

Ré-identification sans coordination dans les types de données répliquées sans conflits (CRDTs)

THÈSE

présentée et soutenue publiquement le TODO : Définir une date

pour l'obtention du

Doctorat de l'Université de Lorraine
(mention informatique)

par

Matthieu Nicolas

Composition du jury

<i>Président :</i>	À déterminer	
<i>Rapporteurs :</i>	Hanifa Boucheneb	Professeure, Polytechnique Montréal
	Davide Frey	Chargé de recherche, HdR, Inria Rennes Bretagne-Atlantique
<i>Examineurs :</i>	Hala Skaf-Molli	Maîtresse de conférences, HdR, Nantes Université
	Stephan Merz	Directeur de Recherche, Inria Nancy - Grand Est
<i>Encadrants :</i>	Olivier Perrin	Professeur des Universités, Université de Lorraine, LORIA
	Gérald Oster	Maître de conférences, Université de Lorraine, LORIA

Mis en page avec la classe thesul.

Remerciements

WIP

WIP

Sommaire

Chapitre 1	
Introduction	1
1.1 Contexte	1
1.1.1 Réplication de données mutables	2
1.1.2 Réplication de données pessimiste	3
1.1.3 Réplication de données optimiste	3
1.2 Questions de recherche et contributions	4
1.2.1 Ré-identification sans coordination synchrone pour Conflict-free Re- plicated Data Types (CRDTs) pour le type Séquence	4
1.2.2 Éditeur de texte collaboratif Pair-à-Pair (P2P) temps réel chiffré de bout en bout	5
1.3 Plan du manuscrit	6
1.4 Publications	7
Chapitre 2	
État de l’art	11
2.1 Modèle du système	11
2.2 Types de données répliquées sans conflits	12
Chapitre 3	
Renommage dans une séquence répliquée	17
3.1 Présentation de l’approche	18
3.1.1 Définition de l’opération de renommage	18
3.2 Introduction de l’opération <i>rename</i>	19
3.2.1 Opération de renommage proposée	19
3.2.2 Gestion des opérations concurrentes au renommage	21
3.2.3 Évolution du modèle de livraison des opérations	23

3.3	Gestion des opérations <i>rename</i> concurrentes	25
3.3.1	Conflits en cas de renommages concurrents	25
3.3.2	Relation de priorité entre renommages	26
3.3.3	Algorithme d’annulation de l’opération de renommage	28
3.3.4	Processus d’intégration d’une opération	32
3.3.5	Règles de récupération de la mémoire des états précédents	37
3.4	Validation	39
3.4.1	Complexité en temps des opérations	39
3.4.2	Expérimentations	43
3.4.3	Résultats	44
3.5	Discussion	51
3.5.1	Stratégie de génération des opérations <i>rename</i>	51
3.5.2	Stockage des états précédents sur disque	52
3.5.3	Compression et limitation de la taille de l’opération <i>rename</i>	53
3.5.4	Définition de relations de priorité pour minimiser les traitements	53
3.5.5	Report de la transition vers la nouvelle époque cible	55
3.5.6	Utilisation de l’opération de renommage comme mécanisme de compression du log d’opérations	55
3.5.7	Implémentation alternative de l’intégration de l’opération <i>rename</i> basée sur le journal d’opérations	57
3.6	Comparaison avec les approches existantes	59
3.6.1	Core-Nebula	59
3.6.2	LSEQ	60
3.7	Conclusion	61

Chapitre 4

Conclusions et perspectives

63

4.1	Résumés des contributions	63
4.1.1	Ré-identification sans coordination pour les CRDTs pour Séquence	63
4.1.2	Éditeur de texte collaboratif P2P chiffré de bout en bout	65
4.2	Perspectives	67
4.2.1	Définition de relations de priorité pour minimiser les traitements	67
4.2.2	Détection et fusion manuelle de versions distantes	68
4.2.3	Étude comparative des différents modèles de synchronisation pour CRDTs	71

4.2.4	Approfondissement du patron de conception de Pure Operation- Based CRDTs	73
-------	---	----

Annexe A
Entrelacement d’insertions concurrentes dans Treedoc

Annexe B
Algorithmes <code>RENAMEID</code>

Annexe C
Algorithmes <code>REVERTRENAMEID</code>

Index	83
--------------	-----------

Bibliographie

Table des figures

2.1	Spécification algébrique du type abstrait usuel Ensemble	13
2.2	Modifications concurrentes d'un Ensemble répliqué provoquant un conflit suite à l'ajout et la suppression d'un même élément	14
3.1	Renommage de la séquence sur le noeud <i>A</i>	20
3.2	Modifications concurrentes menant à une anomalie	21
3.3	Renommage de la modification concurrente avant son intégration en utilisant <code>RENAMEID</code> afin de maintenir l'ordre souhaité	23
3.4	Livraison d'une opération <i>insert</i> sans avoir reçu l'opération <i>rename</i> précédente	23
3.5	Livraison désordonnée d'une opération <i>rename</i> et de l'opération <i>insert</i> qui la précède	25
3.6	Opérations <i>rename</i> concurrentes menant à des états divergents	25
3.7	<i>Arbre des époques</i> correspondant au scénario décrit dans la Figure 3.6 . . .	26
3.8	Sélectionner l'époque cible d'une exécution d'opérations <i>rename</i> concurrentes	27
3.9	Annulation d'une opération <i>rename</i> intégrée précédemment en présence d'un identifiant inséré en concurrence	28
3.10	Annulation d'une opération <i>rename</i> intégrée précédemment en présence d'un identifiant inséré causalement après	31
3.11	Intégration d'une opération <i>insert</i> distante	34
3.12	Intégration d'une opération <i>rename</i> distante	36
3.13	Suppression des époques obsolètes et récupération de la mémoire des <i>anciens états</i> associés	38
3.14	Évolution de la taille du document en fonction du CRDT utilisé et du nombre de <i>renaming bots</i> dans la collaboration	45
3.15	Temps d'intégration des opérations standards	47
3.16	Progression du nombre d'opérations du log rejouées en fonction du temps .	50
3.17	Livraison d'une opération <i>rename</i> d'un noeud	54
A.1	Modifications concurrentes d'une séquence Treedoc résultant en un entrelacement	77

Chapitre 1

Introduction

Sommaire

1.1	Contexte	1
1.1.1	Réplication de données mutables	2
1.1.2	Réplication de données pessimiste	3
1.1.3	Réplication de données optimiste	3
1.2	Questions de recherche et contributions	4
1.2.1	Ré-identification sans coordination synchrone pour CRDTs pour le type Séquence	4
1.2.2	Éditeur de texte collaboratif P2P temps réel chiffré de bout en bout	5
1.3	Plan du manuscrit	6
1.4	Publications	7

1.1 Contexte

- Systèmes collaboratifs (wikis, plateformes de contenu, réseaux sociaux) et leurs bienfaits (qualité de l'info, vitesse de l'info (exemple de crise?), diffusion de la parole). Démocratisation (sic) de ces systèmes au cours de la dernière décennie.
- En raison du volume de données et de requêtes, adoptent architecture décentralisée. Permet ainsi de garantir disponibilité, tolérance aux pannes et capacité de passage à l'échelle.
- Mais échoue à adresser problèmes non-techniques : confidentialité, souveraineté, protection contre censure, dépendance et nécessité de confiance envers autorité centrale.
- À l'heure où les entreprises derrière ces systèmes font preuve d'ingérence et d'intérêts contraires à ceux de leurs utilisateur-rices (Cambridge Analytica, Prism, non-modération/mise en avant de contenus racistes^{1 2 3}, invisibilisation de contenus fé-

1. *Algorithms of Oppression*, Safiya Umoja Noble
2. https://www.researchgate.net/publication/342113147_The_YouTube_Algorithm_and_the_Alt-Right_Filter_Bubble
3. <https://www.wsj.com/articles/the-facebook-files-11631713039>

ministres, dissolution du comité d'éthique de Google⁴, inégalité d'accès à la méta-machine affectante^{5 6 7}), paraît fondamental de proposer les moyens technologiques accessibles pour concevoir et déployer des alternatives.

- *Matthieu: TODO : Voir si angle écologique/réduction consommation d'énergie peut être pertinent.*
- Systèmes pair-à-pair sont une direction intéressante pour répondre à ces problématiques, de part leur absence d'autorités centrales, la distribution des tâches et leur conception mettant le pair au centre et les permettant de fonctionner aussi bien en mode connecté ou déconnecté. Mais posent de nouvelles problématiques de recherche.
- Ces systèmes ne disposent d'aucun contrôle sur les noeuds qui les composent. Le nombre de noeuds peut donc croître de manière non-bornée et atteindre des centaines de milliers de noeuds. La complexité des algorithmes de ces systèmes ne doit donc pas dépendre de ce paramètre, ou alors de manière logarithmique.
- De plus, ces noeuds n'offrent aucune garantie sur leur stabilité. Ils peuvent donc rejoindre et participer au système de manière éphémère. S'agit du phénomène connu sous le nom de churn. Les algorithmes de ces systèmes ne peuvent donc pas reposer sur des mécanismes nécessitant une coordination synchrone d'une proportion des noeuds.
- Finalement, ces noeuds n'offrent aucune garanties sur leur fiabilité et intentions. Les noeuds peuvent se comporter de manière byzantine. Pour assurer la confidentialité, l'absence de confiance requise et le bon fonctionnement du système, ce dernier doit être conçu pour résister aux comportements byzantins de ses acteurs.
- Ainsi, il est nécessaire de faire progresser les technologies existantes pour les rendre compatible avec ce nouveau modèle de système. Dans le cadre de cette thèse, nous nous intéressons aux mécanismes de réplication de données mutables dans les systèmes collaboratifs P2P sans autorités centrales.

1.1.1 Réplication de données mutables

Les techniques de réplication de données mutables introduisent de la redondance de données dans les systèmes. Cette redondance a pour but et effet d'améliorer plusieurs propriétés des systèmes :

Définition 1 (Disponibilité).

Définition 2 (Tolérance aux pannes).

Définition 3 (Capacité de passage à l'échelle).

Définition 4 (Latence).

4. <https://www.bbc.com/news/technology-56135817>

5. *Je suis une fille sans histoire*, Alice Zeniter, p. 75

6. Qui cite *Les affects de la politique*, Frédéric Lordon

7. <https://www.bbc.com/news/technology-59011271>

Les techniques de réplication de données peuvent être classées en deux approches : les *techniques de réplication de données pessimistes* et les *techniques de réplication de données optimistes*. Ces deux catégories offrent des compromis différents vis-à-vis des propriétés décrites par les théorèmes CAP [1] et PACELC [2]. Notamment, la différence entre ces catégories concerne le cas de la propriété de *cohérence*.

Définition 5 (Cohérence).

1.1.2 Réplication de données pessimiste

Les techniques de réplication de données dites *pessimistes* privilégient la cohérence des données. Notamment, ces techniques empêchent les modifications en concurrence d'une même donnée. Pour cela, plusieurs approches sont possibles.

- Première approche consiste à utiliser un verrou.
- Seconde approche consiste à utiliser un système de vote pour décider de la prochaine modification. Consensus, élection de leader, SMR.
- Le choix de privilégier la cohérence des données se fait au détriment de la disponibilité, tolérance aux pannes, capacité de passage à l'échelle et latence. Par exemple, un système basé sur un leader deviendra temporairement indisponible lors d'une panne de son leader, le temps que la panne soit détectée et qu'un nouveau leader soit élu.

1.1.3 Réplication de données optimiste

À l'inverse, les techniques de réplication de données dites *optimistes* jugent acceptable de relâcher les contraintes existantes sur la cohérence des données. Dans ce paradigme, chaque nœud qui possède une copie de la donnée répliquée peut la consulter et la modifier à tout moment, sans coordination préalable avec les autres nœuds. Les copies des nœuds sont donc autorisées à diverger de manière temporaire.

Les modifications effectuées par chacun sont ensuite diffusées pour être intégrées par l'ensemble des nœuds et converger de nouveau, c.-à-d. atteindre des états équivalents. Cependant, certaines modifications effectuées en concurrence par les nœuds peuvent provoquer des conflits. *Matthieu: NOTE : Pourrait insérer exemple de conflits de l'édition collaborative ici.* Des mécanismes de résolution de conflits, potentiellement automatiques, sont alors requis pour assurer la convergence à terme.

Cette approche permet donc de privilégier la disponibilité, tolérance aux pannes, capacité de passage à l'échelle et latence en échange de la cohérence forte.

Dans le cadre de cette thèse, nous nous intéressons aux techniques de réplication de données optimistes.

1.2 Questions de recherche et contributions

1.2.1 Ré-identification sans coordination synchrone pour CRDTs pour le type Séquence

- Les systèmes collaboratifs permettent à des utilisateur·rices de manipuler et d'éditer un contenu partagé. Ces systèmes peuvent adopter le paradigme de la réplication optimiste [3] pour favoriser leur disponibilité, latence et tolérance aux pannes. Ce paradigme autorise les noeuds possédant une copie de la donnée partagée de la consulter et de la modifier sans se coordonner préalablement avec les autres noeuds. Leur copies peuvent alors diverger momentanément. Un mécanisme de synchronisation permet ensuite à chaque noeud de récupérer l'ensemble des modifications et de les intégrer de façon à converger, c.-à-d. obtenir de nouveau des états équivalents. Cependant, le paradigme de la réplication optimiste permet la génération en concurrence de modifications provoquant un conflit, e.g. l'ajout et la suppression d'un même élément dans un Ensemble. Un mécanisme de résolution de conflits est alors nécessaire pour assurer la convergence à terme des noeuds [4].
- Les CRDTs [5, 6] sont des types de données répliqués. Ils sont conçus pour être répliqués par les noeuds d'un système et pour permettre à ces derniers de modifier les données partagées sans aucune coordination. Dans ce but, ils incluent des mécanismes de résolution de conflits automatiques directement au sein leur spécification. Ces mécanismes leur permettent de résoudre le problème évoqué précédemment. Cependant, ces mécanismes induisent un surcoût, aussi bien en termes de métadonnées, calculs et bande-passante. Notamment, certains CRDTs comme ceux pour le type Séquence souffrent d'une croissance monotone de leur surcoût. Ce surcoût s'avère handicapant dans le contexte des collaborations à large échelle.
- Dans le contexte des CRDTs pour le type Séquence, le surcoût du type de données répliquées provient de la croissance des métadonnées. Notamment, ces métadonnées correspondent à des identifiants associés aux éléments de la Séquence par les CRDTs. Ces identifiants sont ensuite utilisées par leur mécanisme de résolution de conflits automatique, e.g. pour référencer les éléments ou pour ordonner les éléments les uns relativement aux autres.
- Plusieurs approches ont été proposées pour réduire le coût induit par ces identifiants. Notamment, [7, 8] proposent un mécanisme de ré-assignation des identifiants pour réduire leur taille. Ce mécanisme provoque cependant des conflits avec les modifications concurrentes de la Séquence, c.-à-d. l'insertion ou la suppression. Pour résoudre ces nouveaux conflits, les auteurs proposent un mécanisme de transformation des modifications concurrentes par rapport au mécanisme de ré-assignation.
- Cependant, des exécutions concurrentes du mécanisme de ré-assignation des identifiants provoquent elles aussi des conflits. Pour éviter ces dernier, les auteurs choisissent de subordonner l'exécution du mécanisme de ré-assignation à un protocole de consensus. Ainsi, le mécanisme de ré-assignation ne peut être déclenché en concurrence par les noeuds du systèmes.

