Une telle solution ne peut être reproduite par n'importe quel développeur puisqu'elle est complexe et

coûteuse.

Les approches CRDT décrites dans la section 2.3.2, assurent la convergence des documents sans transformation et sans consensus dans les systèmes pair à pair comme la démontre les expérimentations de la section 4. [99, 100]. Cependant, les algorithmes CRDT utilisent des identifiants uniques pour chaque caractère, ce qui augmente l'espace de mémoire occupée et le temps d'intégration des opérations distantes.

Dans le but de surmonter les inconvénients de la centralisation, de minimiser les ressources sur le dispositif client, d'assurer le contrôle de la concurrence, le passage à l'échelle et de fournir une faible latence, nous proposons une architecture hybride, simple en combinant les deux protocoles (OT et CRDT), de telle façon à éviter le consensus inter centre de données et satisfaire l'utilisateur tout en respectant les contraintes des dispositifs clients. Dans cette architecture, les CRDTs sont utilisés comme un *backbone* pour répliquer les données entre les centres de données. OT est utilisé pour fournir une faible surcharge de réplication entre les clients et le centre de données où ils sont connectés. Dans ce qui suit, nous allons présenter cette architecture.

6.1.1 Plateforme

L'architecture de notre système considère plusieurs centres de données. Nous distinguons les noeuds serveurs qui sont hébergés dans un centre de données et les noeuds clients qui sont les dispositifs de l'utilisateur et qui sont répartis sur l'ensemble de la planète.

Afin de permettre la répartition de charge et de tolérer les pannes, plusieurs serveurs identiques sont déployés dans un centre de données. La cohérence entre les serveurs dans un centre de données particulier peut être assurée par un consensus [18]. Dans ce qui suit, nous considérerons l'ensemble des serveurs d'un centre de données comme un seul noeud serveur.

L'idée de base de notre architecture est la suivante :

— utiliser un algorithme centralisé OT entre les clients et le centre de données de leur choix,

— coupler fortement chaque serveur central OT à une réplique CRDT,

— assurer la cohérence inter centre de données de manière pair-à-pair à l'aide du mécanisme CRDT.

La figure 6.2 montre l'architecture de notre système. Les clients exécutent les opérations localement. Un séquenceur OT se trouve dans chaque centre de données avec un serveur. Il gère l'ordre entre les opérations générées par les clients qui se trouvent dans le même centre de données. Une fois que le serveur reçoit et intègre l'opération OT, il génère une nouvelle opération CRDT correspondante et diffuse cette opération aux autres serveurs. Le serveur qui reçoit l'opération CRDT, génère une nouvelle opération OT correspondante et l'envoie à ses clients. Les deux opérations sont exécutées en isolation pour assurer que les deux états OT et CRDT dans le serveur soient toujours équivalents.

Donc, les clients exécutent un algorithme OT. Alors que les serveurs qui sont dans le centre de données, exécutent deux protocoles : OT pour accepter les opérations des clients qui sont dans la même région ; CRDT pour envoyer et recevoir les opérations entre les centres de données.

L'avantage de cette approche est que le dispositif du client reste léger et le flux de données sur le réseau est réduit, car les clients utilisent un l'algorithme OT centralisé. De plus, le système n'utilise aucun consensus entre les centres de données, car les opérations sont propagées entre les centres de données en utilisant l'approche CRDT. Cependant, l'inconvénient est que les serveurs doivent exécuter chaque opération deux fois, mais nous supposons que les serveurs dans les centres de données possèdent une puissance de calcul suffisante.

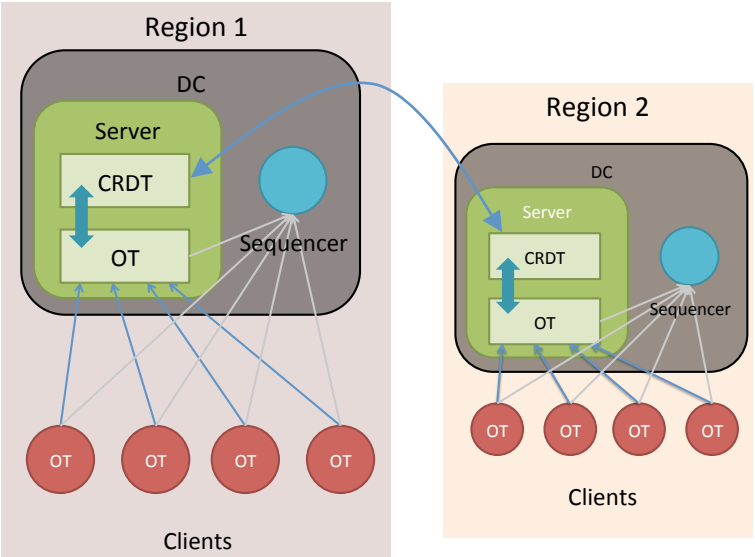


FIGURE 6.2 – Architecture hybride OT/CRDT.

6.1.2 Algorithmes

Les serveurs ont deux parties : une partie OT et une partie CRDT. La partie OT est en charge de la diffusion et de la réception des opérations OT des clients. La partie CRDT est en charge de communiquer avec d'autres parties CRDT qui se trouvent dans d'autres centres de données.

Pour assurer la cohérence de l'architecture, l'état du noeud OT doit être égal à l'état du noeud CRDT tel que présenté dans la figure 6.3.

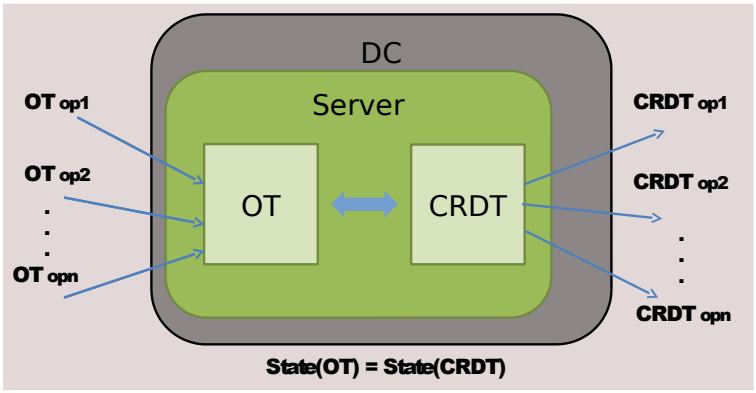


FIGURE 6.3 – Statut des serveurs

Comme les algorithmes OT centralisés, nous utilisons un séquenceur pour ordonner les opérations. Nous proposons dans notre architecture, un séquenceur pour chaque centre de données. Le client OT gère l'ordre des opérations dans son buffer. Lorsque le client produit une opération, il l'ajoute à la fin du buffer et demande un timestamp pour cette opération. Pour répondre à l'intention de l'utilisateur, cette opération est exécutée immédiatement localement mais elle n'est diffusée aux autres clients que lorsqu'elle reçoit un timestamp. Une fois reçue, l'opération est diffusée et supprimée du buffer. Lorsque le client OT reçoit une opération distante il la transforme

avec les opérations concurrentes qui sont dans un buffer en attente d'exécution. Un algorithme OT centralisé qui utilise le même mécanisme est l'algorithme SOCT4 [113] décrit dans la section 2.3.1.