- Cependant, reposer sur un protocole de consensus est une contrainte forte dans un système distribué. Nous la jugeons même prohibitive pour les systèmes P2P à large échelle sujets au churn. Notre problématique de recherche est donc la suivante : *pouvons-nous proposer un mécanisme sans coordination synchrone de réduction du surcoût des CRDTs pour Séquence, c.-à-d. compatible avec les systèmes P2P à large échelle sujets au churn ?*
- Nous répondons à cette problématique en proposant RenamableLogootSplit, un nouveau CRDT pour le type Séquence. Ce CRDT intègre un mécanisme de renommage directement au sein de sa spécification, ainsi qu'un mécanisme de résolution de conflits automatique pour résoudre le conflit provoqué par des déclenchements concurrents de ce dernier. Ainsi, nous proposons un mécanisme de renommage permettant de réduire le surcoût du CRDT pour Séquence et qui est utilisable par les noeuds sans aucune coordination synchrone entre eux.

1.2.2 Éditeur de texte collaboratif P2P temps réel chiffré de bout en bout

- Les systèmes collaboratifs permettent à plusieurs utilisateur-rices de collaborer pour la réalisation d'une tâche. Les systèmes collaboratifs actuels adoptent principalement une architecture décentralisée, c.-à-d. un ensemble de serveurs avec lesquels les utilisateur-rices interagissent pour réaliser leur tâche, e.g. Google Docs [9]. Par rapport à une architecture centralisée, cette architecture leur permet d'améliorer leur disponibilité et tolérance aux pannes, notamment grâce aux méthodes de réplication de données. Cette architecture à base de serveurs facilite aussi la collaboration, les serveurs permettant d'intégrer les modifications effectuées par les utilisateur-rices, de stocker les données, d'assurer la communication entre les utilisateur-rices ou encore de les authentifier.
- De part le rôle qui leur incombe, ces serveurs jouent donc un rôle central dans ces systèmes. Il en découle plusieurs problématiques :
 - (i) Ces serveurs manipulent et hébergent les données faisant l'objet de collaborations. Ces systèmes ont donc connaissance des données manipulées et de l'identité des auteur-rices de ces modifications. Les systèmes collaboratifs décentralisés demandent donc à leurs utilisateur-rices d'abandonner la souveraineté et la confidentialité de leur travail.
 - (ii) Ces serveurs sont gérés par des autorités centrales, e.g. Google. Les systèmes collaboratifs devenant non-fonctionnels en cas d'arrêt des serveurs, les utilisateur-rices de ces systèmes dépendent alors de ces autorités centrales et de leurs intérêts. Par exemple, les autorités centrales ayant le pouvoir de vie et de mort sur leurs systèmes collaboratifs, elles menacent la pérennité de ces systèmes.
- Pour répondre à ces problématiques, c.-à-d. confidentialité et souveraineté des données, dépendance envers des tiers, pérennité des systèmes, un nouveau paradigme d'applications proposent de concevoir des Local-First Softwares (LFS), c.-à-d. des applications mettant l'utilisateur-riche et son appareil au coeur du système. Dans ce

cadre d'applications, les serveurs sont relégués à un rôle de support à la collaboration.

- Dans le cadre de cette mouvance, notre équipe de recherche étudie notamment la conception de systèmes P2P sans autorités centrales. Ce changement de modèle, d'une architecture décentralisée appartenant à des autorités centrales à une architecture P2P sans autorités centrales, introduit un ensemble de problématiques de domaines variés, e.g.
 - (i) Comment permettre aux utilisateur-rices de collaborer en l'absence d'autorités centrales pour résoudre les conflits de modifications ?
 - (ii) Comment authentifier les utilisateur-rices en l'absence d'autorités centrales ?
 - (iii) Comment structurer le réseau de manière efficace, c.-à-d. en limitant le nombre de connexion par pair ?
- Cet ensemble de questions peut être résumé en la problématique suivante : *pouvons-nous concevoir une application collaborative P2P à large échelle, sûre et sans autorités centrales ?*
- Pour étudier cette problématique, l'équipe Coast développe Multi User Text Editor (MUTE) [10]. Il s'agit d'un éditeur de texte web collaboratif P2P temps réel chiffré de bout en bout. Ce projet permet de présenter les travaux de recherche de l'équipe, c.-à-d. mécanismes de résolutions de conflits automatiques pour le type Séquence [11, 12, 13] et mécanisme d'authentification des pairs sans autorités centrales [14, 15]. Il permet aussi d'offrir un tour d'horizon des nombreux travaux de recherche nécessaires à la conception de tels systèmes, c.-à-d. mécanismes de conscience de groupe *Matthieu: TODO : Trouver et ajouter références*, protocoles d'appartenance aux groupes [16, 17], topologies réseaux P2P [18] et protocoles d'établissement de clés de chiffrement de groupe [19]. À notre connaissance, il s'agit du Proof of Concept (PoC) le plus complet d'applications LFS P2P sans autorités centrales. *Matthieu: TODO : Vérifier du côté des applis de IPFS*

1.3 Plan du manuscrit

- Ce manuscrit de thèse est organisé de la manière suivante :
- Dans le chapitre 2, nous introduisons le modèle du système que nous considérons, c.-à-d. les systèmes P2P à large échelle sujet au churn. Puis nous présentons dans ce chapitre l'état de l'art des mécanismes de résolution de conflits automatiques utilisés dans les systèmes adoptant le paradigme de la réplication optimiste. À partir de cet état de l'art, nous identifions et motivons notre problématique de recherche, c.-à-d. l'absence de mécanisme adapté aux systèmes P2P à large échelle sujet au churn permettant de réduire le surcoût induit par les mécanismes de résolution de conflits automatiques pour le type Séquence.
- Dans le chapitre 3, nous présentons notre approche pour présenter un tel mécanisme, c.-à-d. un mécanisme de résolution de conflits automatiques pour le type Séquence auquel nous associons un mécanisme de Garbage Collection (GC) de son surcoût

ne nécessitant pas de coordination synchrone entre les noeuds du système. Nous détaillons le fonctionnement de notre approche, sa validation par le biais d’une évaluation empirique puis comparons notre approche par rapport aux approches existantes. Finalement, nous concluons la présentation de notre approche en identifiant et en détaillant plusieurs de ses limites.

- Dans le ??, nous présentons MUTE, l’éditeur de texte collaboratif temps réel P2P chiffré de bout en bout que notre équipe de recherche développe dans le cadre de ses travaux de recherche. Nous présentons les différentes couches logicielles formant un pair et les services tiers avec lesquels les pairs interagissent, et détaillons nos travaux dans le cadre de ce projet, c.-à-d. l’intégration de notre mécanisme de résolution de conflits automatiques pour le type Séquence et le développement de la couche de livraison des messages associée. Pour chaque couche logicielle, nous identifions ses limites et présentons de potentielles pistes d’améliorations.
- Finalement, nous récapitulons dans le chapitre 4 les contributions réalisées dans le cadre de cette thèse. Puis nous clotûrons ce manuscrit en introduisant plusieurs des pistes de recherches que nous souhaiterons explorer dans le cadre de nos travaux futurs.

1.4 Publications

Notre travail sur la problématique identifiée dans la sous-section 1.2.1, c.-à-d. la proposition d’un mécanisme ne nécessitant aucune coordination synchrone pour réduire le surcoût des CRDTs pour le type Séquence, a donné lieu à des publications à différents stades de son avancement :

- (i) Dans [20], nous motivons le problème identifié et présentons l’idée de notre approche pour y répondre.
- (ii) Dans [21], nous détaillons une première partie de notre approche et présentons notre protocole d’évaluation expérimentale ainsi que ses premiers résultats.
- (iii) Dans [13], nous détaillons notre proposition dans son entièreté. Nous accompagnons cette proposition d’une évaluation expérimentale poussée. Finalement, nous complétons notre travail d’une discussion analysant ses limites connues et présentant des pistes de travail possibles pour y répondre.

Nous précisons ci-dessous les informations relatives à chacun de ces articles.

Efficient renaming in CRDTs [20]

Auteur Matthieu Nicolas

Article de position à Middleware 2018 - 19th ACM/IFIP International Middleware Conference (Doctoral Symposium), Dec 2018, Rennes, France.

Abstract *Sequence Conflict-free Replicated Data Types (CRDTs)* allow to replicate and edit, without any kind of coordination, sequences in distributed systems. To ensure convergence, existing works from the literature add metadata to each element but they do not bound its footprint, which impedes their adoption. Several approaches were proposed to address this issue but they do not fit a fully distributed setting. In this paper, we present our ongoing work on the design and validation of a fully distributed renaming mechanism, setting a bound to the metadata’s footprint. Addressing this issue opens new perspectives of adoption of these CRDTs in distributed applications.

Efficient Renaming in Sequence CRDTs [21]

Auteurs Matthieu Nicolas, G  rald Oster, Olivier Perrin

Article de workshop    PaPoC 2020 - 7th Workshop on Principles and Practice of Consistency for Distributed Data, Apr 2020, Heraklion / Virtual, Greece.

Abstract To achieve high availability, large-scale distributed systems have to replicate data and to minimise coordination between nodes. Literature and industry increasingly adopt Conflict-free Replicated Data Types (CRDTs) to design such systems. CRDTs are data types which behave as traditional ones, e.g. the Set or the Sequence. However, unlike traditional data types, they are designed to natively support concurrent modifications. To this end, they embed in their specification a conflict-resolution mechanism.

To resolve conflicts in a deterministic manner, CRDTs usually attach identifiers to elements stored in the data structure. Identifiers have to comply with several constraints, such as uniqueness or belonging to a dense order. These constraints may hinder the identifiers’ size from being bounded. As the system progresses, identifiers tend to grow. This inflation deepens the overhead of the CRDT over time, leading to performance issues.

To address this issue, we propose a new CRDT for Sequence which embeds a renaming mechanism. It enables nodes to reassign shorter identifiers to elements in an uncoordinated manner. Experimental results demonstrate that this mechanism decreases the overhead of the replicated data structure and eventually limits it.

Efficient Renaming in Sequence CRDTs [13]

Auteurs Matthieu Nicolas, G  rald Oster, Olivier Perrin

Article de journal dans IEEE Transactions on Parallel and Distributed Systems, Institute of Electrical and Electronics Engineers, 2022, 33 (12), pp.3870-3885.

Abstract To achieve high availability, large-scale distributed systems have to replicate data and to minimise coordination between nodes. For these purposes, literature and industry increasingly adopt Conflict-free Replicated Data Types (CRDTs) to design such systems. CRDTs are new specifications of existing data types, e.g. Set or Sequence. While CRDTs have the same behaviour as previous specifications in sequential executions,

they actually shine in distributed settings as they natively support concurrent updates. To this end, CRDTs embed in their specification conflict resolution mechanisms. These mechanisms usually rely on identifiers attached to elements of the data structure to resolve conflicts in a deterministic and coordination-free manner. Identifiers have to comply with several constraints, such as being unique or belonging to a dense total order. These constraints may hinder the identifier size from being bounded. Identifiers hence tend to grow as the system progresses, which increases the overhead of CRDTs over time and leads to performance issues. To address this issue, we propose a novel Sequence CRDT which embeds a renaming mechanism. It enables nodes to reassign shorter identifiers to elements in an uncoordinated manner. Experimental results demonstrate that this mechanism decreases the overhead of the replicated data structure and eventually minimises it.

Chapitre 2

État de l'art

Sommaire

2.1	Modèle du système	11
2.2	Types de données répliquées sans conflits	12

2.1 Modèle du système

Le système que nous considérons est un système Pair-à-Pair (P2P) à large échelle. Il est composé d'un ensemble de noeuds dynamique. En d'autres termes, un noeud peut rejoindre ou quitter le système à tout moment.

À un instant donné, un noeud est soit connecté, soit déconnecté. Nous considérons possible qu'un noeud se déconnecte de manière définitive, sans indication au préalable. Ainsi, du point de vue des autres noeuds du système, il est impossible de déterminer le statut d'un noeud déconnecté. Ce dernier peut être déconnecté de manière temporaire ou définitive. Toutefois, nous assimilons les noeuds déconnectés de manière définitive à des noeuds ayant quittés le système, ceux-ci ne participant plus au système.

Dans ce système, nous considérons comme confondus les noeuds et clients. Un noeud correspond alors à un appareil d'un-e utilisateur-riche du système. Un-e même utilisateur-riche peut prendre part au système au travers de différents appareils, nous considérons alors chaque appareil comme un noeud distinct.

Le système consiste en une application permettant de répliquer une donnée. Chaque noeud du système possède en local une copie de la donnée. Les noeuds peuvent consulter et éditer leur copie locale à tout moment, sans se coordonner entre eux. Les modifications sont appliquées à la copie locale immédiatement et de manière atomique. Les modifications sont ensuite transmises aux autres noeuds de manière asynchrone par le biais de messages, afin qu'ils puissent à leur tour intégrer les modifications à leur copie. L'application garantit la convergence à terme des copies.

Définition 6 (Convergence à terme). La convergence à terme est une propriété de sûreté indiquant que l'ensemble des noeuds du système ayant intégrés le même ensemble de

modifications obtiendront des états équivalents⁸.

Les noeuds communiquent entre eux par l'intermédiaire d'un réseau non-fiable. Les messages envoyés peuvent être perdus, ré-ordonnés et/ou dupliqués. Le réseau est aussi sujet à des partitions, qui séparent les noeuds en des sous-groupes disjoints. Aussi, nous considérons que les noeuds peuvent initier de leur propre chef des partitions réseau : des groupes de noeuds peuvent décider de travailler de manière isolée pendant une certaine durée, avant de se reconnecter au réseau.

Pour compenser les limitations du réseau, les noeuds reposent sur une couche de livraison de messages. Cette couche permet de garantir un modèle de livraison donné des messages à l'application. En fonction des garanties du modèle de livraison sélectionné, cette couche peut ré-ordonner les messages reçus avant de les livrer à l'application, dé-dupliquer les messages, et détecter et ré-échanger les messages perdus. Nous considérons a minima que la couche de livraison garantit la livraison à terme des messages.

Définition 7 (Livraison à terme). La livraison à terme est un modèle de livraison garantissant que l'ensemble des messages du système seront livrés à l'ensemble des noeuds du système à terme.

Finalement, nous supposons que les noeuds du système sont honnêtes. Les noeuds ne peuvent dévier du protocole de la couche de livraison des messages ou de l'application. Les noeuds peuvent cependant rencontrer des défaillances. Nous considérons que les noeuds disposent d'une mémoire durable et fiable. Ainsi, nous considérons que les noeuds peuvent restaurer le dernier état valide, c.-à-d. pas en cours de modification, qu'il possédait juste avant la défaillance.

2.2 Types de données répliquées sans conflits

Afin d'offrir une haute disponibilité à leurs clients et afin d'accroître leur tolérance aux pannes [22], les systèmes distribués peuvent adopter le paradigme de la réplication optimiste [3]. Ce paradigme consiste à ce que chaque noeud composant le système possède une copie de la donnée répliquée. Chaque noeud possède le droit de la consulter et de la modifier, sans coordination préalable avec les autres noeuds. Les noeuds peuvent alors temporairement diverger, c.-à-d. posséder des états différents. Un mécanisme de synchronisation leur permet ensuite de partager leurs modifications respectives et d'obtenir de nouveau des états équivalents, c.-à-d. de converger à terme [4].

Pour permettre aux noeuds de converger, les protocoles de réplication optimiste ordonnent généralement les événements se produisant dans le système distribué. Pour les ordonner, la littérature repose généralement sur la relation de causalité entre les événements, qui est définie par la relation *happens-before* [23]. Nous l'adaptions ci-dessous à notre contexte, en ne considérant que les modifications⁹ effectuées et celles intégrées :

8. Nous considérons comme équivalents deux états pour lesquels chaque observateur du type de données renvoie un même résultat, c.-à-d. les deux états sont indifférenciables du point de vue des utilisatrices du système.

9. Nous utilisons le terme *modifications* pour désigner les *opérations de modifications* des types abstraits de données afin d'éviter une confusion avec le terme *opération* introduit ultérieurement.

Définition 8 (Relation *happens-before*). La relation *happens-before* indique qu’une modification m_1 a eu lieu avant une modification m_2 , notée $m_1 \rightarrow m_2$, si et seulement si une des conditions suivantes est satisfaite :

- (i) m_1 a été effectuée avant m_2 sur le même noeud.
- (ii) m_1 a été intégrée par le noeud auteur de m_2 avant qu’il n’effectue m_2 .
- (iii) Il existe une modification m telle que $m_1 \rightarrow m \wedge m \rightarrow m_2$.

Dans le cadre d’un système distribué, nous notons que la relation *happens-before* ne permet pas d’établir un ordre total entre les modifications. En effet, deux modifications m_1 et m_2 peuvent être effectuées en parallèle par deux noeuds différents, sans avoir connaissance de la modification de leur pair respectif. De telles modifications sont alors dites *concurrentes* :

Définition 9 (Concurrence). Deux modifications m_1 et m_2 sont concurrentes, noté $m_1 \parallel m_2$, si et seulement si $m_1 \nrightarrow m_2 \wedge m_1 \nrightarrow m_2$.

Lorsque les modifications possibles sur un type de données sont commutatives, l’intégration des modifications effectuées par les autres noeuds, même concurrentes, ne nécessite aucun mécanisme particulier. Cependant, les modifications permises par un type de données ne sont généralement pas commutatives car de sémantiques contraires, e.g. l’ajout et la suppression d’un élément dans une Collection. Ainsi, une exécution distribuée peut mener à la génération de modifications concurrentes non commutatives. Nous parlons alors de conflits.

Avant d’illustrer notre propos avec un exemple, nous introduisons la spécification algébrique du type Ensemble dans la Figure 2.1 sur laquelle nous nous basons.

```

payload
 $S \in \text{Set}\langle E \rangle$ 

constructor
 $emp : \quad \longrightarrow S$ 

mutators
 $add : S \times E \longrightarrow S$ 
 $rmv : S \times E \longrightarrow S$ 

queries
 $len : S \longrightarrow \mathbb{N}$ 
 $rd : S \longrightarrow S$ 

```

FIGURE 2.1 – Spécification algébrique du type abstrait usuel Ensemble

Un Ensemble est une collection dynamique non-ordonnée d’éléments de type E . Cette spécification définit que ce type dispose d’un constructeur, *emp*, permettant de générer un ensemble vide.

La spécification définit deux modifications sur l’ensemble :

- (i) $add(s, e)$, qui permet d'ajouter un élément donné e à un ensemble s . Cette modification renvoie un nouvel ensemble construit de la manière suivante :

$$add(s, e) = s \cup \{e\}$$

- (ii) $rmv(s, e)$, qui permet de retirer un élément donné e d'un ensemble s . Cette modification renvoie un nouvel ensemble construit de la manière suivante :

$$rmv(s, e) = s \setminus \{e\}$$

Elle définit aussi deux observateurs :

- (i) $len(s)$, qui permet de récupérer le nombre d'éléments présents dans un ensemble s .
(ii) $rd(s)$, qui permet de consulter l'état d'ensemble s . Dans le cadre de nos exemples, nous considérons qu'une consultation de l'état est effectuée de manière implicite à l'aide de rd après chaque modification.

Dans le cadre de ce manuscrit, nous travaillons sur des ensembles de caractères. Cette restriction du domaine se fait sans perte en généralité. En se basant sur cette spécification, nous présentons dans la Figure 2.2 un scénario où des noeuds effectuent en concurrence des modifications provoquant un conflit.

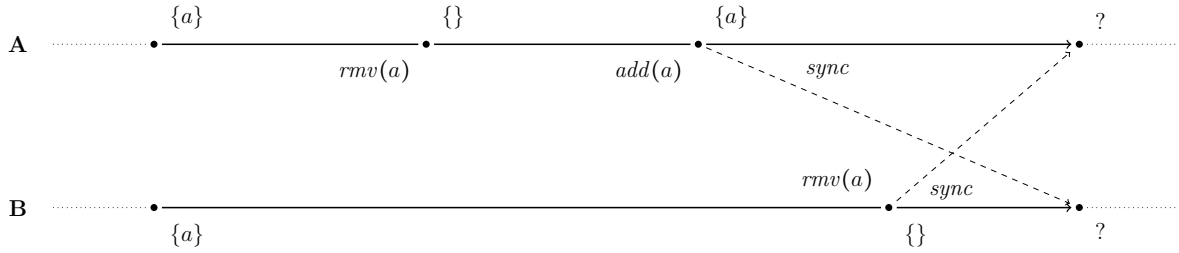


FIGURE 2.2 – Modifications concurrentes d'un Ensemble répliqué provoquant un conflit suite à l'ajout et la suppression d'un même élément

Dans cet exemple, deux noeuds A et B répliquent et partagent une même structure de données de type Ensemble. Les deux noeuds possèdent le même état initial : $\{a\}$. Le noeud A retire l'élément a de l'ensemble, en procédant à la modification $rmv(a)$. Puis, le noeud A ré-ajoute l'élément a dans l'ensemble via la modification $add(a)$. En concurrence, le noeud B retire lui aussi l'élément a de l'ensemble. Les deux noeuds se synchronisent ensuite.

À l'issue de ce scénario, l'état à produire n'est pas trivial : le noeud A a exprimé son intention d'ajouter l'élément a à l'ensemble, tandis que le noeud B a exprimé son intention contraire de retirer l'élément a de ce même ensemble. Ainsi, les états $\{a\}$ et $\{\}$ semblent tous les deux corrects et légitimes dans cette situation. Il est néanmoins primordial que les noeuds choisissent et convergent vers un même état pour leur permettre de poursuivre leur collaboration. Pour ce faire, il est nécessaire de mettre en place un mécanisme de résolution de conflits, potentiellement automatique.

Les Conflict-free Replicated Data Types (CRDTs) [6, 24, 25] répondent à ce besoin.

Définition 10 (Conflict-free Replicated Data Type). Les CRDTs sont de nouvelles spécifications des types de données existants, e.g. l'Ensemble ou la Séquence. Ces nouvelles spécifications sont conçues pour être utilisées dans des systèmes distribués adoptant la réplication optimiste. Ainsi, elles offrent les deux propriétés suivantes :

- (i) Les CRDTs peuvent être modifiés sans coordination avec les autres noeuds.
- (ii) Les CRDTs garantissent la *convergence forte* [6].

Définition 11 (Convergence forte). La convergence forte est une propriété de sûreté indiquant que l'ensemble des noeuds d'un système ayant intégrés le même ensemble de modifications obtiendront des états équivalents, sans échange de message supplémentaire.

Pour offrir la propriété de *convergence forte*, la spécification des CRDTs reposent sur la théorie des treillis [26] :

Définition 12 (Spécification des CRDTs). Les CRDTs sont spécifiés de la manière suivante :

- (i) Les différents états possibles d'un CRDT forment un sup-demi-treillis, possédant une relation d'ordre partiel \leq .
- (ii) Les modifications génèrent par inflation un nouvel état supérieur ou égal à l'état original d'après \leq .
- (iii) Il existe une fonction de fusion qui, pour toute paire d'états, génère l'état minimal supérieur d'après \leq aux deux états fusionnés. Nous parlons alors de borne supérieure ou de Least Upper Bound (LUB) pour catégoriser l'état résultant de cette fusion.

Malgré leur spécification différente, les CRDTs partagent la même sémantique, c.-à-d. le même comportement, et la même interface que les types séquentiels¹⁰ correspondants du point de vue des utilisateur-rices. Ainsi, les CRDTs partagent le comportement des types séquentiels dans le cadre d'exécutions séquentielles. Cependant, ils définissent aussi une sémantique additionnelle pour chaque type de conflit ne pouvant se produire que dans le cadre d'une exécution distribuée.

Plusieurs sémantiques valides peuvent être proposées pour résoudre un type de conflit. Un CRDT se doit donc de préciser quelle sémantique il choisit.

L'autre aspect définissant un CRDT donné est le modèle qu'il adopte pour propager les modifications. Au fil des années, la littérature a établi et défini plusieurs modèles dit de synchronisation, chacun ayant ses propres besoins et avantages. De fait, plusieurs CRDTs peuvent être proposés pour un même type donné en fonction du modèle de synchronisation choisi.

Ainsi, ce qui définit un CRDT est sa ou ses sémantiques en cas de conflits et son modèle de synchronisation. Dans les prochaines sections, nous présentons les différentes sémantiques possibles pour un type donné, l'Ensemble, en guise d'exemple. Nous présentons ensuite les différents modèles de synchronisation proposés dans la littérature, et détaillons leurs contraintes et impact sur les CRDT les adoptant, toujours en utilisant le même exemple.

Matthieu: TODO : Faire le lien avec les travaux de Burckhardt [27] et les MRDTs [28]

10. Nous dénotons comme *types séquentiels* les spécifications usuelles des types de données supposant une exécution séquentielle de leurs modifications.

Chapitre 3

Renommage dans une séquence répliquée

Sommaire

3.1	Présentation de l'approche	18
3.1.1	Définition de l'opération de renommage	18
3.2	Introduction de l'opération <i>rename</i>	19
3.2.1	Opération de renommage proposée	19
3.2.2	Gestion des opérations concurrentes au renommage	21
3.2.3	Évolution du modèle de livraison des opérations	23
3.3	Gestion des opérations <i>rename</i> concurrentes	25
3.3.1	Conflits en cas de renommages concurrents	25
3.3.2	Relation de priorité entre renommages	26
3.3.3	Algorithme d'annulation de l'opération de renommage	28
3.3.4	Processus d'intégration d'une opération	32
3.3.5	Règles de récupération de la mémoire des états précédents	37
3.4	Validation	39
3.4.1	Complexité en temps des opérations	39
3.4.2	Expérimentations	43
3.4.3	Résultats	44
3.5	Discussion	51
3.5.1	Stratégie de génération des opérations <i>rename</i>	51
3.5.2	Stockage des états précédents sur disque	52
3.5.3	Compression et limitation de la taille de l'opération <i>rename</i>	53
3.5.4	Définition de relations de priorité pour minimiser les traitements	53
3.5.5	Report de la transition vers la nouvelle époque cible	55
3.5.6	Utilisation de l'opération de renommage comme mécanisme de compression du log d'opérations	55
3.5.7	Implémentation alternative de l'intégration de l'opération <i>rename</i> basée sur le journal d'opérations	57
3.6	Comparaison avec les approches existantes	59

3.6.1	Core-Nebula	59
3.6.2	LSEQ	60
3.7	Conclusion	61

3.1 Présentation de l'approche

Nous proposons un nouveau CRDT pour la *Sequence* appartenant à l'approche des identifiants densément ordonnées : *RenamableLogootSplit* [21, 13]. Cette structure de données permet aux pairs d'insérer et de supprimer des éléments au sein d'une séquence répliquée. Nous introduisons une opération *rename* qui permet de (i) réassigner des identifiants plus courts aux différents éléments de la séquence (ii) fusionner les blocs composant la séquence. Ces deux actions permettent à l'opération *rename* de produire un nouvel état minimisant son surcoût en métadonnées.

3.1.1 Définition de l'opération de renommage

L'objectif de l'opération *rename* est de réassigner de nouveaux identifiants aux éléments de la séquence répliquée sans modifier son contenu. Puisque les identifiants sont des métadonnées utilisées par la structure de données uniquement afin de résoudre les conflits, les utilisateurs ignorent leur existence. Les opérations *rename* sont donc des opérations systèmes : elles sont émises et appliquées par les noeuds en coulisses, sans aucune intervention des utilisateurs.

Afin de garantir le respect du modèle de cohérence Cohérence forte à terme (SEC), nous définissons plusieurs propriétés de sécurité que l'opération *rename* doit respecter. Ces propriétés sont inspirées principalement par celles proposées dans [8].

Propriété 1. (Déterminisme) Les opérations *rename* sont intégrées par les noeuds sans aucune coordination. Pour assurer que l'ensemble des noeuds atteigne un état équivalent à terme, une opération *rename* donnée doit toujours générer le même nouvel identifiant à partir de l'identifiant courant.

Propriété 2. (Préservation de l'intention de l'utilisateur) Bien que l'opération *rename* n'est pas elle-même n'incarne pas une intention de l'utilisateur, elle ne doit pas entrer en conflit avec les actions des utilisateurs. Notamment, les opérations *rename* ne doivent pas annuler ou altérer le résultat d'opérations *insert* et *remove* du point de vue des utilisateurs.

Propriété 3. (Séquence bien formée) La séquence répliquée doit être bien formée. Appliquée une opération *rename* sur une séquence bien formée doit produire une nouvelle séquence bien formée. Une séquence bien formée doit respecter les propriétés suivantes :

Propriété 3.1. (Préservation de l'unicité) Chaque identifiant doit être unique. Donc, pour une opération *rename* donnée, chaque identifiant doit être associé à un nouvel identifiant distinct.

Propriété 3.2. (Préservation de l'ordre) Les éléments de la séquence doivent être triés en fonction de leur identifiants. L'ordre existant entre les identifiants initiaux doit donc être préservé par l'opération *rename*.

Propriété 4. (Commutativité avec les opérations concurrentes) Les opérations concurrentes peuvent être livrées dans des ordres différents à chaque noeud. Afin de garantir la convergence des répliquas, l'ordre d'application d'un ensemble d'opérations concurrentes ne doit pas avoir d'impact sur l'état obtenu. L'opération *rename* doit donc être commutative avec n'importe quelle opération concurrente.

La Propriété 4 est particulièrement difficile à assurer. Cette difficulté est due au fait que les opérations *rename* modifient les identifiants assignés aux éléments. Cependant, les autres opérations telles que les opérations *insert* et *remove* reposent sur ces identifiants pour spécifier où insérer les éléments ou lesquels supprimer. Les opérations *rename* sont donc intrinsèquement incompatibles avec les opérations *insert* et *remove* concurrentes. De la même manière, des opérations *rename* concurrentes peuvent réassigner des identifiants différents à des mêmes éléments. Les opérations *rename* concurrentes ne sont donc pas commutatives. Par conséquent, il est nécessaire de concevoir et d'utiliser des méthodes de résolution de conflits pour assurer la Propriété 4.

Dans un souci de simplicité, la présentation de l'opération *rename* est divisée en deux parties. Dans la section 3.2, nous présentons l'opération *rename* proposée avec l'hypothèse qu'aucune opération *rename* concurrente ne peut être générée. Cette hypothèse nous permet de nous concentrer sur le fonctionnement de l'opération *rename* elle-même ainsi que sur comment gérer les opérations *insert* et *remove* concurrentes. Ensuite, dans la section 3.3, nous supprimons cette hypothèse. Nous présentons alors notre approche pour gérer les scénarios avec des opérations *rename* concurrentes.

3.2 Introduction de l'opération *rename*

3.2.1 Opération de renommage proposée

Notre opération de renommage permet à RenamableLogootSplit de réduire le surcoût en métadonnées des séquences répliquées. Pour ce faire, elle réassigne des identifiants arbitraires aux éléments de la séquence.