Pour éviter les problèmes de communication entre les clients, toutes les opérations générées par les clients sont envoyées au serveur OT-CRDT qui se charge de la diffusion aux autres clients. L'algorithme SOCT4 original stocke dans son historique tous les opérations produites localement. Dans notre implémentation, *nous sauvegardons dans le buffer des clients seulement les opérations locales qui ne sont pas encore diffusées.* Ainsi, la taille du buffer reste limitée. Cette optimisation est linéaire puisque les opérations sont délivrées suivant un ordre total et SOCT4 original n'utilise jamais les opérations qui se trouve avant le dernier timestamp reçu.

Lorsque le serveur OT-CRDT reçoit une opération d'un client OT, le noeud OT applique l'effet de l'opération sur le noeud CRDT à travers son interface locale. Une opération reçue du client est traitée en trois étapes :

1. le serveur envoie l'opération directement aux autres clients connectés sur le même centre de données,
2. l'opération est exécutée sur le noeud OT et son effet est transmis au noeud CRDT pour générer une nouvelle opération CRDT. L'exécution de l'opération entre le noeud OT et CRDT est atomique,
3. le serveur envoie ensuite l'opération CRDT aux autres centres de données.

Algorithm 7 Réception d'une opération OT

```
1: procedure ReceptionOT(op)
2:     broadcast(clients,op)              ▷ diffuser l'opération aux autres clients
3:     Atomic
4:         ExecutionOT(op)                ▷ exécuter l'opération OT dans l'ordre
5:         opc ← equiv(op)                ▷ produire une opération CRDT correspondante
6:         ExecutionCRDT(opc)             ▷ exécuter l'opération CRDT
7:     End
8:     broadcast(serveurs,opc)            ▷ diffuser l'opération CRDT aux autres data-center
```

Lorsqu'un centre de données reçoit une opération CRDT, le serveur la traite en trois étapes :

1. L'opération est intégrée dans le noeud CRDT et son effet est appliqué sur le noeud OT,
2. le noeud OT exécute localement l'opération et demande un timestamp pour l'ordonner. l'exécution dans le noeud OT et CRDT est atomique.
3. l'opération est ensuite diffusée aux clients connectés sur ce centre de données.

Suivant les étapes décrites précédemment, chaque fois qu'une opération CRDT est exécutée, une opération OT correspondante est aussi exécutée. Cette dernière est diffusée ensuite aux clients connectés sur le centre de données. Les clients et le serveur vont partager donc le même document. Suivant le même principe, lorsque le noeud OT reçoit une opération, il l'exécute localement et une opération CRDT correspondante est générée et exécutée sur le noeud CRDT. L'opération CRDT est diffusée ensuite aux autres centres de données. Suivant cette logique, le noeud OT et CRDT sur le même serveur partagent le même document et l'état des différents serveurs est cohérent.

Algorithm 8 Réception d'une opération CRDT

```
1: procedure ReceptionCRDT(opc)
2:     op.timestamp ← ticket()                ▷ demander un timestamp pour l'opération
3:     when op.timestamp = H_S do
4:         Atomic
5:             ExecutionCRDT(opc)             ▷ exécuter l'opération CRDT dans l'ordre
6:             op.operation ← equivOT(opc)    ▷ produire une opération OT correspondante
7:             ExecutionOT(op)                ▷ exécuter l'opération OT
8:         End
9:         broadcast(clients,op)              ▷ diffuser l'opération OT aux clients
```

6.1.3 Mode déconnecté

Une architecture dédiée à l'édition collaborative doit offrir aux clients la possibilité de travailler en mode déconnecté. Notre architecture permet aux utilisateurs de générer des opérations locales sans accès au séquenceur. Ces opérations sont stockées dans un buffer jusqu'à ce qu'elles reçoivent un timestamp. Lorsque le client se reconnecte sur un centre de données, il peut à nouveau recevoir et émettre des opérations. Un système OT pour la déconnexion/connexion est décrit dans [14].

Cependant, dans le cas où un centre de données n'est pas disponible pour le client, ce dernier ne peut pas se reconnecter directement à un autre centre de données. Pour éviter que les modifications effectuées en mode déconnecté soient perdues, le client doit stocker plus d'informations. L'idée est d'offrir aux clients qui en ont les capacités une réplique CRDT et de transformer le client en mode OT vers CRDT en cas de déconnexion. Pour offrir une faible latence et réduire la surcharge sur le client, un algorithme OT est exécuté. Cependant, en cas de déconnexion du centre de données, le client CRDT est activé [30].

6.1.4 Garantir la cohérence

Notre solution vise à assurer une cohérence stricte inéluctable (ou SEC pour **S**trong **E**ventual **C**onsistency) du document.

Definition 1. cohérence stricte inéluctable*[100] Un objet est strictement inéluctablement consistant, si l'objet est inéluctablement consistant et que les répliques qui ont exécutés les mêmes mises à jour partagent le même état.*

Un type de données répliqué est considéré CRDT s'il atteint la cohérence stricte inéluctable [100]. Les approches OT assurent aussi la SEC, même si les publications originales ne considèrent que la cohérence inéluctable. Cependant, pour atteindre la cohérence stricte inéluctable, les approches OT nécessitent un algorithme d'intégration correct qui transforme les opérations en assurant la causalité et l'ordre total, et des fonctions de transformation qui respectent la propriétés TP_1 (voir section 2.3.1).

Théorème 1. *L'algorithme SOCT4 avec les fonctions de transformation qui respectent TP_1 assure la cohérence stricte inéluctable.*

30. Il y a peu de temps, GoogleDoc avait une fonction similaire où il fallait charger explicitement les documents qui devraient être accessible en hors ligne (mais en lecture seulement)

Démonstration. L'algorithme SOCT4 avec les fonctions de transformation qui respectent TP_1 assure la cohérence inéluctable [113]. Les opérations délivrées aux répliques sont immédiatement exécutées. Donc, les répliques qui diffusent les mêmes opérations, ont les mêmes états. SOCT4 avec TP_1 garantit la cohérence stricte inéluctable. □

Ainsi, dans notre architecture, tous les noeuds OT – y compris celui sur le serveur – partagent le même état à terme. Aussi, tous les noeuds CRDT partagent le même état à terme. Afin de s'assurer que tous les noeuds partagent le même état, nous devons démontrer que les noeuds CRDT et OT couplés partagent aussi un état commun. Nous avons introduit donc, une propriété d'opération équivalente entre les opérations CRDT et OT.