Son comportement est illustré dans la Figure 3.1. Dans cet exemple, le noeud A initie une opération *rename* sur son état local. Tout d'abord, le noeud A génère un nouvel identifiant à partir du premier tuple de l'identifiant du premier élément de la séquence (i_0^{B0}). Pour générer ce nouvel identifiant, le noeud A reprend la position de ce tuple (i) mais utilise son propre identifiant de noeud (**A**) et numéro de séquence actuel (1). De plus, son offset est mis à 0. Le noeud A réassigne l'identifiant résultant (i_0^{A1}) au premier élément de la séquence, comme décrit dans la Figure 3.1a. Ensuite, le noeud A dérive des identifiants contigus pour tous les éléments restants en incrémentant de manière successive l'offset (i_1^{A1} , i_2^{A1} , i_3^{A1}), comme présenté dans la Figure 3.1b. Comme nous assignons des identifiants consécutifs à tous les éléments de la séquence, nous pouvons au final agréger

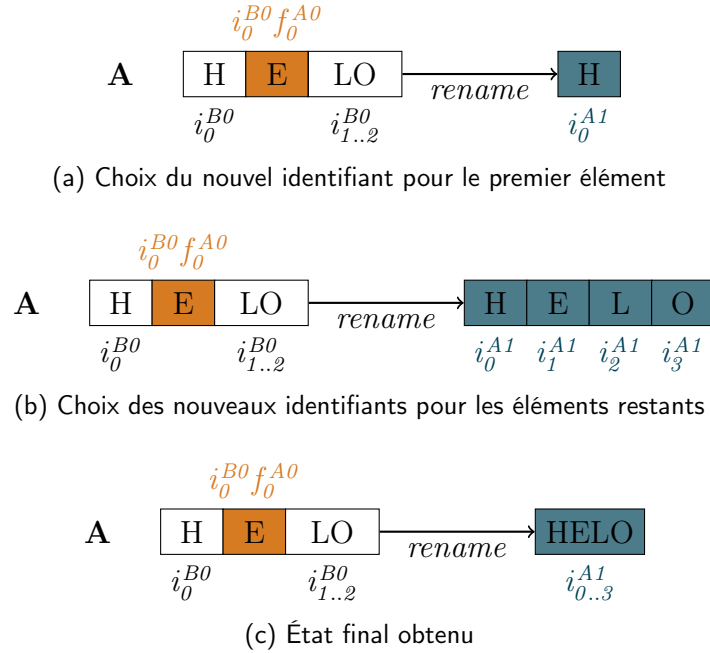


FIGURE 3.1 – Renommage de la séquence sur le noeud A

ces éléments en un seul bloc, comme illustré en Figure 3.1c. Ceci permet aux noeuds de bénéficier au mieux de la fonctionnalité des blocs et de minimiser le surcoût en métadonnées de l'état résultat.

Pour converger, les autres noeuds doivent renommer leur état de manière identique. Cependant, ils ne peuvent pas simplement remplacer leur état courant par l'état généré par le renommage. En effet, ils peuvent avoir modifié en concurrence leur état. Afin de ne pas perdre ces modifications, les noeuds doivent traiter l'opération *rename* eux-mêmes. Pour ce faire, le noeud qui a généré l'opération *rename* diffuse son *ancien état* aux autres.

Définition 13 (Ancien état). Un *ancien état* est la liste des intervalles d'identifiants qui composent l'état courant de la séquence répliquée au moment du renommage.

De ce fait, nous définissons l'opération *rename* de la manière suivante :

Définition 14 (*rename*). Une opération *rename* est un triplet $\langle \text{nodeId}, \text{nodeSeq}, \text{formerState} \rangle$ où

- *nodeId* est l'identifiant du noeud qui a générée l'opération *rename*.
- *nodeSeq* est le numéro de séquence du noeud au moment de la génération de l'opération *rename*.
- *formerState* est l'ancien état du noeud au moment du renommage.

En utilisant ces données, les autres noeuds calculent le nouvel identifiant de chaque identifiant renommé. Concernant les identifiants insérés de manière concurrente au renommage, nous expliquons dans la sous-section 3.2.2 comment les noeuds peuvent les renommer de manière déterministe.

3.2.2 Gestion des opérations concurrentes au renommage

Après avoir appliqué des opérations *rename* sur leur état local, les noeuds peuvent recevoir des opérations concurrentes. La Figure 3.2 illustre de tels cas.

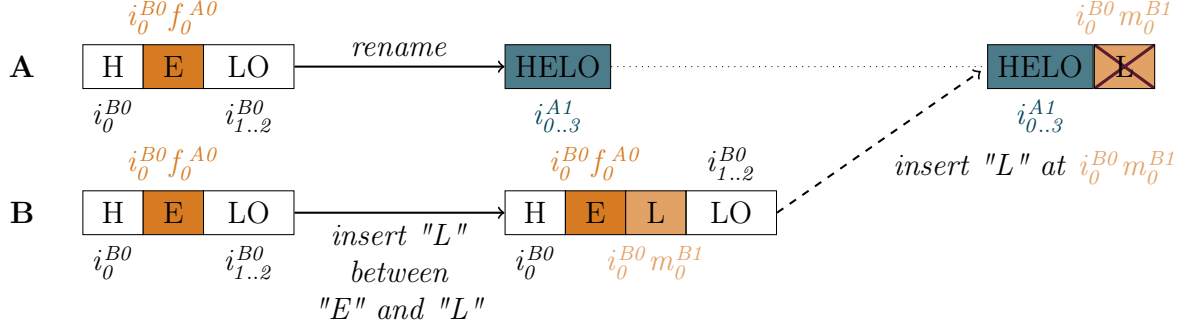


FIGURE 3.2 – Modifications concurrentes menant à une anomalie

Dans cet exemple, le noeud B insère un nouvel élément "L", lui assigne l'identifiant $i_0^{B0} m_0^{B1}$ et diffuse cette modification, de manière concurrente à l'opération *rename* décrite dans la Figure 3.2. À la réception de l'opération *insert*, le noeud A ajoute l'élément inséré au sein de sa séquence, en utilisant l'identifiant de l'élément pour déterminer sa position. Cependant, puisque les identifiants ont été modifiés par l'opération *rename* concurrente, le noeud A insère le nouvel élément à la fin de sa séquence (puisque $i_3^{A1} <_{id} i_0^{B0} m_0^{B1}$) au lieu de l'insérer à la position souhaitée. Comme illustré par cet exemple, appliquer naïvement les modifications concurrentes provoquerait des anomalies. Il est donc nécessaire de traiter les opérations concurrentes aux opérations *rename* de manière particulière.

Tout d'abord, les noeuds doivent détecter les opérations concurrentes aux opérations *rename*. Pour cela, nous utilisons un système basé sur des *époques*. Initialement, la séquence répliquée débute à l'époque *origine* notée ε_0 . Chaque opération *rename* introduit une nouvelle époque et permet aux noeuds d'y avancer depuis l'époque précédente. Par exemple, l'opération *rename* décrite dans Figure 3.2 permet aux noeuds de faire progresser leur état de ε_0 à ε_{A1} . Nous définissons les époques de la manière suivante :

Définition 15 (Époque). Une époque est un couple $\langle \text{nodeId}, \text{nodeSeq} \rangle$ où

- *nodeId* est l'identifiant du noeud qui a générée cette époque.
- *nodeSeq* est le numéro de séquence du noeud au moment de la génération de cette époque.

Notons que l'époque générée est caractérisée et identifiée de manière unique par son couple $\langle \text{nodeId}, \text{nodeSeq} \rangle$.

Au fur et à mesure que les noeuds reçoivent des opérations *rename*, ils construisent et maintiennent localement la *chaîne des époques*. Cette structure de données ordonne les époques en fonction de leur relation *parent-enfant* et associe à chaque époque l'*ancien état* correspondant (c.-à-d. l'*ancien état* inclus dans l'opération *rename* qui a généré cette époque). De plus, les noeuds marquent chaque opération avec leur époque courante au moment de génération de l'opération. À la réception d'une opération, les noeuds comparent l'époque de l'opération à l'époque courante de leur séquence.

Si les époques diffèrent, les noeuds doivent transformer l'opération avant de pouvoir l'intégrer. Les noeuds déterminent par rapport à quelles opérations *rename* doit être transformée l'opération reçue en calculant le chemin entre l'époque de l'opération et leur époque courante en utilisant la *chaîne des époques*.

Les noeuds utilisent la fonction `RENAMEID`, décrite dans l'Algorithme 1, pour transformer les opérations *insert* et *remove* par rapport aux opérations *rename*. Cet algorithme associe les identifiants d'une époque *parente* aux identifiants correspondant dans l'époque *enfant*. L'idée principale de cet algorithme est de renommer les identifiants inconnus au moment de la génération de l'opération *rename* en utilisant leur prédécesseur. Un exemple est présenté dans la Figure 3.3. Cette figure décrit le même scénario que la Figure 3.2, à l'exception que le noeud A utilise `RENAMEID` pour renommer les identifiants générés de façon concurrente avant de les insérer dans son état.

Algorithme 1 Fonctions principales pour renommer un identifiant

```

function RENAMEID(id, renamedIds, nId, nSeq)
    length ← renamedIds.length
    firstId ← renamedIds[0]
    lastId ← renamedIds[length - 1]
    pos ← position(firstId)

    if id < firstId then
        newFirstId ← new Id(pos, nId, nSeq, 0)
        return renIdLessThanFirstId(id, newFirstId)
    else if id ∈ renamedIds then
        index ← findIndex(id, renamedIds)
        return new Id(pos, nId, nSeq, index)
    else if lastId < id then
        newLastId ← new Id(pos, nId, nSeq, length - 1)
        return renIdGreaterThanLastId(id, newLastId)
    else
        return renIdFromPredId(id, renamedIds, pos, nId, nSeq)
    end if
end function

function RENIDFROMPREDID(id, renamedIds, pos, nId, nSeq)
    index ← findIndexOffPred(id, renamedIds)
    newPredId ← new Id(pos, nId, nSeq, index)

    return concat(newPredId, id)
end function

```

L'algorithme procède de la manière suivante. Tout d'abord, le noeud récupère le prédécesseur de l'identifiant donné $i_0^{B0} m_0^{B1}$ dans l'ancien état : $i_0^{B0} f_0^{A0}$. Ensuite, il calcule l'équivalent de $i_0^{B0} f_0^{A0}$ dans l'état renommé : i_1^{A1} . Finalement, le noeud A concatène cet identifiant et l'identifiant donné pour générer l'identifiant correspondant dans l'époque *enfant* : $i_1^{A1} i_0^{B0} m_0^{B1}$. En réassignant cet identifiant à l'élément inséré de manière concurrente, le noeud A peut l'insérer à son état tout en préservant l'ordre souhaité.

`RENAMEID` permet aussi aux noeuds de gérer le cas contraire : intégrer des opérations *rename* distantes sur leur copie locale alors qu'ils ont précédemment intégré des modi-

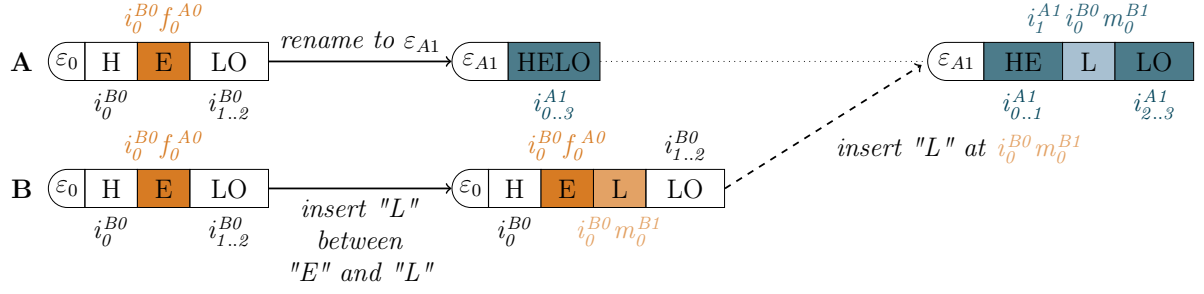


FIGURE 3.3 – Renommage de la modification concurrente avant son intégration en utilisant RENAMEID afin de maintenir l'ordre souhaité

fications concurrentes. Ce cas correspond à celui du noeud B dans la Figure 3.3. À la réception de l'opération *rename* du noeud A, le noeud B utilise RENAMEID sur chacun des identifiants de son état pour le renommer et atteindre un état équivalent à celui du noeud A.

L'Algorithme 1 présente seulement le cas principal de RENAMEID, c.-à-d. le cas où l'identifiant à renommer appartient à l'intervalle des identifiants formant l'ancien état ($firstId \leq_{id} id \leq_{id} lastId$). Les fonctions pour gérer les autres cas, c.-à-d. les cas où l'identifiant à renommer n'appartient pas à cet intervalle ($id <_{id} firstId$ ou $lastId <_{id} id$), sont présentées dans l'Annexe B.

L'algorithme que nous présentons ici permet aux noeuds de renommer leur état identifiant par identifiant. Une extension possible est de concevoir RENAMEBLOCK, une version améliorée qui renomme l'état bloc par bloc. RENAMEBLOCK réduirait le temps d'intégration des opérations *rename*, puisque sa complexité en temps ne dépendrait plus du nombre d'identifiants (c.-à-d. du nombre d'éléments) mais du nombre de blocs. De plus, son exécution réduirait le temps d'intégration des prochaines opérations *rename* puisque le mécanisme de renommage regroupe les éléments en moins de blocs.

3.2.3 Évolution du modèle de livraison des opérations

L'introduction de l'opération *rename* nécessite de faire évoluer le modèle de livraison des opérations associé à RenamableLogootSplit. Afin d'illustrer cette nécessité, considérons l'exemple suivant :

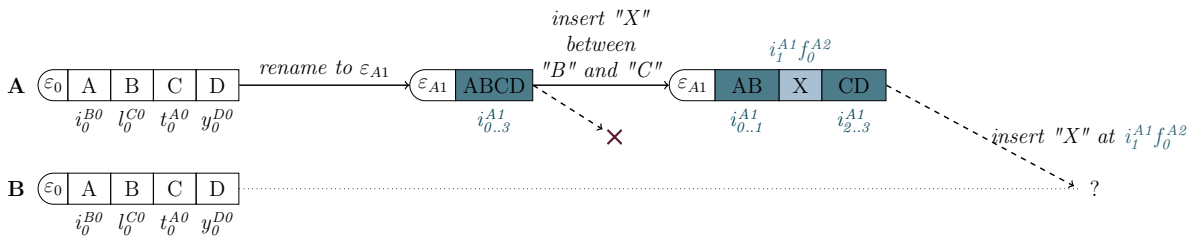


FIGURE 3.4 – Livraison d'une opération *insert* sans avoir reçu l'opération *rename* précédente

Dans la Figure 3.4, les noeuds A et B répliquent tous deux une même séquence, contenant les éléments "ABCD". Tout d'abord, le noeud A procède au renommage de cet état. Puis il insère un nouvel élément, "X", entre "B" et "C". Les opérations correspondantes aux actions du noeud A sont diffusées sur le réseau.

Cependant, l'opération *rename* n'est pas livrée au noeud B, par exemple suite à un problème réseau. L'opération *insert* est quant à elle correctement livrée à ce dernier. Le noeud B doit alors intégrer dans son état un élément et l'identifiant qui lui est attaché. Mais cet identifiant est issu d'une époque (ε_{A1}) différente de son époque actuelle (ε_0) et dont le noeud n'avait pas encore connaissance. Il convient de s'interroger sur l'état à produire dans cette situation.

Comme nous l'avions déjà illustré par la Figure 3.2, les identifiants d'une époque ne peuvent être comparés qu'aux identifiants de la même époque. Tenter d'intégrer une opération *insert* ou *remove* provenant d'une époque encore inconnue ne résulterait qu'en un état incohérent et une transgression de l'intention utilisateur. Il est donc nécessaire d'empêcher ce scénario de se produire.

Pour cela, nous proposons de faire évoluer le modèle de livraison des opérations de RenamableLogootSplit. Celui-ci repose sur celui de LogootSplit, que nous avons défini dans la ???. Pour rappel, ce modèle requiert que (i) les opérations soient livrées qu'une seule et unique fois au CRDT, (ii) les opérations *remove* soient livrées au CRDT qu'après les opérations *insert* ajoutant les éléments à supprimer.