Definition 2. Opération Équivalente *: soit S et S' deux états. Pour chaque opération OT op d'un utilisateur il existe une opération CRDT op' de telle sorte que si $S \equiv S' \Rightarrow S.op \equiv S'.op'$; et réciproquement.*

Avec ces notions, nous sommes en mesure de synchroniser le noeud interne OT et le noeud CRDT du serveur. Et donc, tous les noeuds de notre architecture partagent finalement le même état.

Théorème 2. *Une architecture OT-CRDT avec les fonctions de transformation qui respectent TP_1 et qui assure la propriété d'opération équivalente garantie la consistance stricte inéluctable.*

Démonstration. Avec la propriété d'opération équivalente, à chaque exécution d'une opération sur le noeud CRDT, l'état du noeud OT est automatiquement modifié. Puisque les deux noeuds OT et CRDT exécutent les opérations en isolation de manière atomique, à terme, les autres noeuds liés à ce serveur vont partager le même état. De même, lorsque le noeud OT du serveur reçoit une opération, une nouvelle opération CRDT équivalente est générée et transmise au noeud CRDT et les clients connectés. À terme, tous les noeuds CRDT et tous les clients OT partagent le même état. □

6.2 Expérimentation

Pour montrer l'intérêt de notre approche, nous avons réalisé un prototype du système. Le prototype peut être utilisé avec différents algorithmes CRDT et OT. Dans notre expérience, nous avons utilisé le simulateur décrit dans la section 3.2.4 pour simuler une session d'édition collaborative en temps réel entre les différents clients. Le simulateur demande à chaque client de générer des opérations selon l'algorithme de réplication exécuté sur la copie locale. À la réception de cette opération au serveur, si ce dernier n'exécute pas le même type de données, l'opération est transformée selon le format de l'algorithme exécuté sur le serveur, avant de l'intégrer et de la diffuser aux autres serveurs.

Notre système est composé de deux types de noeuds. Les noeuds OT qui représentent les clients, et les noeuds OT-CRDT pour les serveurs. L'interface de chaque noeud permet l'insertion ou la suppression d'un texte (un caractère ou un bloc de caractères) et l'intégration des opérations distantes. Les serveurs utilisent un système de notification, pour permettre l'échange et la coopération entre les noeuds OT et CRDT.

Architecture hybride sur les clouds

Notre outil propose une architecture hybride sur les clouds, en mettant en oeuvre des instances de machines –client et serveur– réelles.

Initialement, l'outil demande à l'utilisateur de spécifier l'emplacement des machines à lancer et l'algorithme de réplication à exécuter. Ensuite, une image (AMI –Amazon Machines Image–) portant une copie du type de données répliqué est créé sur chaque emplacement spécifié.

L'outil demande ensuite à l'utilisateur le nombre de machines à créer ainsi que d'autres paramètres tels que la clé de sécurité, le nom et le type de l'instance, le nombre de machines à lancer, etc. Un processus est exécuté ensuite pour intégrer sur chaque machine une copie du type de données, et pour assurer la connexion entre les clients et le serveur.

6.2.1 protocole d'exécution

Les noeuds clients exécutent les opérations séquentiellement et les propagent selon l'ordre FIFO vers les serveurs. Le serveur exécute chaque opération reçue du client de façon atomique et isolée. Cela est nécessaire pour garantir que les noeuds OT et CRDT observent le même état du document.

Quand une opération d'un client arrive sur le serveur, elle est exécutée en tant qu'opération distante sur le noeud OT. Dès qu'une opération est reçue, le noeud délivre une notification au noeud CRDT, avec l'opération qu'il doit appliquer. Le noeud OT attend jusqu'à ce que le noeud CRDT termine l'exécution, avant d'exécuter une nouvelle opération. Le noeud CRDT exécute l'opération en tant qu'opération locale. Ensuite, il notifie le noeud OT pour lui annoncer qu'il a terminé l'exécution et que l'opération est envoyée aux autres serveurs.

À la réception de l'opération sur le centre de données. Un processus similaire est déroulé. Seulement cette fois, l'opération est reçue par le noeud CRDT. Ce dernier délivre une notification au noeud OT pour exécuter l'opération. Lorsque le noeud OT est prêt, les noeuds OT et CRDT exécutent l'opération. Le noeud CRDT l'exécute en tant qu'une opération distante et le noeud OT l'exécute en tant qu'une opération locale. Lorsque l'exécution de l'opération se termine, elle est livrée à tous les clients connectés sur le serveur, en respectant l'ordre séquentiel produit au niveau du serveur.

Pour garantir l'atomicité de l'exécution, toutes les opérations distantes reçues lorsque le serveur exécute une opération (que ce soit les opérations OT envoyées par les clients ou les opérations CRDT envoyées par un autre serveur) sont stockées dans une file d'attente. Une fois que le serveur termine l'exécution de l'opération courante, il peut extraire une nouvelle opération de la file d'attente pour l'exécuter.

6.2.2 Algorithmes

Le système supporte n'importe quel algorithme d'édition collaborative OT ou CRDT. Cependant , les algorithmes doivent être compatibles :

Opérations de bloc certaines fonctions de transformations des algorithmes OT supportent les opérations de blocs, mais ce n'est pas le cas pour tous les algorithmes CRDT. Dans le cas où le système utilise un algorithme CRDT qui ne prend pas en charge les opérations de bloc, le noeud client doit décomposer automatiquement la modification de bloc en une suite de modifications de caractère,

Résolution de conflit Pour garantir que les noeuds CRDT et OT produisent la même séquence d'opérations, la politique de contrôle des opérations concurrentes doit produire un résultat identique. Par exemple, lorsque deux clients insèrent en concurrence le même caractère à la même position, les deux algorithmes doivent éliminer ou conserver les deux caractères.

6.2.3 Évaluation

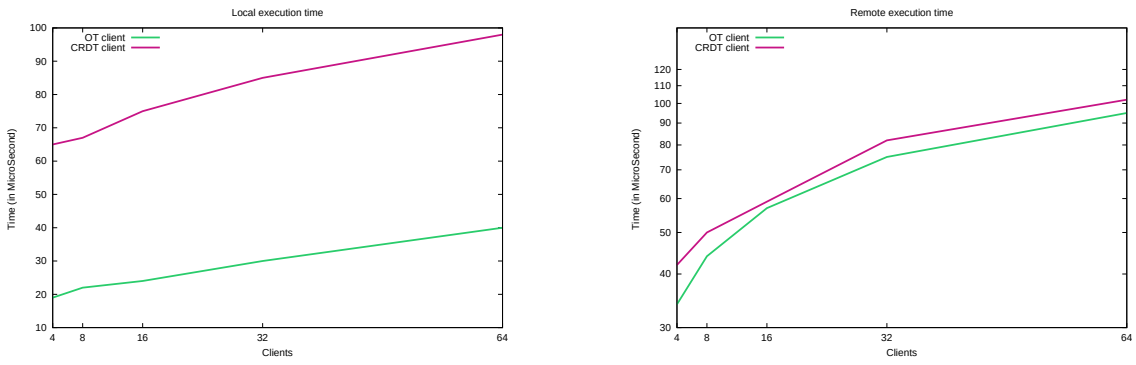
Pour évaluer les algorithmes sur une architecture réelle, nous avons déployé l'architecture proposée sur des machines sur Amazon Web Services (AWS). Nous avons loué deux serveurs et 64 clients dispersés dans différentes régions. Les serveurs en Irlande et en Californie. Alors que les clients sont dispersés équitablement dans 4 régions : Tokyo, Virginia, Singapore et Sydney.