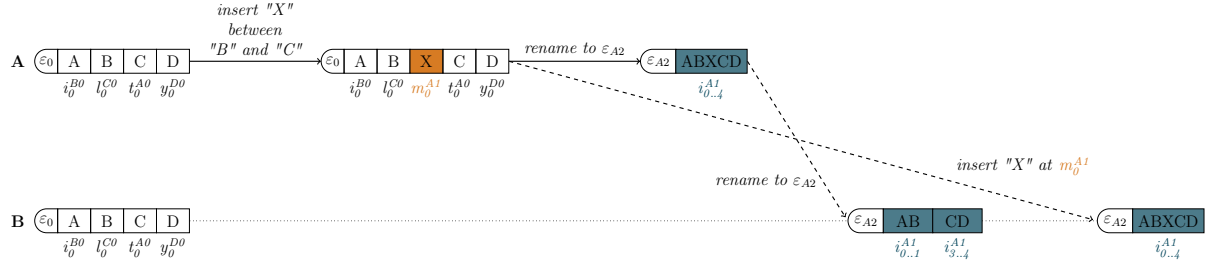
Pour prévenir les scénarios tels que celui illustré par la Figure 3.4 nous y ajoutons la règle suivante : les opérations *rename* doivent être livrées à la structure de données avant les opérations qui ont une dépendance causale vers ces dernières. Nous obtenons donc le modèle de livraison suivant :

Définition 16 (Exactly-once + Causal remove + Epoch-based). Le modèle de livraison *Exactly-once + Causal remove + Epoch-based* définit les 4 règles suivantes sur la livraison des opérations :

- (i) Une opération doit être livrée à l'ensemble des noeuds à terme,
- (ii) Une opération doit être livrée qu'une seule et unique fois aux noeuds,
- (iii) Une opération *remove* doit être livrée à un noeud une fois que les opérations *insert* des éléments concernés par la suppression ont été livrées à ce dernier.
- (iv) Une opération doit être livrée à un noeud une fois que l'opération *rename* une fois que l'opération *rename* qui introduit son époque de génération a été délivrée à ce dernier.

Il est cependant intéressant de noter que la livraison de l'opération *rename* ne requiert pas de contraintes supplémentaires. Notamment, une opération *rename* peut être livrée dans le désordre par rapport aux opérations *insert* et *remove* dont elle dépend causalement. La Figure 3.5 présente un exemple de ce cas figure.

Dans cet exemple, les noeuds A et B répliquent tous deux une même séquence, contenant les éléments "ABCD". Le noeud A commence par insérer un nouvel élément, "X", entre les éléments "B" et "C". Puis il procède au renommage de son état. Les opérations correspondantes aux actions du noeud A sont diffusées sur le réseau.

3.3. Gestion des opérations *rename* concurrentesFIGURE 3.5 – Livraison désordonnée d'une opération *rename* et de l'opération *insert* qui la précède

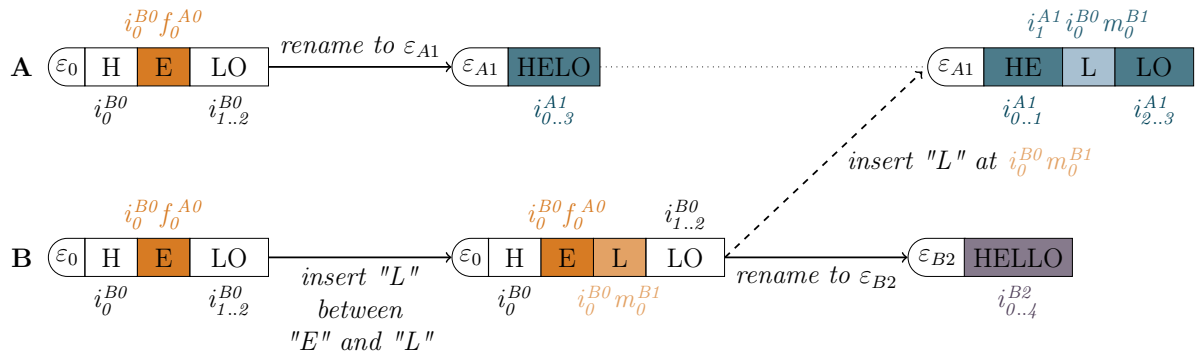
Cependant, suite à un aléa du réseau, le noeud B reçoit les deux opérations *insert* et *rename* dans le désordre. L'opération *rename* est donc livrée en première au noeud B. En utilisant les informations contenues dans l'opération, le noeud B est renommé chaque identifiant composant son état.

Ensuite, le noeud B reçoit l'opération *insert*. Comme l'époque de génération de l'opération *insert* (ε_0) est différente de celle de son état courant (ε_{A2}), le noeud B utilise *RENAMEID* pour renommer l'identifiant avant de l'insérer. m_0^{A1} faisant partie de l'*ancien état*, le noeud B utilise l'index de cet identifiant dans l'*ancien état* (2) pour calculer son équivalent à l'époque ε_{A2} (i_2^{A2}). Le noeud B insère l'élément "X" avec ce nouvel identifiant et converge alors avec le noeud A, malgré la livraison dans le désordre des opérations.

3.3 Gestion des opérations *rename* concurrentes

3.3.1 Conflits en cas de renommages concurrents

Nous considérons à présent les scénarios avec des opérations *rename* concurrentes. Figure 3.6 développe le scénario décrit précédemment dans Figure 3.3.

FIGURE 3.6 – Opérations *rename* concurrentes menant à des états divergents

Après avoir diffusé son opération *insert*, le noeud B effectue une opération *rename* sur son état. Cette opération réassigne à chaque élément un nouvel identifiant à partir de l'identifiant du premier élément de la séquence (i_0^{B0}), de l'identifiant du noeud (**B**) et de son numéro de séquence courant (2). Cette opération introduit aussi une nouvelle

époque : ε_{B2} . Puisque l'opération *rename* de A n'a pas encore été livrée au noeud B à ce moment, les deux opérations *rename* sont concurrentes.

Puisque des époques concurrentes sont générées, les époques forment désormais l'*arbre des époques*. Nous représentons dans la Figure 3.7 l'*arbre des époques* que les noeuds obtiennent une fois qu'ils se sont synchronisés à terme. Les époques sont représentées sous la forme de noeuds de l'arbre et la relation *parent-enfant* entre elles est illustrée sous la forme de flèches noires.

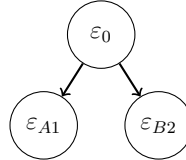


FIGURE 3.7 – *Arbre des époques* correspondant au scénario décrit dans la Figure 3.6

À l'issue du scénario décrit dans la Figure 3.6, les noeuds A et B sont respectivement aux époques ε_{A1} et ε_{B2} . Pour converger, tous les noeuds devraient atteindre la même époque à terme. Cependant, la fonction `RENAMEID` décrite dans l'Algorithme 1 permet seulement aux noeuds de progresser d'une époque *parente* à une de ses époques *enfants*. Le noeud A (resp. B) est donc dans l'incapacité de progresser vers l'époque du noeud B (resp. A). Il est donc nécessaire de faire évoluer notre mécanisme de renommage pour sortir de cette impasse.

Tout d'abord, les noeuds doivent se mettre d'accord sur une époque commune de l'*arbre des époques* comme époque cible. Afin d'éviter des problèmes de performances dus à une coordination synchrone, les noeuds doivent sélectionner cette époque de manière non-coordonnée, c.-à-d. en utilisant seulement les données présentes dans l'*arbre des époques*. Nous présentons un tel mécanisme dans la sous-section 3.3.2.

Ensuite, les noeuds doivent se déplacer à travers l'*arbre des époques* afin d'atteindre l'époque cible. La fonction `RENAMEID` permet déjà aux noeuds de descendre dans l'arbre. Les cas restants à gérer sont ceux où les noeuds se trouvent actuellement à une époque *soeur* ou *cousine* de l'époque cible. Dans ces cas, les noeuds doivent être capable de remonter dans l'*arbre des époques* pour retourner au Plus Petit Ancêtre Commun (PPAC) de l'époque courante et l'époque cible. Ce déplacement est en fait similaire à annuler l'effet des opérations *rename* précédemment appliquées. Nous proposons un algorithme, `REVERTRENAMEID`, qui remplit cet objectif dans la sous-section 3.3.3.

3.3.2 Relation de priorité entre renommages

Pour que chaque noeud sélectionne la même époque cible de manière non-coordonnée, nous définissons la relation *priority*.

Définition 17 (Relation *priority* $<_{\varepsilon}$). La relation *priority* $<_{\varepsilon}$ est un ordre strict total sur l'ensemble des époques. Elle permet aux noeuds de comparer n'importe quelle paire d'époques.

En utilisant la relation *priority*, nous définissons l'époque cible de la manière suivante :

Définition 18 (Époque cible). L'époque cible est l'époque de l'ensemble des époques vers laquelle les noeuds doivent progresser. Les noeuds sélectionnent comme époque cible l'époque maximale d'après l'ordre établi par *priority*.

Pour définir la relation *priority*, nous pouvons choisir entre plusieurs stratégies. Dans le cadre de ce travail, nous utilisons l'ordre lexicographique sur le chemin des époques dans l'*arbre des époques*. La Figure 3.8 fournit un exemple.

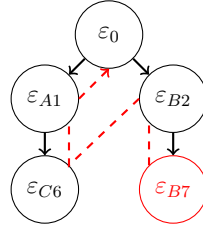
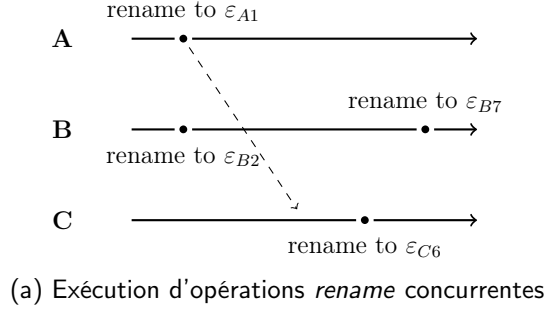


FIGURE 3.8 – Sélectionner l'époque cible d'une exécution d'opérations *rename* concurrentes

La Figure 3.8a décrit une exécution dans laquelle trois noeuds A, B et C génèrent plusieurs opérations avant de se synchroniser à terme. Comme seules les opérations *rename* sont pertinentes pour le problème qui nous occupe, nous représentons seulement ces opérations dans cette figure. Initialement, le noeud A génère une opération *rename* qui introduit l'époque ε_{A1} . Cette opération est livrée au noeud C, qui génère ensuite sa propre opération *rename* qui introduit l'époque ε_{C6} . De manière concurrente à ces opérations, le noeud B génère deux opérations *rename*, introduisant ε_{B2} et ε_{B7} .

Une fois que les noeuds se sont synchronisés, ils obtiennent l'*arbre des époques* représenté dans la Figure 3.8b. Dans cette figure, la flèche tireté rouge représente l'ordre entre les époques d'après la relation *priority* tandis que l'époque cible choisie est représentée sous la forme d'un noeud rouge.

Pour déterminer l'époque cible, les noeuds reposent sur la relation *priority*. D'après l'ordre lexicographique sur le chemin des époques dans l'*arbre des époques*, nous avons $\varepsilon_0 < \varepsilon_0\varepsilon_{A1} < \varepsilon_0\varepsilon_{A1}\varepsilon_{C6} < \varepsilon_0\varepsilon_{B2} < \varepsilon_0\varepsilon_{B2}\varepsilon_{B7}$. Chaque noeud sélectionne donc ε_{B7} comme époque cible de manière non-coordonnée.

D'autres stratégies pourraient être proposées pour définir la relation *priority*. Par exemple, *priority* pourrait reposer sur des métriques intégrées au sein des opérations *rename* pour représenter le travail accumulé sur le document. Cela permettrait de favoriser

la branche de l'arbre des époques avec le plus de collaborateurs actifs pour minimiser la quantité globale de calculs effectués par les noeuds du système. Nous approfondissons ce sujet dans la sous-section 3.5.4.

3.3.3 Algorithme d'annulation de l'opération de renommage

À présent, nous développons le scénario présenté dans la Figure 3.6. Dans la Figure 3.9, le noeud A reçoit l'opération *rename* du noeud B. Cette opération est concurrente à l'opération *rename* que le noeud A a appliqué précédemment. D'après la relation *priority* proposée, le noeud A sélectionne l'époque introduite ε_{B2} comme l'époque cible ($\varepsilon_{A1} <_{\varepsilon} \varepsilon_{B2}$). Mais pour pouvoir renommer son état vers l'époque ε_{B2} , il doit au préalable faire revenir son état courant de l'époque ε_{A1} à un état équivalent à l'époque ε_0 . Nous devons définir un mécanisme permettant aux noeuds d'annuler les effets d'une opération *rename* appliquée précédemment.

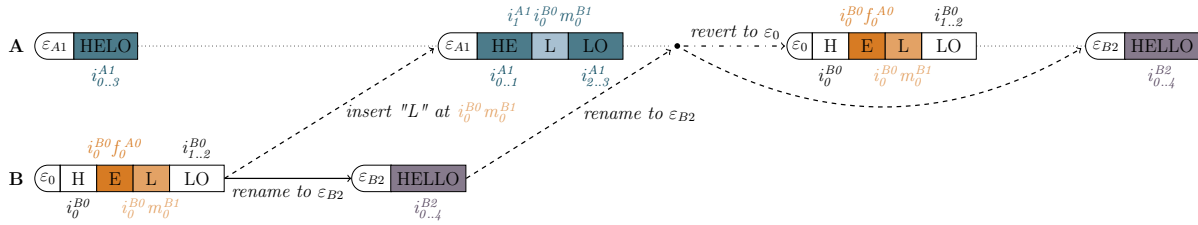


FIGURE 3.9 – Annulation d'une opération *rename* intégrée précédemment en présence d'un identifiant inséré en concurrence

C'est précisément le but de REVERTRENAMEID, qui associe les identifiants de l'époque *enfant* aux identifiants correspondant dans l'époque *parente*. Nous décrivons cette fonction dans l'Algorithme 2.

Les objectifs de REVERTRENAMEID sont les suivants : (i) Restaurer à leur ancienne valeur les identifiants générés causalement avant l'opération *rename* annulée (ii) Restaurer à leur ancienne valeur les identifiants générés de manière concurrente à l'opération *rename* annulée (iii) Assigner de nouveaux identifiants respectant l'ordre souhaité aux éléments qui ont été insérés causalement après l'opération *rename* annulée.

Le cas (i) est le plus trivial. Pour retrouver la valeur de *id* à partir de *newId*¹¹, REVERTRENAMEID utilise simplement la valeur de offset de *newId*. En effet, cette valeur correspond à l'index de *id* dans l'*ancien état* (c.-à-d. $renamedIds[offset] = id$). Par exemple, dans la Figure 3.9, l'identifiant i_0^{A1} a pour offset 0, REVERTRENAMEID renvoie donc $renamedIds[0] = i_0^{B0}$.

Les cas (ii) et (iii) sont gérés en utilisant les stratégies suivantes. Le motif générique pour l'identifiant *newId* est de la forme *newPredId tail*. Deux invariants sont associés à ce motif. D'après la Propriété 3.2, nous avons :

$$newId \in]newPredId, newSuccId[$$

11. Nous appelons *newX* les identifiants dans l'époque résultant de l'application d'une opération *rename*, tandis que *X* décrit leur équivalent à l'époque initiale.