Pour présenter une architecture réelle, nous avons choisi une machine puissante pour les serveurs dans les centres de données par rapport aux clients. Nous avons exécuté les serveurs sur des machines dites *moyennes*[31], tandis que les clients sont sur des *micros*[32] machine.

De plus, pour minimiser les perturbations extérieures telle que la latence du réseau, l'ensemble des expérimentations ont été conduites dans la même période et avec les mêmes caractéristiques de simulation.

Nous avons simulé une session d'édition collaborative suivant des caractéristiques réelles présentées dans la section 4.1.1. Nous avons mené des expériences différentes selon le nombre de clients et le nombre d'opérations. On a mesuré le temps d'exécution moyen que prend chaque client pour générer et intégrer une opération. On a calculé également, l'espace mémoire occupée par chaque réplique. Afin d'observer l'intérêt de notre architecture, l'expérience a été faite sur deux architectures différentes :

1. OT/CRDT : Les clients utilisent un algorithme OT, et le serveur utilise l'algorithme OT pour la communication avec les clients et un algorithme CRDT pour la communication avec les centres de données,
2. CRDT/CRDT : Les clients utilisent un algorithme CRDT, et le serveur devient une passerelle CRDT. C'est à dire que les clients envoient des opérations CRDT au serveur, et ce dernier l'envoie directement aux autres centres de donnée.



(a) Temps d'exécution local

(b) Temps d'intégration

FIGURE 6.4 – Temps d'exécution en fonction de nombre de répliques

L'algorithme OT que nous avons évalué est SOCT4 [113].

31. 2 CPU Intel (R) Xeon, 2.5GHz, Mémoire 4Gio
32. 1 CPU Intel (R) Xeon, 2.5GHz, Mémoire 1Gio

Même si SOCT4 peut supporter les opérations de bloc (copie/coller), si le client insère ou supprime un bloc de caractères, cette opération est transformée en une séquence d'opérations (caractère par caractère) pour simplifier l'expérimentation et le couplage OT/CRDT.

L'algorithme CRDT que nous avons évalué est Logoot. Cependant, l'algorithme original nécessite la préservation de la causalité, ce qui peut être coûteux en performance en utilisant les vecteurs d'horloges. La solution appropriée est d'utiliser le *degré* de visibilité [120] pour chaque identifiant Logoot. Contrairement aux autres approches, cette solution ne nécessite pas la préservation de la causalité pour assurer la cohérence du document.

Lorsqu'un caractère est inséré pour la première fois, il a un degré de visibilité de 1. Chaque fois le caractère est supprimé, son degré de visibilité est décrémenté. Si le degré de visibilité est inférieur ou supérieur à 0, l'élément est inséré dans un cimetière [33]. Les caractères sont visibles et apparaissent dans le document seulement si leur degré de visibilité est 1. Sinon, le caractère est invisible et ne figure pas dans le document.

Temps d'exécution Les figures 6.4a et 6.4b présentent respectivement le temps d'exécution moyen des algorithmes en fonction de nombre de clients, sur les architectures OT/CRDT (client OT) et CRDT/CRDT (client CRDT). Le temps moyen de génération est le temps moyen nécessaire pour générer une opération sur les dispositifs clients. Le temps d'intégration moyen est le temps moyen nécessaire pour intégrer une opération distante sur toutes les répliques. Dans chaque expérience, nous multiplions le nombre de clients par deux. Chaque client génère 10 000 opérations.

L'algorithme SOCT4 est plus performant que Logoot en temps d'exécution locale (figure 6.4a). En effet, SOCT4 ne dépasse pas $40ms$ avec 64 utilisateurs (donc 64 000 opérations), alors que Logoot dépasse $100ms$. L'algorithme Logoot a besoin de générer un identifiant unique pour chaque opération. Lorsque le nombre d'opérations augmente, le nombre d'insertions au milieu du texte augmente, et par conséquence la taille des identifiants augmente. Alors que SOCT4 est un algorithme OT, il exécute l'opération directement et ne nécessite ni identifiant ni transformation. En conséquence, SOCT4 est beaucoup plus performant et efficace que Logoot en temps d'exécution locale. Cette différence on l'attendait et c'est pour cette raison nous avons proposé l'algorithme SOCT4 sur les dispositifs clients.

En temps d'intégration, les deux algorithmes perdent en performance. Cette dégradation est due au nombre d'opérations concurrentes. Ces dernières sont proportionnelles aux nombres de répliques. L'algorithme SOCT4 à besoin de parcourir l'historique des opérations concurrentes et de transformer l'opération distante avec celle ci. Puisque le nombre d'opérations concurrentes augmente, le nombre de transformation augmente aussi. Pour cette raison, le temps d'intégration d'une opération distante dans le document local augmente.

Logoot a besoin de parcourir le document avec une complexité de $(k.log(n))$ pour trouver la position correcte de l'identifiant où n est la taille du document et k est la taille de l'identifiant. Le paramètre n est proportionnel aux nombres d'opérations et k est influencé par le nombre d'opérations concurrentes. Par conséquent, Logoot prend plus du temps pour trouver la bonne position.

De manière bien plus surprenante, l'algorithme Logoot est moins performant en temps d'intégration que SOCT4. Même si ce dernier nécessite des transformations avec les opérations de l'historique, la recherche de l'identifiant par Logoot prend plus de temps. L'algorithme SOCT4 purge les opérations exécutées de l'historique. Ainsi, la taille de l'historique reste faible.

33. Comme les pierres tombales, mais pas besoin d'être inclus dans le document

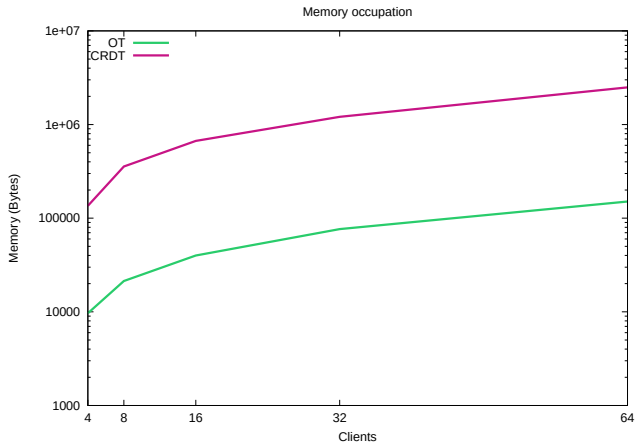


FIGURE 6.5 – Espace mémoire

Occupation mémoire La mémoire occupée(figure 6.5) par Logoot connaît une croissance rapide qui peut être problématique, particulièrement sur les dispositifs avec des ressources limitées telle que les mobiles et les tablettes. En effet, Logoot stocke pour chaque caractère un identifiant. De plus, la taille de l'identifiant Logoot est relative aux nombres d'opérations. Pour cette raison, plus le nombre de clients devient important et plus la mémoire occupée devient grande. L'algorithme SOCT4 sauvegarde dans son buffer uniquement ses propres opérations, le temps qu'elles soient émises vers le serveur SOCT4. Une fois émises, elles sont supprimées de son historique. De ce fait, SOCT4 occupe moins de mémoire que Logoot.

6.3 Conclusion

Dans ce chapitre, nous avons proposé une architecture pour l'édition collaborative sur les nuages. Contrairement aux architectures actuelles, l'architecture proposée est simple, peu coûteuse, limite les ressources sur les dispositifs clients et ne nécessite pas de consensus entre les centres de données.

Nous avons évalué les performances des algorithmes OT et CRDT et étudié leur pertinence sur cette architecture. Nous avons simulé sur cette architecture une édition collaborative. Les clients ont été exécutés sur des machines réelles et les traces d'exécution sont inspirées d'une édition collaborative réelle. Nous avons calculé le temps d'exécution local, le temps d'intégration des opérations distante et l'occupation mémoire pour chaque client. Nous avons confirmé que les algorithmes OT sont plus légers sur les dispositifs clients et de manière surprenante SOCT4 surpasse l'algorithme Logoot en temps d'intégration. Nous avons constaté que les applications d'édition collaborative peuvent être améliorées par la fusion des approches OT et CRDT. Contrairement aux systèmes d'éditions OT centralisés, notre solution assure la cohérence inéluctable, la disponibilité et la tolérance aux pannes.

Chapitre 7

Conclusion et perspectives

Sommaire

7.1 Conclusion

La réplication optimiste est largement utilisée dans les systèmes de partage de données. Elle offre une disponibilité permanente des données et réduit la latence réseau par l'accès local à l'objet partagé. Face aux modifications concurrentes, l'état de l'objet peut être différent entre les différentes répliques. Les systèmes de réplication optimiste intègrent des algorithmes spécifiques pour garantir la consistance de l'objet partagé. Selon le type de données répliqué, plusieurs algorithmes de réplication optimiste ont été proposés. Cependant, appliquer ces algorithmes sur un système réel nécessite une étude approfondie préalable. Une étude théorique n'est pas suffisante, car la complexité des algorithmes est en fonction des paramètres différents. De plus, pour connaître quel algorithme est le mieux adapté pour quelle situation, il faut les confronter à des situations réelles.

Les travaux précédents [62, 120, 85] ont évalué les algorithmes sur des traces soit générées aléatoirement soit sérialisées sans concurrence telles que les traces de Wikipedia et SVN. Jusqu'à présent aucune étude sur les performances n'a été effectuée sur des traces réelles incluant la concurrence. À cet égard, nous avons proposé dans cette thèse :

1. Une méthodologie d'évaluation pour les différents types de données répliqués. Cette méthodologie garantit la reproductibilité des expérimentations, contrôle les paramètres d'évaluation et représente une abstraction de la réalité. Nous avons mis en oeuvre cette méthodologie sur un outil d'évaluation. Ce dernier intègre : *a*) plusieurs algorithmes de réplication optimiste pour les différents types de données répliqués, *b*) un mécanisme de génération de traces qui propose le choix entre des traces synchrones ou asynchrones, *c*) un simulateur d'édition collaborative qui reproduit des sessions d'édition collaborative sur les traces générées en utilisant les algorithmes de réplication.

   Nous avons ensuite mis en oeuvre cette méthodologie sur un outil d'évaluation. Cet outil est open-source et générique. Les chercheurs peuvent intégrer, évaluer et comparer leurs solutions avec les approches existantes.

2. Nous avons ensuite réalisé des expérimentations pour évaluer les performances des algorithmes de réplication suivant la méthodologie proposée. Cette étude avait pour but de savoir quel est l'algorithme le plus adapté pour quelle application. Pour les algorithmes de réplication textuels par exemple, nous avons trouvé que Treedoc est un algorithme générique souhaitable pour n'importe quelle application, car il obtient de bonnes performances dans la quasi-totalité des expériences. Cependant, dans une collaboration avec un taux élevé de copier/coller, LogootSplit est le mieux adapté. Dans une session d'édition collaborative massive où seulement quelques participants éditent, RGA et WOOTH sont les plus appropriés car ils sont les plus performants en temps d'intégration. Dans le cas d'utilisation des dispositifs avec une mémoire de stockage limitée tels que les appareils mobiles, l'utilisation de LogootSplit est la plus appropriée, car il consomme peu de mémoire et génère moins de flux dans le réseau. D'autre part, les algorithmes destinés à l'édition collaborative des documents structurés dépendent de la politique utilisée pour la gestion des conflits. Cependant, les algorithmes OT décentralisés restent les moins performants par rapport aux algorithmes CRDT.
3. Nous avons ensuite proposé une métrique pour calculer l'effort effectué par les utilisateurs en cas de conflit. Pour calculer cet effort, nous avons reproduit des sessions d'édition collaborative à partir de l'historique de quelques répertoires Git en utilisant les algorithmes de réplication. Nous avons trouvé que les cas les plus fréquents qui créent les conflits sont les mises à jour concurrentes produites dans la même zone de texte, l'accidental clean merge (acm) et le undo/redo. Nous avons ensuite essayé de comprendre quel est l'impact de chaque cas de collaboration sur la qualité du résultat. Nous avons trouvé par exemple que l'impact des accidental clean merge est beaucoup plus important que undo/redo.
4. Après avoir analysé les performances et mesuré l'effort des développeurs, nous avons proposé des solutions pour améliorer les performances des algorithmes et la qualité du résultat. Pour les approches dédiées au document linéaire, nous avons proposé WOOTH. Ce dernier est basé sur WOOT et est inspirée par l'approche RGA. Pour améliorer la qualité du résultat, nous avons adapté un algorithme OT pour les cas les cas les plus fréquents qui créent les conflits. Nous avons réussi à réduire jusqu'à 50% le nombre de conflits.
5. Dans le dernier chapitre, nous avons proposé une nouvelle architecture dédiée aux applications d'édition collaborative sur les clouds. L'idée principale est de combiner les deux approches OT et CRDT afin d'éviter le consensus inter centre de données, tirer profit des deux approches et satisfaire l'utilisateur en respectant les contraintes des dispositifs clients. Nous avons déployé un algorithme centralisé OT entre les clients et le centre de données. La communication inter centre de données est assurée de manière pair-à-pair à l'aide du mécanisme CRDT. Nous avons confirmé que les algorithmes OT sont plus légers et souhaitable sur les dispositifs mobiles que les algorithmes CRDT. Ils sont rapides en temps d'exécution locale, occupent peu de mémoire et génèrent moins de flux dans le réseau. À notre grande surprise, nous avons constaté que l'algorithme OT (SOCT4) surpasse un algorithme CRDT (Logoot) dans toutes nos mesures y compris en temps d'intégration.