Algorithme 2 Fonctions principales pour annuler le renommage appliqué précédemment à un identifiant

```

function REVERTRENAMEID(id, renamedIds, nId, nSeq)
  length ← renamedIds.length
  firstId ← renamedIds[0]
  lastId ← renamedIds[length - 1]
  pos ← position(firstId)

  newFirstId ← new Id(pos, nId, nSeq, 0)
  newLastId ← new Id(pos, nId, nSeq, length - 1)

  if id < newFirstId then
    return revRenIdLessThanNewFirstId(id, firstId, newFirstId)
  else if isRenamedId(id, pos, nId, nSeq, length) then
    index ← getFirstOffset(id)
    return renamedIds[index]
  else if newLastId < id then
    return revRenIdGreaterThanNewLastId(id, lastId)
  else
    index ← getFirstOffset(id)
    return revRenIdfromPredId(id, renamedIds, index)
  end if
end function

function REVRENIDFROMPREDID(id, renamedIds, index)
  predId ← renamedIds[index]
  succId ← renamedIds[index + 1]
  tail ← getTail(id, 1)

  if tail < predId then
    ▷ id has been inserted causally after the rename op
    return concat(predId, MIN_TUPLE, tail)
  else if succId < tail then
    ▷ id has been inserted causally after the rename op
    offset ← getLastOffset(succId) - 1
    predOfSuccId ← createIdFromBase(succId, offset)
    return concat(predOfSuccId, MAX_TUPLE, tail)
  else
    return tail
  end if
end function

```

et nous devons obtenir :

$$id \in]predId, succId[$$

Le premier sous-cas se produit quand nous avons $tail \in]predId, succId[$. Dans ce cas, $newId$ peut résulter d'une opération *insert* concurrent à l'opération *rename* (c.-à-d. le cas (ii)). Nous avons alors :

$$newId \in]newPredId\ predId, newPredId\ succId[$$

Dans cette situation, $newId$ a été obtenu en utilisant `RENIDFROMPREDID` et nous avons $id = tail$. Nous observons qu'en renvoyant $tail$, `REVERTRENAMEID` valident les deux

contraintes, c.-à-d. préserver l'ordre souhaité et restaurer à sa valeur initiale l'identifiant. Pour illustrer ce cas, considérons l'identifiant $i_1^{A1} i_0^{B0} m_0^{B1}$ dans Figure 3.9. Pour cet identifiant, nous avons :

- $newPredId = i_1^{A1}$, donc $predId = i_0^{B0} f_0^{A0}$ d'après le cas (i)
- $newSuccId = i_2^{A1}$, donc $succId = i_1^{B0}$ d'après le cas (i)

Nous avons donc bien :

$$i_1^{A1} i_0^{B0} m_0^{B1} \in]i_1^{A1} i_0^{B0} f_0^{A0}, i_1^{A1} i_1^{B0}]$$

et $tail = i_0^{B0} m_0^{B1}$. Renvoyer cette valeur nous permet ainsi de conserver l'ordre entre les identifiants puisque :

$$i_0^{B0} f_0^{A0} <_{id} i_0^{B0} m_0^{B1} <_{id} i_1^{B0}$$

Le second sous-cas correspond au cas où nous avons $tail < predId$. $newId$ ne peut avoir été inséré que causalement après l'opération *rename* (c.-à-d. le cas (iii)). Nous avons alors :

$$newId \in]newPredId, newPredId \ predId[$$

Puisque $newId$ a été inséré causalement après l'opération *rename*, il n'existe pas de contrainte sur la valeur à retourner autre que la Propriété 3.2. Pour gérer ce cas, nous introduisons deux nouveaux tuples exclusifs au mécanisme de renommage : *MIN_TUPLE* et *MAX_TUPLE*, notés respectivement \perp et \top . Ils sont respectivement le tuple minimal et maximal utilisables pour générer des identifiants. En utilisant *MIN_TUPLE*, REVERTRENAMEID est capable de renvoyer une valeur pour id adaptée à l'ordre souhaité (avec $id = predId \perp tail$). Nous justifions ce comportement à l'aide de la Figure 3.10.

Dans la Figure 3.10, les noeuds C et D répliquent une même séquence contenant les éléments "WOD". Dans la Figure 3.10a, le noeud C commence par renommer son état. En concurrence, le noeud D insère l'élément "L" entre les éléments "O" et "D". L'opération *insert* correspondante est livrée au noeud C, qui l'intègre en suivant le comportement défini en sous-section 3.2.2. Le noeud C procède ensuite à l'insertion de l'élément "R" entre les éléments "O" et "L". Cette insertion dépend donc causalement de l'opération *rename* effectuée précédemment par C. En parallèle, le noeud D effectue un renommage de son état. Cette opération *rename* est donc concurrente à l'opération *rename* générée précédemment par C.

Dans la Figure 3.10b, l'opération *rename* de D est livrée au noeud C. L'époque introduite par cette opération étant prioritaire par rapport à l'époque actuelle de C ($\varepsilon_{C1} <_\varepsilon \varepsilon_{D3}$), le noeud C procède à l'annulation de son opération *rename*.

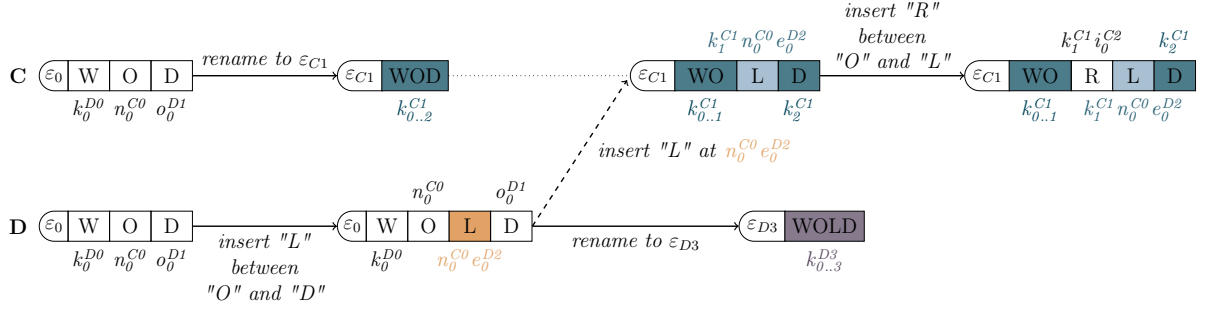
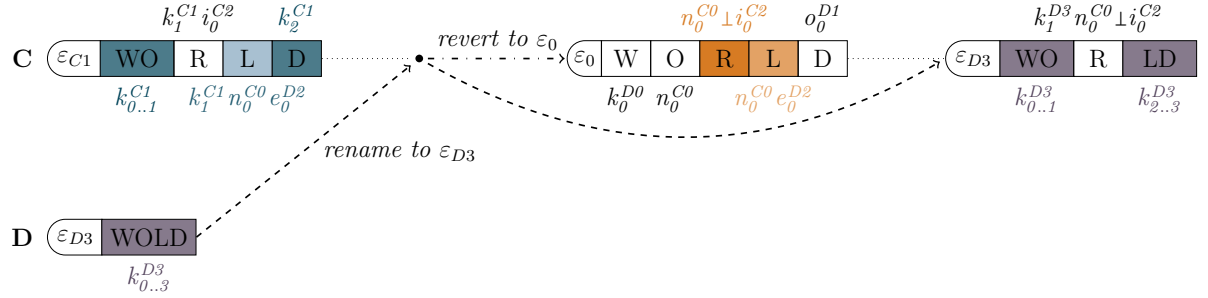
L'identifiant qui nous intéresse ici est l'identifiant inséré causalement après l'opération *rename* annulée : $k_1^{C1} i_0^{C2}$. Cet identifiant est compris entre les identifiants suivants :

$$k_1^{C1} <_{id} k_1^{C1} i_0^{C2} <_{id} k_1^{C1} n_0^{C0} e_0^{D2}$$

D'après les règles présentées précédemment :

- k_1^{C1} est transformé en n_0^{C0} (cas (i))
- $k_1^{C1} n_0^{C0} e_0^{D2}$ est transformé en $n_0^{C0} e_0^{D2}$ (cas (ii))

3.3. Gestion des opérations rename concurrentes

(a) Génération d'une opération *insert* dépendante causalement d'une opération *rename*(b) Annulation de l'opération *rename* précédente au profit d'une opération *rename* concurrenteFIGURE 3.10 – Annulation d'une opération *rename* intégrée précédemment en présence d'un identifiant inséré causalement après

Nous devons générer un identifiant id à partir de $k_1^{C1} i_0^{C2}$ tel que :

$$n_0^{C0} <_{id} id <_{id} n_0^{C0} e_0^{D2}$$

Utiliser $predId(n_0^{C0})$ en tant que préfixe de id nous permet de garantir que $n_0^{C0} <_{id} id$. Cependant, appliquer la même stratégie que pour le cas (ii) pour générer id transgresserait la Propriété 3.2. En effet, nous obtiendrions $id = n_0^{C0} i_0^{C2}$, or $n_0^{C0} i_0^{C2} \not<_{id} n_0^{C0} e_0^{D2}$.

Ainsi, nous devons choisir un autre préfixe dans cette situation, notamment pour garantir que l'identifiant résultant sera plus petit que les identifiants suivants. C'est pour cela que nous introduisons MIN_TUPLE . En concaténant $predId$ et le tuple minimal, nous obtenons un préfixe nous permettant à la fois de garantir que $n_0^{C0} <_{id} id$ et que $id <_{id} n_0^{C0} e_0^{D2}$. Nous obtenons donc $id = n_0^{C0} \perp i_0^{C2}$, ce qui respecte la Propriété 3.2.

Finalement, le dernier sous-cas est le pendant du sous-cas précédent et se produit lorsque nous avons $succId < tail$. Nous avons alors :

$$newId \in]newPredId\ succId, newSuccId[$$

La stratégie pour gérer ce cas est similaire et consiste à ajouter un préfixe pour créer l'ordre souhaité. Pour générer ce préfixe, `REVERTRENAMEID` utilise $predOfSuccId$ et MAX_TUPLE . $predOfSuccId$ est obtenu en décrémentant le dernier offset de $succId$. Ainsi, pour préserver l'ordre souhaité, `REVERTRENAMEID` renvoie id avec $id = predOfSuccId \top tail$.

Comme pour l’Algorithme 1, l’Algorithme 2 ne présente seulement que le cas principal de REVERTRENAMEID. Il s’agit du cas où l’identifiant à restaurer appartient à l’intervalle des identifiants renommés $newFirstId \leq_{id} id \leq_{id} newLastId$). Les fonctions pour gérer les cas restants sont présentées dans l’Annexe C.

Notons que RENAMEID et REVERTRENAMEID ne sont pas des fonctions réciproques. REVERTRENAMEID restaure à leur valeur initiale les identifiants insérés causalement avant ou de manière concurrente à l’opération *rename*. Par contre, RENAMEID ne fait pas de même pour les identifiants insérés causalement après l’opération *rename*. Rejouer une opération *rename* précédemment annulée altère donc ces identifiants. Cette modification peut entraîner une divergence entre les noeuds, puis qu’un même élément sera désigné par des identifiants différents.

Ce problème est toutefois évité dans notre système grâce à la relation *priority* utilisée. Puisque la relation *priority* est définie en utilisant l’ordre lexicographique sur le chemin des époques dans l’*arbre des époques*, les noeuds se déplacent seulement vers l’époque la plus à droite de l’*arbre des époques* lorsqu’ils changent d’époque. Les noeuds évitent donc d’aller et revenir entre deux mêmes époques, et donc d’annuler et rejouer les opérations *rename* correspondantes.

3.3.4 Processus d’intégration d’une opération

Le processus d’intégration d’une opération distante distingue deux cas différents : (i) le cas de figure où l’opération reçue est une opération *insert* ou *remove* (ii) le cas de figure où l’opération reçue est une opération *rename*.

Intégration d’une opération *insert* ou *remove* distante

Dans l’Algorithme 3, nous présentons l’algorithme d’intégration d’une opération *insert* distante dans RenamableLogootSplit.

Cet algorithme se décompose en de multiples étapes. Afin d’illustrer chacune d’entre elles, nous utilisons l’exemple représenté par la Figure 3.11.

Dans la Figure 3.11a, deux noeuds A et B éditent une séquence répliquée via RenamableLogootSplit. Initialement, les deux noeuds possèdent des répliques identiques. Le noeud A commence par effectuer une opération *rename*. Il génère alors l’état équivalent à son état précédent, à la nouvelle époque ε_{A2} . Puis il effectue une opération *insert*, insérant un nouvel élément "X" entre les éléments "B" et "C". L’identifiant $i_1^{A2}f_0^{A3}$ est attribué à ce nouvel élément. Chacune des opérations du noeud A est diffusée sur le réseau.

De son côté, le noeud B génère en concurrence sa propre opération *rename* sur l’état initial. Il obtient alors un état équivalent, à l’époque ε_{B2} . Il reçoit ensuite l’opération *rename* du noeud A, qu’il intègre. Puisque $\varepsilon_{A2} <_{\varepsilon} \varepsilon_{B2}$, le noeud B ne modifie pas son époque courante (ε_{B2}). Le noeud B obtient toutefois l’*arbre des époques* représenté dans la Figure 3.11b.

Puis le noeud B reçoit l’opération *insert* de l’élément "X" à la position $i_1^{A2}f_0^{A3}$. C’est le traitement de cette opération que nous allons détailler ici.

Tout d’abord, le noeud B compare l’époque de l’opération avec l’époque courante de la séquence. Si les deux époques correspondaient, le noeud B pourrait intégrer l’opération

Algorithme 3 Algorithme d'intégration d'une opération *insert* distante

```

function INSREMOTE(seq, epochTree, currentEpoch, insOp)
  if currentEpoch = opEpoch then
    insert(seq, getIdBegin(insertOp), getContent(insertOp))
  else
5:    insertedIdInterval ← getInsertedIdInterval(insOp)
    ids ← expand(insertedIdInterval)

    opEpoch ← getEpoch(insOp)
    {epochsToRevert, epochsToApply} ← getPathBetweenEpochs(epochTree, opEpoch, currentE-
    poch)
10:
    for epoch in epochsToRevert do
      renamedIds ← getRenamedIds(epochTree, epoch)
      nId ← getNodeId(epochTree, epoch)
      nSeq ← getNodeSeq(epochTree, epoch)
15:      revertRenameIdpartial ← papply(revertRenameId, renamedIds, nId, nSeq)
      ids ← map(ids, revertRenameIdpartial)
    end for

    for epoch in epochsToApply do
20:      renamedIds ← getRenamedIds(epochTree, epoch)
      nId ← getNodeId(epochTree, epoch)
      nSeq ← getNodeSeq(epochTree, epoch)
      renameIdpartial ← papply(renameId, renamedIds, nId, nSeq)
      ids ← map(ids, renameIdpartial)
25:    end for

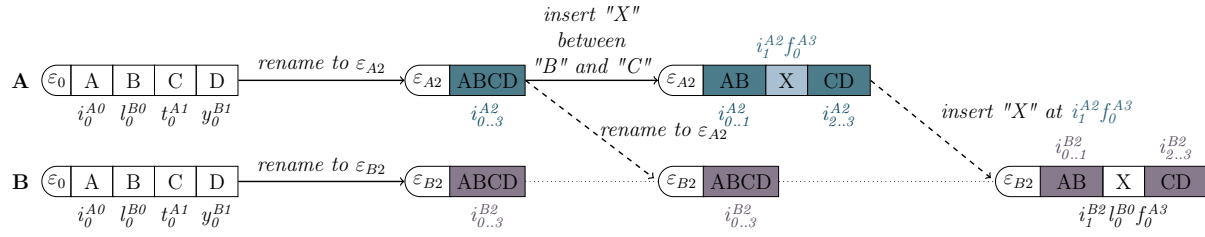
    content ← getContent(insOp)
    newIdIntervals ← aggregate(ids)
    insertOps ← generateInsertOps(newIdIntervals, content)
30:    for insertOp in insertOps do
      insert(seq, getIdBegin(insertOp), getContent(insertOp))
    end for
  end if
end function

```

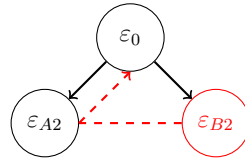
directement en utilisant l'algorithme de LogootSplit dénommé ici INSERT. Mais dans le cas présent, l'époque de l'opération (ε_{A2}) est différente de l'époque courante (ε_{B2}). Il lui est donc nécessaire de transformer l'opération avant de pouvoir l'appliquer.