7.2 Perspectives

Dans cette thèse, nous avons proposé une méthodologie d'évaluation pour les types de données répliqués. Le cadre de notre étude était l'édition collaborative. Nous avons proposé un outil d'évaluation qui intègre un mécanisme d'extraction de traces et un simulateur pour rejouer une édition collaborative. Le simulateur demande aux algorithmes de réplication de transformer les

traces en une suite d'opérations. Plusieurs points s'avèrent intéressants à explorer et à développer :

1. Étendre notre étude pour les systèmes de fichier distribués : les types de données *arbre* sont conçus pour gérer des traces structurées. Ces traces sont disponibles sur les DVCS. Analyser les performances des algorithmes de réplication de type arbre sur des traces de système de fichier est plus réaliste,
2. État au lieu d'opérations : l'évaluation que nous avons mené jusqu'à présent traite les modifications comme une suite d'opérations. Cependant, certains algorithmes de réplication sont conçus pour la gestion des états et non pas d'opérations. Pour mesurer les performances de ces algorithmes, il est important d'intégrer dans l'outil d'évaluation un mécanisme de gestion d'états,
3. Mode de diffusion : le simulateur intégré dans l'outil d'évaluation diffuse les opérations en mode broadcast. Ce type de diffusion augmente le flux dans le réseau. Les éditeurs collaboratifs synchrones propagent les opérations selon des protocoles spécifiques tel que l'anti-entropie. Alors que le système Git synchronise les répliques deux à deux. Nous envisageons d'intégrer le type de diffusion correspondant pour chaque type de collaboration,
4. Analyse automatique : on souhaite intégrer dans l'outil d'évaluation un mécanisme qui analyse le résultat automatiquement. Par exemple, pour faciliter la compréhension du comportement des utilisateurs durant une collaboration asynchrone, il est plus préférable de détecter automatiquement les scénarii de collaboration qui créent les conflits.

D'autre part, l'évaluation des algorithmes de réplication sur l'architecture proposée nous a démontré qu'il était possible de déployer une application d'édition collaborative sur les nuages, sans utilisation de consensus inter centre de données, tout en respectant les contraintes des dispositifs des clients. Cependant, il est important de comparer cette architecture avec des architectures actuellement utilisées en terme de temps de convergence.

Bibliographie

[1] Google drive. https ://drive.google.com.

[2] Redis, [a key-value store. http ://www.redis.io/.

[3] Riak key-value store. http ://basho.com/riak/.

[4] Tuning garbage collection with the 5.0 java[tm] virtual machine. http ://www.oracle.com/technetwork/java/gc-tuning-5-138395.html.

[5] Microsoft skydrive, 2013. https ://skydrive.live.com/.

[6] I. .-.-. R. 2014-06-23. Eclipse community survey 2014 results.

[7] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso. Evaluating crdts for real-time document editing. In ACM, editor, *ACM Symposium on Document Engineering*, page 10 pages, San Francisco, CA, USA, september 2011.

[8] M. Ahmed-Nacer, P. Urso, and F. Charoy. Improving textual merge result. In *Collaborative Computing : Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*, pages 390–399, 2013.

[9] M. Ahmed-Nacer, P. Urso, F. Charoy, et al. Evaluating software merge quality. In *18th International Conference on Evaluation and Assessment in Software Engineering*, 2014.

[10] S. Alshattnawi. *Concurrence et Conscience de Groupe dans l'Édition Collaborative sur Réseaux Pair-à-Pair.* PhD thesis, Université Henri Poincaré-Nancy I, 2008.

[11] L. André, S. Martin, G. Oster, and C.-L. Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *CollaborateCom*, pages 50–59, 2013.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987. ISBN 0-201-10715-5.

[13] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv :1210.3368*, 2012.

[14] A. Bouazza, P. Molli, et al. Unifying coupled and uncoupled collaborative work in virtual teams. In *ACM CSCW'2000 workshop on collaborative editing systems, Philadelphia, Pennsylvania, USA*. Citeseer, 2000.

[15] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 168–178, New York, NY, USA, 2011. ACM.

[16] J. Buffenbarger. Syntactic software merging. In *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, pages 153–172, London, UK, UK, 1995. Springer-Verlag.

[17] S. Burckhardt, M. Fahndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*. Springer, June 2012.

[18] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350. USENIX Association, 2006.

[19] Canonical and community. What is Bazaar ?, (December 2007). http ://bazaar.canonical.com/en/.

[20] H. Casanova, A. Legrand, and M. Quinson. Simgrid : A generic framework for large-scale distributed experiments. In *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*, pages 126–131, April 2008.

[21] P. Cederqvist. *Version Management with CVS*. Network Theory Ltd., December 2002.

[22] R. Choudhary and P. Dewan. A general multi-user undo/redo model. In *ECSCW'95 : Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, pages 231–246, Norwell, MA, USA, 1995. Kluwer Academic Publishers.

[23] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab : An overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, pages 3–12, 2003.

[24] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion*. O'Reilly Media, 2007.

[25] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 1 :1–1 :14, New York, NY, USA, 2012. ACM.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner : Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3) :8, 2013.

[27] R. de Cepêda, A. Magdaleno, L. Murta, and C. Werner. Evoltrack : improving design evolution awareness in software development. *Journal of the Brazilian Computer Society*, 16(2) :117–131, 2010.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo : amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6) :205–220, 2007.

[29] A. Deftu and J. Griebsch. A scalable conflict-free replicated set data type. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 186–195, Washington, DC, USA, 2013. IEEE Computer Society.

[30] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenkcr, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixteenth annual ACM Symposium on Principles of Distributed Computing - PODC'87*, pages 1–12. ACM Press, 1987.

[31] P. Dewan and R. Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *ECSCW*, pages 159–178, 2007.

[32] H. Du and D. J. S. Hilaire. Multi-paxos : An implementation and evaluation.

[33] D. M. P. Eggert and R. Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd, January 2003.

[34] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *SIGMOD Conference*, pages 399–407. ACM Press, 1989.

[35] C. A. Ellis, S. J. Gibbs, and G. Rein. Groupware : Some issues and experiences. *Commun. ACM*, 34(1) :39–58, January 1991.

[36] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulieacute ;. Epidemic information dissemination in distributed systems. *Computer*, 37(5) :60–67, May 2004.

[37] T. E. Foundation. Etherpad.

[38] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33 :51–59, June 2002.

[39] S. Gilbert and N. A. Lynch. Perspectives on the cap theorem. Institute of Electrical and Electronics Engineers, 2012.

[40] git-mergetool. manual page. `http://git-scm.com/docs/git-merge`.

[41] F. S. F. GNU. Diff3. Three way file comparison program, (Septembre 2005).

[42] Google Help. google docs, sheets, and slides size limits. `https://support.google.com/drive/answer/37603?hl=en`.

[43] J. Gustedt, E. Jeannot, and M. Quinson. Experimental Validation in Large-Scale Systems : a Survey of Methodologies. *Parallel Processing Letters*, page 16, 2009. RR-6859 RR-6859.

[44] R. G. Guy, J. S. Heidemann, and T. W. Page, Jr. The ficus replicated file system. *SIGOPS Oper. Syst. Rev.*, 26(2) :26–, April 1992.

[45] C.-L. Ignat and M. C. Norrie. Customizable collaborative editor relying on treeopt algorithm. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, ECSCW'03, pages 315–334, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[46] C.-L. Ignat, S. Papadopoulou, G. Oster, and M. C. Norrie. Providing awareness in multi-synchronous collaboration without compromising privacy. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, pages 659–668. ACM, 2008.

[47] A. Imine. *Conception formelle d'algorithmes de réplication optimiste. Vers l'édition Collaborative dans les réseaux Pair-a-Pair.* PhD thesis, PhD thesis, Université Henri Poincaré, Nancy, 2006.

[48] T. Issariyakul and E. Hossain. *Introduction to Network Simulator NS2.* Springer Publishing Company, Incorporated, 1 edition, 2008.

[49] D. Jackson and D. A. Ladd. Semantic diff : A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[50] R. Jain. The art of computer system performance analysis : techniques for experimental design, measurement, simulation and modeling. *New York : John Willey*, 1991.

[51] C. Jay, M. Glencross, and R. Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Transactions on Computer-Human Interaction*, 14(2), August 2007.

[52] P. R. Johnson and R. H. Thomas. RFC 677 : Maintenance of duplicate databases, January 1975, (Septembre 2005). `http://www.ietf.org/rfc/rfc677.txt`.

[53] B. K. Kasi and A. Sarma. Cassandra : Proactive conflict minimization through optimized task scheduling. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 732–741, Piscataway, NJ, USA, 2013. IEEE Press.

[54] S. Khanna, K. Kunal, and B. C. Pierce. A formal investigation of diff3. In *FSTTCS 2007 : Foundations of Software Technology and Theoretical Computer Science*, pages 485–496. Springer, 2007.

[55] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10(1) :3–25, February 1992.

[56] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. In *Proceedings of The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, Big Sky, MT, USA, October 2009. SIGOPS.

[57] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, 1978.

[58] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9) :690–691, 1979.

[59] L. Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2002.

[60] S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000 : a large scale and highly reconfigurable grid experimental testbed, 2006.

[61] D. Li and R. Li. Preserving operation effects relation in group editors. In *Proceedings of the ACM conference on Computer Supported Cooperative Work - CSCW '04*, pages 457–466. ACM Press, 2004.

[62] D. Li and R. Li. A performance study of group editing algorithms. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, page 8 pp., 0-0 2006.

[63] Google Docs. limits on sharing. `https://support.google.com/drive/answer/2494827?hl=en`.

[64] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, Californie, États-Unis, 1996.

[65] M. Mackall. Work easier Work faster, (April 2005). http ://mercurial.selenic.com/.

[66] J. Maeda. *The laws of simplicity*. MIT Press, 2006.

[67] S. Martin. *Edition collaborative des documents semi-structurés*. PhD thesis, Université de Provence, 2011.

[68] S. Martin, M. Ahmed-Nacer, and P. Urso. Controlled conflict resolution for replicated document. In *CollaborateCom*, pages 471–480, 2012.

[69] S. Martin and D. Lugiez. Collaborative peer to peer edition : Avoiding conflicts is better than solving conflicts. In H. Weghorn and P. T. Isaías, editors, *IADIS AC (2)*, pages 124–128. IADIS Press, 2009.

[70] S. Martin, P. Urso, and S. Weiss. Scalable xml collaborative editing with undo. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems : OTM 2010*, volume 6426 of *Lecture Notes in Computer Science*, pages 507–514. Springer, 2010.

[71] M. C. MAZILU. Database replication. *Database Systems Journal*, 1(2) :33–38, 2010.

[72] A.-N. Mehdi, PascalUrso, and F. Charoy. Merging by decentralized eventual consistency algorithms. *EAI collaborative computing.*, 2015.

[73] A.-N. Mehdi, P. Urso, V. Balegas, and N. Perguiça. Merging ot and crdt algorithms. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 9 :1–9 :4, New York, NY, USA, 2014. ACM.

[74] T. Mens. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on*, 28(5) :449–462, 2002.

[75] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proceedings of the ACM SIGGROUP Conference on Supporting Group Work - GROUP 2003*, pages 212–220, Sanibel Island, Florida, USA, November 2003. ACM Press.

[76] A. Montresor and M. Jelasity. PeerSim : A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

[77] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2) :251–266, 1986.

[78] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, UIST '95, pages 111–120, New York, NY, USA, 1995. ACM.

[79] B. M. Oki and B. H. Liskov. Viewstamped replication : A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[80] H. Oliveira, L. Murta, and C. Werner. Odyssey-vcs : A flexible version control system for uml model elements. In *Proceedings of the 12th International Workshop on Software Configuration Management*, SCM '05, pages 1–16, New York, NY, USA, 2005. ACM.

[81] G. Oster, P. Urso, and P. Molli. Proving correctness of transformation functions in collaborative editing systems. Rapport de Recherche 5795, INRIA, December 2005.

[82] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 259–267, Banff, Alberta, Canada, nov 2006. ACM Press.

[83] G. Oster, P. Urso, P. Molli, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *The Second International Conference on Collaborative Computing : Networking, Applications and Worksharing (CollaborateCom 2006)*, Atlanta, Georgia, USA, November 2006. IEEE Press.

[84] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development : an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3) :308–337, July 2001.

[85] N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *ICDCS*, pages 395–403. IEEE Computer Society, 2009.

[86] J. G. Prudencio, L. Murta, C. Werner, and R. Cepeda. To lock, or not to lock : That is the question. *Journal of Systems and Software*, 85(2) :277 – 289, 2012. Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering.

[87] M. Raynal and M. Singhal. Logical time : Capturing causality in distributed systems. *Computer*, 29(2) :49–56, 1996.

[88] M. Ressel and R. Gunzenhäuser. Reducing the problems of group undo. In *GROUP '99 : Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 131–139, New York, NY, USA, 1999. ACM.

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

[90] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types : Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3) :354 – 368, 2011.

[91] S. Ross, L. Fang, and K. W. Hipel. A case-based reasoning system for conflict resolution : design and implementation. *Engineering Applications of Artificial Intelligence*, 15(3) :369–383, 2002.

[92] D. Roundy. Darcs : distributed version management in haskell. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, pages 1–4, New York, NY, USA, 2005. ACM.

[93] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1) :42–81, 2005.

[94] A. Sarma, Z. Noroozi, and A. Van der Hoek. Palantir : raising awareness among configuration management workspaces. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 444–454, 2003.