Pour cela, le noeud doit identifier les transformations à appliquer à l'opération. Pour ce faire, le noeud calcule le chemin entre l'époque de l'opération et l'époque courante à l'aide de la fonction GETPATHBETWEENEPOCHS (ligne 9).

La fonction GETPATHBETWEENEPOCHS applique l'algorithme suivant : (i) elle calcule le chemin entre l'époque de l'opération et la racine de l'*arbre des époques* ($[\varepsilon_{A2}, \varepsilon_0]$) (ii) elle calcule le chemin entre l'époque courante et la racine de l'*arbre des époques* ($[\varepsilon_{B2}, \varepsilon_0]$) (iii) elle détermine la première intersection entre ces deux chemins (ε_0). Cette époque correspond au Plus Petit Ancêtre Commun (PPAC) entre l'époque de l'opération et l'époque courante. (iv) elle tronque les deux chemins au niveau du PPAC ($[\varepsilon_{A2}]$ et $[\varepsilon_{B2}]$) (v) elle inverse l'ordre des époques du chemin entre l'époque courante et la racine ($[\varepsilon_{B2}]$)



(a) Exécution nécessitant l'intégration d'une opération *insert* provenant d'une époque concurrente



(b) Arbre des époques de B à la réception de l'opération *insert*

FIGURE 3.11 – Intégration d'une opération *insert* distante

(vi) elle retourne les deux chemins obtenus ($([\varepsilon_{A2}], [\varepsilon_{B2}])$).

Le chemin entre l'époque de l'opération et l'époque PPAC ($[\varepsilon_{A2}]$) correspond aux renommages dont les effets doivent être retirés de l'opération. Pour cela, le noeud récupère les informations de chaque renommage via l'*arbre des époques* (lignes 12-14). Puis il applique REVERTRENAMEID sur chaque identifiant de l'opération (ligne 16). Le noeud procède ensuite de manière similaire pour les époques appartenant au chemin entre l'époque PPAC et l'époque courante ($[\varepsilon_{B2}]$), qui correspondent aux renommages dont les effets doivent être intégrés à l'opération (lignes 19-25).

À ce stade, le noeud obtient la liste des identifiants à insérer à l'époque courante. Il peut alors réutiliser la fonction INSERT pour les intégrer à son état. Pour minimiser le nombre de parcours de la séquence, le noeud agrège les identifiants en intervalles d'identifiants au préalable à l'aide de la fonction AGGREGATE (ligne 28). Cette fonction regroupe simplement les identifiants contigus en intervalles d'identifiants et retourne la liste des intervalles obtenus.

À partir des intervalles d'identifiants obtenus et du contenu initial de l'opération *insert*, le noeud régénère une liste d'opérations *insert*. Ces opérations sont ensuite successivement intégrées à la séquence.

L'algorithme d'intégration d'une opération *remove* distante est très similaire à l'algorithme d'intégration d'une opération *insert* que nous venons de présenter. Seules les lignes permettant de récupérer les identifiants supprimés (5), de générer l'opération *remove* transformée (29) et de l'appliquer (3 et 31) diffèrent.

Intégration d'une opération *rename* distante

L'autre cas de figure que RenamableLogootSplit doit gérer est l'intégration d'une opération *rename* distante. Pour cela, RenamableLogootSplit repose sur l'algorithme présenté dans l'Algorithme 4.

Algorithme 4 Algorithme d'intégration d'une opération *rename* distante

```

function RENREMOTE(seq, epochTree, currentEpoch, renOp)
  opEpoch  $\leftarrow$  getEpoch(renOp)
  renamedIds  $\leftarrow$  getRenamedIds(renOp)
  introducedEpoch  $\leftarrow$  getIntroducedepoch(renOp)
5:   newEpochTree  $\leftarrow$  addEpoch(epochTree, introducedEpoch, opEpoch, renamedIds)

  if introducedEpoch  $<_{\varepsilon}$  currentEpoch then
    return (seq, newEpochTree, currentEpoch)
10:  else
    idIntervals  $\leftarrow$  getIdIntervals(seq)
    ids  $\leftarrow$  flatMap(idIntervals, expand)

    (epochsToRevert, epochsToApply)  $\leftarrow$  getPathBetweenEpochs(newEpochTree, currentEpoch,
    introducedEpoch)
15:    for epoch in epochsToRevert do
      renamedIds  $\leftarrow$  getRenamedIds(newEpochTree, epoch)
      nId  $\leftarrow$  getNodeId(newEpochTree, epoch)
      nSeq  $\leftarrow$  getNodeSeq(newEpochTree, epoch)
20:      revertRenameIdpartial  $\leftarrow$  papply(revertRenameId, renamedIds, nId, nSeq)
      ids  $\leftarrow$  map(ids, revertRenameIdpartial)
    end for

    for epoch in epochsToApply do
25:      renamedIds  $\leftarrow$  getRenamedIds(newEpochTree, epoch)
      nId  $\leftarrow$  getNodeId(newEpochTree, epoch)
      nSeq  $\leftarrow$  getNodeSeq(newEpochTree, epoch)
      renameIdpartial  $\leftarrow$  papply(renameId, renamedIds, nId, nSeq)
      ids  $\leftarrow$  map(ids, renameIdpartial)
30:    end for

    nId  $\leftarrow$  getNodeId(seq)
    nSeq  $\leftarrow$  getNodeSeq(seq)
    newIdIntervals  $\leftarrow$  aggregate(ids)
35:    content  $\leftarrow$  getContent(seq)
    blocks  $\leftarrow$  generateBlocks(newIdIntervals, content)
    newSeq  $\leftarrow$  new LogootSplit(nId, nSeq, blocks)

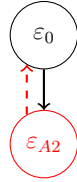
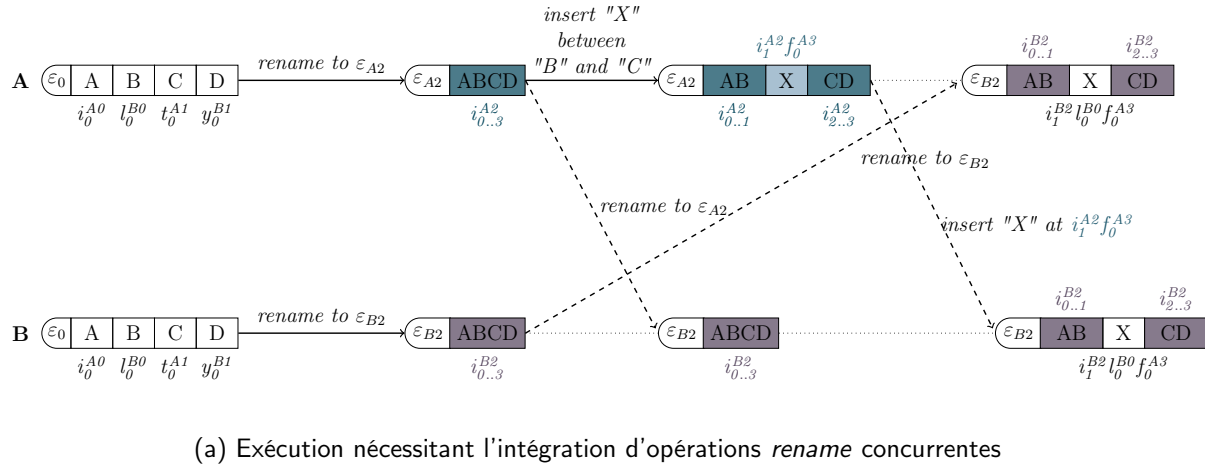
    return (newSeq, newEpochTree, introducedEpoch)
40:  end if
end function

```

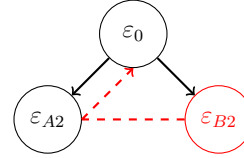
Comme précédemment, nous utilisons l'exemple illustré dans la Figure 3.12 pour présenter le fonctionnement de cet algorithme.

La Figure 3.12 reprend le scénario décrit précédemment dans la Figure 3.11. Elle complète ce dernier en faisant apparaître la réception de l'opération *rename* vers l'époque ε_{B2} par le noeud A. C'est sur ce point que nous allons nous focaliser ici.

À la réception de l'opération *rename* vers l'époque ε_{B2} , le noeud A utilise RENREMOTE pour intégrer cette opération. Tout d'abord, le noeud A ajoute l'époque ε_{B2} et les



(b) Arbre des époques de A avant la réception de l'opération *rename* vers l'époque ε_{B2}



(c) Arbre des époques de A après la réception de l'opération *rename* vers l'époque ε_{B2}

FIGURE 3.12 – Intégration d'une opération *rename* distante

métadonnées associées (ancien état, auteur de l'opération *rename*, numéro de séquence de l'auteur de l'opération *rename*) à son propre arbre des époques (ligne 6).

Le noeud compare ensuite l'époque introduite (ε_{B2}) à son époque courante (ε_{A2}) en utilisant la relation $<_{\varepsilon}$. Si l'époque introduite était plus petite que l'époque courante, aucun traitement supplémentaire ne serait nécessaire. RENREMOTE se contenterait de renvoyer comme résultats la séquence et l'époque courante, inchangées, et le nouvel *arbre des époques* (ligne 9).

Dans le cas présent, nous avons $\varepsilon_{A2} <_{\varepsilon} \varepsilon_{B2}$. ε_{B2} devient donc la nouvelle époque courante. Le noeud A procède au renommage de son état vers cette nouvelle époque.

Pour cela, le noeud récupère l'ensemble des identifiants formant son état courant (lignes 11-12). Puis, comme dans INSREMOTE, le noeud récupère le chemin entre son époque courante et l'époque cible à l'aide de GETPATHBETWEENEPOCHS puis renomme chaque identifiant à travers les différents époques (lignes 16-30).

Le noeud obtient alors la liste des identifiants courant, à la nouvelle époque cible. Il ne lui reste plus qu'à construire une nouvelle séquence à partir de ces identifiants. Pour cela, le noeud régénère des blocs à partir des intervalles d'identifiants obtenus et du contenu de la séquence courante. Le noeud utilise ensuite ces données pour instancier une nouvelle séquence équivalente à l'époque cible (ligne 37). Finalement, RENREMOTE renvoie cette nouvelle séquence, la nouvelle époque courante ainsi que le nouvel *arbre des époques*.

3.3.5 Règles de récupération de la mémoire des états précédents

Les noeuds stockent les époques et les *anciens états* correspondant pour transformer les identifiants d'une époque à l'autre. Au fur et à mesure que le système progresse, certaines époques et métadonnées associées deviennent obsolètes puisque plus aucune opération ne peut être émise depuis ces époques. Les noeuds peuvent alors supprimer ces époques. Dans cette section, nous présentons un mécanisme permettant aux noeuds de déterminer les époques obsolètes.

Pour proposer un tel mécanisme, nous nous reposons sur la notion de *stabilité causale des opérations* [29]. Une opération est causalement stable une fois qu'elle a été livrée à tous les noeuds. Dans le contexte de l'opération *rename*, cela implique que tous les noeuds ont progressé à l'époque introduite par cette opération ou à une époque plus grande d'après la relation *priority*. À partir de ce constat, nous définissons les *potentielles époques courantes* :

Définition 19 (Potentielles époques courantes). L'ensemble des époques auxquelles les noeuds peuvent se trouver actuellement et à partir desquelles ils peuvent émettre des opérations, du point de vue du noeud courant. Il s'agit d'un sous-ensemble de l'ensemble des époques, composé de l'époque maximale introduite par une opération *rename* causalement stable et de toutes les époques plus grande que cette dernière d'après la relation *priority*.

Pour traiter les prochaines opérations, les noeuds doivent maintenir les chemins entre toutes les époques de l'ensemble des *potentielles époques courantes*. Nous appelons *époques requises* l'ensemble des époques correspondant.

Définition 20 (Époques requises). L'ensemble des époques qu'un noeud doit conserver pour traiter les potentielles prochaines opérations. Il s'agit de l'ensemble des époques qui forment les chemins entre chaque époque appartenant à l'ensemble des *potentielles époques courantes* et leur Plus Petit Ancêtre Commun (PPAC).

Il s'ensuit que toute époque qui n'appartient pas à l'ensemble des *époques requises* peut être retirée par les noeuds. La Figure 3.13 illustre un cas d'utilisation du mécanisme de récupération de mémoire proposé.

Dans la Figure 3.13a, nous représentons une exécution au cours de laquelle deux noeuds A et B génèrent respectivement plusieurs opérations *rename*. Dans la Figure 3.13b, nous représentons les *arbres des époques* respectifs de chaque noeud. Les époques introduites par des opérations *rename* causalement stables sont représentées en utilisant des doubles cercles. L'ensemble des *potentielles époques courantes* est montré sous la forme d'un rectangle noir tireté, tandis que l'ensemble des *époques requises* est représenté par un rectangle vert pointillé.

Le noeud A génère tout d'abord une opération *rename* vers ε_{A1} et ensuite une opération *rename* vers ε_{A8} . Il reçoit ensuite une opération *rename* du noeud B qui introduit ε_{B2} . Puisque ε_{B2} est plus grand que son époque courante actuelle ($\varepsilon_{e0\varepsilon_{A1}\varepsilon_{A8}} < \varepsilon_{e0\varepsilon_{B2}}$), le noeud A la sélectionne comme sa nouvelle époque cible et procède au renommage de son état en conséquence. Finalement, le noeud A génère une troisième opération *rename* vers ε_{A9} .

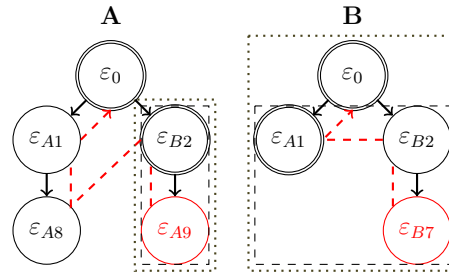
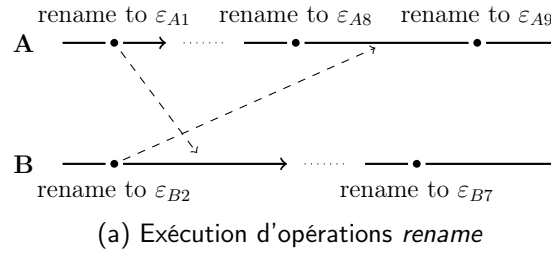


FIGURE 3.13 – Suppression des époques obsolètes et récupération de la mémoire des anciens états associés

De manière concurrente, le noeud B génère l'opération *rename* vers ε_{B2} . Il reçoit ensuite l'opération *rename* vers ε_{A1} du noeud A. Cependant, le noeud B conserve ε_{B2} comme époque courante (puisque $\varepsilon_{e0}\varepsilon_{A1} < \varepsilon_{e0}\varepsilon_{B2}$). Après, le noeud B génère une autre opération *rename* vers ε_{B7} .