[95] A. Sarma, D. Redmiles, and A. van der Hoek. Palantir : Early detection of development conflicts arising from parallel code changes. *Software Engineering, IEEE Transactions on*, 38(4) :889–908, 2012.

[96] T. Schümmer and J. M. Haake. Supporting distributed software development by modes of collaboration. In *Proceedings of the seventh conference on European Conference on Computer Supported Cooperative Work*, pages 79–98. Kluwer Academic Publishers, 2001.

[97] B. Shao, D. Li, and N. Gu. A Fast Operational Transformation Algorithm for Mobile and Asynchronous Collaboration. *IEEE Transactions on Parallel and Distributed Systems*, 21(12) :1707–1720, December 2010.

[98] M. Shapiro and N. Preguiça. Designing a commutative replicated data type. Rapport de recherche INRIA RR-6320, INRIA, October 2007.

[99] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, INRIA, January 2011.

[100] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976, pages 386–400, Grenoble, France, October 2011.

[101] B. Shneiderman. Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys*, 16(3) :265–285, September 1984.

[102] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work : the integration challenge*, GROUP '97, pages 435–445, New York, NY, USA, 1997. ACM.

[103] C. Sun. Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(4) :309–361, December 2002.

[104] C. Sun and C. A. Ellis. Operational transformation in real-time group editors : Issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW'98*, pages 59–68, New York, New York, États-Unis, November 1998. ACM Press.

[105] 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, March 1998.

[106] A. Tanenbaum and M. Van Steen. *Distributed systems*. Pearson Prentice Hall, 2007.

[107] TeamEdit. a collaborative text editor. `http://teamedit.sourceforge.net/`.

[108] D. Terry. Replicated data consistency explained through baseball. Technical Report MSR-TR-2011-137, October 2011.

[109] W. F. Tichy. Should computer scientists experiment more ? *Computer*, 31(5) :32–40, 1998.

[110] L. Torvalds. git, (April 2005). http ://git-scm.com/.

[111] M. Van Steen. Distributed systems principles and paradigms. *Network*, 2 :28, 2002.

[112] J. Vesperman. *Essential CVS - version control and source-code management (2. ed.)*. O'Reilly, 2007.

[113] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, CSCW '00, pages 171–180, New York, NY, USA, 2000. ACM.

[114] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1) :40–44, January 2009.

[115] S. Weiss, P. Urso, and P. Molli. Wooki : a P2P Wiki-based Collaborative Writing Tool. In *Web Information Systems Engineering*, pages 503–512, Nancy, France, December 2007. Springer.

[116] S. Weiss, P. Urso, and P. Molli. Wooki : a P2P Wiki-based Collaborative Writing Tool (long version) . Rapport de recherche INRIA RR-6226, INRIA, May 2007.

[117] S. Weiss, P. Urso, and P. Molli. An Undo Framework for P2P Collaborative Editing . In *CollaborateCom*, pages 529–544, Orlando, USA, November 2008.

[118] S. Weiss, P. Urso, and P. Molli. Logoot : a P2P collaborative editing system. Rapport de recherche INRIA RR-6713, INRIA, December 2008.

[119] S. Weiss, P. Urso, and P. Molli. Logoot : A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, pages 404 –412, Montréal, Québec, Canada, jun. 2009. IEEE Computer Society.

[120] S. Weiss, P. Urso, and P. Molli. Logoot-undo : Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel and Distributed Systems*, 21 :1162–1174, 2010.

Résumé

Pour fournir une disponibilité permanente des données et réduire la latence réseau, les systèmes de partage de données se basent sur la réplication optimiste. Dans ce paradigme, il existe plusieurs copies de l'objet partagé dite répliques stockées sur des sites. Ces répliques peuvent être modifiées librement et à tout moment. Les modifications sont exécutées en local puis propagées aux autres sites pour y être appliquées. Les algorithmes de réplication optimiste sont chargés de gérer les modifications parallèles. L'objectif de cette thèse est de proposer une méthodologie d'évaluation pour les algorithmes de réplication optimiste. Le contexte de notre étude est l'édition collaborative. Nous allons concevoir pour cela un outil d'évaluation qui intègre un mécanisme de génération de corpus et un simulateur d'édition collaborative. À travers cet outil, nous allons dérouler plusieurs expériences sur deux types de corpus : synchrone et asynchrone. Dans le cas d'une édition collaborative synchrone, nous évaluerons les performances des différents algorithmes de réplication sur différents critères tels que le temps d'exécution, l'occupation mémoire, la taille des messages, etc. Nous proposerons ensuite quelques améliorations. En plus, dans le cas d'une édition collaborative asynchrone, lorsque deux répliques se synchronisent, les conflits sont plus nombreux à apparaître. Le système peut bloquer la fusion des modifications jusqu'à ce que l'utilisateur résolut les conflits. Pour réduire le nombre de ces conflits et l'effort des utilisateurs, nous proposerons une métrique d'évaluation et nous évaluerons les différents algorithmes sur cette métrique. Nous analyserons le résultat pour comprendre le comportement des utilisateurs et nous proposerons ensuite des algorithmes pour résoudre les conflits les plus important et réduire ainsi l'effort des développeurs. Enfin, nous proposerons une nouvelle architecture hybride basée sur deux types d'algorithmes de réplication. Contrairement aux architectures actuelles, l'architecture proposée est simple, limite les ressources sur les dispositifs clients et ne nécessite pas de consensus entre les centres de données.

Mots-clés: Réplication optimiste, Type de données répliqués, Méthodologie d'évaluation.

Abstract

To provide a high availability from any where, at any time, with low latency, data is optimistically replicated. This model allows any replica to apply updates locally, while the operations are later sent to all the others. In this way, all replicas eventually apply all updates, possibly even in different order. Optimistic replication algorithms are responsible for managing the concurrent modifications and ensure the consistency of the shared object. In this thesis, we present an evaluation methodology for optimistic replication algorithms. The context of our study is collaborative editing. We designed a tool that implements our methodology. This tool integrates a mechanism to generate a corpus and a simulator to simulate sessions of collaborative editing. Through this tool, we made several experiments on two different corpus : synchronous and asynchronous. In synchronous collaboration, we evaluate the performance of optimistic replication algorithms following several criteria such as execution time, memory occupation, message's size, etc. After analysis, some improvements were proposed. In addition, in asynchronous collaboration, when replicas synchronize their modifications, more conflicts can appear in the document. In this case, the system cannot merge the modifications until a user resolves them. In order to reduce the conflicts and the user's effort, we propose an evaluation metric and we evaluate the different algorithms on this metric. Afterward, we analyze the quality of the merge to understand the behavior of the users and the collaboration cases that create conflicts. Then, we propose algorithms for resolving the most important conflicts, therefore reducing the user's effort. Finally, we propose a new architecture for supporting cloud-based collaborative editing system. This architecture is based on two optimistic replication algorithms. Unlike current architectures, the proposed one removes the problems of the centralization and consensus between data centers, is simple and accessible for any developers.

Keywords: Optimistic replication, Replicated data types, Evaluation methodology.