À la livraison de l'opération *rename* introduisant l'époque ε_{B2} au noeud A, cette opération devient causalement stable. À partir de ce point, le noeud A sait que tous les noeuds ont progressé jusqu'à cette époque ou une plus grande d'après la relation *priority*. Les époques ε_{B2} et ε_{A9} forment donc l'ensemble des *potentielles époques courantes* et les noeuds peuvent seulement émettre des opérations depuis ces époques ou une de leur descendante encore inconnue. Le noeud A procède ensuite au calcul de l'ensemble des *époques requises*. Pour ce faire, il détermine le PPAC des *potentielles époques courantes* : ε_{B2} . Il génère ensuite l'ensemble des *époques requises* en ajoutant toutes les époques formant les chemins entre ε_{B2} et les *potentielles époques courantes*. Les époques ε_{B2} et ε_{A9} forment donc l'ensemble des *époques requises*. Le noeud A déduit que les époques ε_0 , ε_{A1} et ε_{A8} peuvent être supprimées de manière sûre.

À l'inverse, la livraison de l'opération *rename* vers ε_{A1} au noeud B ne lui permet pas de supprimer la moindre métadonnée. À partir de ses connaissances, le noeud B calcule que ε_{A1} , ε_{B2} et ε_{B7} forment l'ensemble des *potentielles époques courantes*. De cette information, le noeud B détermine que ces époques et leur PPAC forment l'ensemble des *époques requises*. Toute époque connue appartient donc à l'ensemble des *époques requises*, empêchant leur suppression.

À terme, une fois que le système devient inactif, les noeuds atteignent la même époque et l'opération *rename* correspondante devient causalement stable. Les noeuds peuvent alors supprimer toutes les autres époques et métadonnées associées, supprimant ainsi le surcoût mémoire introduit par le mécanisme de renommage.

Notons que le mécanisme de récupération de mémoire peut être simplifié dans les systèmes empêchant les opérations *rename* concurrentes. Puisque les époques forment une chaîne dans de tels systèmes, la dernière époque introduite par une opération *rename* causalement stable devient le PPAC des *potentielles époques courantes*. Il s'ensuit que cette époque et ses descendantes forment l'ensemble des *époques requises*. Les noeuds n'ont donc besoin que de suivre les opérations *rename* causalement stables pour déterminer quelles époques peuvent être supprimées dans les systèmes sans opérations *rename* concurrentes.

Pour déterminer qu'une opération *rename* donnée est causalement stable, les noeuds doivent être conscients des autres et de leur avancement. Un protocole de gestion de groupe tel que [16, 17] est donc requis.

La stabilité causale peut prendre un certain temps à être atteinte. En attendant, les noeuds peuvent néanmoins décharger les anciens états sur le disque dur puisqu'ils ne sont seulement nécessaires que pour traiter les opérations concurrentes aux opérations *rename*. Nous approfondissons ce sujet dans la sous-section 3.5.2.

3.4 Validation

3.4.1 Complexité en temps des opérations

Afin d'évaluer RenamableLogootSplit, nous analysons tout d'abord la complexité en temps de ses opérations. Ces complexités dépendent de plusieurs paramètres : nombre d'identifiants et de blocs stockés au sein de la structure, taille des identifiants, structures de données utilisées...

Hypothèses

Afin d'établir les valeurs de complexité des différentes opérations, nous prenons les hypothèses suivantes vis-à-vis des paramètres. Nous supposons que le nombre n d'identifiants présents dans la séquence a tendance à croître, c.-à-d. que plus d'insertions sont effectuées que de suppressions. Nous considérons que la taille des identifiants, qui elle croît avec le nombre d'insertions mais qui est réinitialisée à chaque renommage, devient négligeable par rapport au nombre d'identifiants. Nous ne prenons donc pas en considération ce paramètre dans nos complexités et considérons que les manipulations d'identifiants (comparaison, génération) s'effectuent en temps constant. Afin de simplifier les complexités, nous considérons que les *anciens états* associés aux époques contiennent aussi n identifiants. Finalement, nous considérons que nous utilisons comme structures de données un arbre AVL pour représenter l'état interne de la séquence, des tableaux pour les *anciens états* et une table de hachage pour l'*arbre des époques*.

Complexité en temps des opérations *insert* et *remove*

À partir de ces hypothèses, nous établissons les complexités en temps des opérations. Pour chaque opération, nous distinguons deux complexités : une complexité pour l'intégration de l'opération locale, une pour l'intégration de l'opération distante.

La complexité de l'intégration de l'opération *insert* locale est inchangée par rapport à celle obtenue pour LogootSplit. Son intégration consiste toujours à déterminer entre quels identifiants se situe les nouveaux éléments insérés, à générer de nouveaux identifiants correspondants à l'ordre souhaité puis à insérer le bloc dans l'arbre AVL. D'après ANDRÉ et al. [11], nous obtenons donc une complexité de $\mathcal{O}(\log b)$ pour cette opération locale, où b représente le nombre de blocs dans la séquence.

La complexité de l'intégration de l'opération *insert* distante, elle, évolue par rapport à celle définie pour LogootSplit. Comme indiqué dans la section 3.3.4, plusieurs étapes se rajoutent au processus d'intégration de l'opération notamment dans le cas où celle-ci provient d'une autre époque que l'époque courante.

Tout d'abord, il est nécessaire d'identifier l'époque PPAC entre l'époque de l'opération et l'époque courante. L'algorithme correspondant consiste à déterminer la première intersection entre deux branches de l'*arbre des époques*. Cette étape peut être effectuée en $\mathcal{O}(h)$, où h représente la hauteur de l'*arbre des époques*.

L'obtention de l'époque PPAC entre l'époque de l'opération et l'époque courante permet de déterminer les k renommages dont les effets doivent être retirés de l'opération et les l renommages dont les effets doivent être intégrés à l'opération. Le noeud intégrant l'opération procède ainsi aux k inversions de renommages successives puis aux l application de renommages, et ce pour tous les s identifiants insérés par l'opération.

Pour retirer les effets des renommages à inverser, le noeud intégrant l'opération utilise REVERTRENAMEID. Cet algorithme retourne pour un identifiant donné un nouvel identifiant correspondant à l'époque précédente. Pour cela, REVERTRENAMEID utilise le prédécesseur et le successeur de l'identifiant donné dans l'*ancien état* renommé. Pour retrouver ces deux identifiants au sein de l'*ancien état*, REVERTRENAMEID utilise l'offset du premier tuple de l'identifiant donné. Par définition, cet élément correspond à l'index du prédécesseur de l'identifiant donné dans l'*ancien état*. Aucun parcours de l'*ancien état* n'est nécessaire. Le reste de REVERTRENAMEID consistant en des comparaisons et manipulations d'identifiants, nous obtenons que REVERTRENAMEID s'effectue en $\mathcal{O}(1)$.

Pour inclure les effets des renommages à appliquer, le noeud utilise ensuite RENAMEID. De manière similaire à REVERTRENAMEID, RENAMEID génère pour un identifiant donné un nouvel identifiant équivalent à l'époque suivante en se basant sur son prédécesseur. Cependant, il est nécessaire ici de faire une recherche pour déterminer le prédécesseur de l'identifiant donné dans l'*ancien état*. L'*ancien état* étant un tableau trié d'identifiants, il est possible de procéder à une recherche dichotomique. Cela permet de trouver le prédécesseur en $\mathcal{O}(\log n)$, où n correspond ici au nombre d'identifiants composant l'*ancien état*. Comme pour REVERTRENAMEID, les instructions restantes consistent en des comparaisons et manipulations d'identifiants. La complexité de RENAMEID est donc de $\mathcal{O}(\log n)$.

Une fois les identifiants introduits par l'opération *insert* renommés pour l'époque courante, il ne reste plus qu'à les insérer dans la séquence. Cette étape se réalise en $\mathcal{O}(\log b)$ pour chaque identifiant, le temps nécessaire pour trouver son emplacement dans l'arbre AVL.

Ainsi, en reprenant l'ensemble des étapes composant l'intégration de l'opération *insert* distante, nous obtenons la complexité suivante : $\mathcal{O}(h + s(k + l \cdot \log n + \log b))$.

Le procédé de l'intégration de l'opération *remove* étant similaire à celui de l'opération *insert*, aussi bien en local qu'en distant, nous obtenons les mêmes complexités en temps.

Complexité en temps de l'opération *rename*

Étudions à présent la complexité en temps de l'opération *rename*.

L'opération *rename* locale se décompose en 2 étapes : (i) La génération de l'*ancien état* à intégrer au message de l'opération (cf. Définition 14) (ii) Le remplacement de la séquence courante par une séquence équivalente, renommée. La première étape consiste à parcourir et à linéariser la séquence actuelle pour en extraire les intervalles d'identifiants. Elle s'effectue donc en $\mathcal{O}(b)$. La seconde consiste à instancier une nouvelle séquence vide, et à y insérer un bloc qui associe le contenu actuel de la séquence à l'intervalle d'identifiants $pos_{0..n-1}^{nodeId\ nodeSeq}$, avec pos la position du premier tuple du premier id de l'état, $nodeId$ et $nodeSeq$ l'identifiant et le numéro de séquence actuel du noeud et n la taille du contenu. Cette seconde étape s'effectue en $\mathcal{O}(1)$. L'opération *rename* locale a donc une complexité de $\mathcal{O}(b)$.

L'intégration de l'opération *rename* se décompose en les étapes suivantes : (i) L'insertion de la nouvelle époque et de l'*ancien état* associé dans l'*arbre des époques* (ii) La récupération des n identifiants formant l'état courant (iii) Le calcul de l'époque PPAC entre l'époque courante et l'époque cible (iv) L'identification des k opérations *rename* à inverser et des l opérations *rename* à jouer (v) Le renommage de chacun des identifiants à l'aide de REVERTRENAMEID et RENAMEID (vi) L'insertion de chacun des identifiants renommés dans une nouvelle séquence L'*arbre des époques* étant représenté à l'aide d'une table de hachage, la première étape s'effectue en $\mathcal{O}(1)$. La seconde étape nécessite elle de parcourir l'arbre AVL et de convertir chaque intervalle d'identifiants en liste d'identifiants, ce qui nécessite $\mathcal{O}(n)$ instructions.

Les étapes (iii) à (vi) peuvent être effectuées en réutilisant pour chaque identifiant l'algorithme pour l'intégration d'opérations *insert* distantes analysé précédemment. Ces étapes s'effectuent donc en $\mathcal{O}(n(k + l \cdot \log n + \log b))$.

Nous obtenons donc une complexité en temps de $\mathcal{O}(h + n(k + l \cdot \log n + \log b))$ pour l'intégration de l'opération *rename* distante.

Nous pouvons néanmoins améliorer ce premier résultat. Notamment, nous pouvons tirer parti des faits suivants : (i) Le fonctionnement de RENAMEID repose sur l'utilisation de l'identifiant prédecesseur comme préfixe (ii) Les identifiants de l'état courant et de l'*ancien état* forment tous deux des listes triées. Ainsi, plutôt que d'effectuer une recherche dichotomique sur l'*ancien état* pour trouver le prédecesseur de l'identifiant à renommer, nous pouvons parcourir les deux listes en parallèle. Ceci nous permet de renommer l'intégralité des identifiants en un seul parcours de l'état courant et de l'*ancien état*, c.-à-d. en $\mathcal{O}(n)$ instructions. Ensuite, plutôt que d'insérer les identifiants un à un dans la nouvelle séquence, nous pouvons recomposer au préalable les différents blocs en parcourant la liste des identifiants et en les agrégeant au fur et à mesure. Il ne reste plus qu'à constituer la nouvelle séquence à partir des blocs obtenus. Ces actions s'effectuent respectivement en $\mathcal{O}(n)$ et $\mathcal{O}(b)$ instructions.

Ainsi, ces améliorations nous permettent d'obtenir une complexité en temps en $\mathcal{O}(h + n(k + l) + b)$ pour le traitement de l'opération *rename* distante.

Récapitulatif

Nous récapitulons les complexités en temps présentées précédemment dans le Tableau 3.1.

TABLE 3.1 – Complexité en temps des différentes opérations

Type d'opération	Complexité en temps	
	Locale	Distante
<i>insert</i>	$\log b$	$h + s(k + l \cdot \log n + \log b)$
<i>remove</i>	$\log b$	$h + s(k + l \cdot \log n + \log b)$
<i>naive rename</i>	b	$h + n(k + l \cdot \log n + \log b)$
<i>rename</i>	b	$h + n(k + l) + b$

b : nombre de blocs, n : nombre d'éléments de l'état courant et des *anciens états*, h : hauteur de l'*arbre des époques*, k : nombre de renommages à inverser, l : nombre de renommages à appliquer, s : nombre d'éléments insérés/supprimés par l'opération

Complexité en temps du mécanisme de récupération de mémoire des époques

Pour compléter notre analyse théorique des performances de RenamableLogootSplit, nous proposons une analyse en complexité en temps du mécanisme présenté en sous-section 3.3.5 qui permet de supprimer les époques devenues obsolètes et de récupérer la mémoire occupée par leur *ancien état* respectif.

L'algorithme du mécanisme de récupération de la mémoire se compose des étapes suivantes. Tout d'abord, il établit le vecteur de version des opérations causalement stables. Pour cela, chaque noeud doit maintenir une matrice des vecteurs de version de tous les noeuds. L'algorithme génère le vecteur de version des opérations causalement stable en récupérant pour chaque noeud la valeur minimale qui y est associée dans la matrice des vecteurs de version. Cette étape correspond à fusionner n vecteurs de version contenant n entrées, elle s'exécute donc en $\mathcal{O}(n^2)$ instructions.

La seconde étape consiste à parcourir l'arbre des époques de manière inverse à l'ordre défini par la relation *priority*. Ce parcours s'effectue jusqu'à trouver l'époque maximale causalement stable, c.-à-d. la première époque pour laquelle l'opération *rename* associée est causalement stable. Pour chaque époque parcourue, le mécanisme de récupération de mémoire calcule et stocke son chemin jusqu'à la racine. Cette étape s'exécute donc en $\mathcal{O}(e \cdot h)$, avec e le nombre d'époques composant l'arbre des époques et h la hauteur de l'arbre.

À partir de ces chemins, le mécanisme calcule l'époque PPAC. Pour ce faire, l'algorithme calcule de manière successive la dernière intersection entre le chemin de la racine jusqu'à l'époque PPAC courante et les chemins précédemment calculés. L'époque PPAC est la dernière époque du chemin résultant. Cette étape s'exécute aussi en $\mathcal{O}(e \cdot h)$.

L'algorithme peut alors calculer l'ensemble des *époques requises*. Pour cela, il parcourt les chemins calculés au cours de la seconde étape. Pour chaque chemin, il ajoute les

époques se trouvant après l'époque PPAC à l'ensemble des *époques requises*. De nouveau, cette étape s'exécute en $\mathcal{O}(e \cdot h)$.

Après avoir déterminé l'ensemble des *époques requises*, le mécanisme peut supprimer les époques obsolètes. Il parcourt l'arbre des époques et supprime toute époque qui n'appartient pas à cet ensemble. Cette étape finale s'exécute en $\mathcal{O}(e)$.

Ainsi, nous obtenons que la complexité en temps du mécanisme de récupération de mémoire des époques est en $\mathcal{O}(n^2 + e \cdot h)$. Nous récapitulons ce résultat dans Tableau 3.2.

TABLE 3.2 – Complexité en temps du mécanisme de récupération de mémoire des époques

Étape	Temps
<i>calculer le vecteur de version des opérations causalement stables</i>	n^2
<i>calculer les chemins de la racine aux potentielles époques courantes</i>	$e \cdot h$
<i>identifier le PPAC</i>	$e \cdot h$
<i>calculer l'ensemble des époques requises</i>	$e \cdot h$
<i>supprimer les époques obsolètes</i>	e
<i>total</i>	$n^2 + e \cdot h$

n : nombre de noeuds du système, e : nombre d'époques dans l'*arbre des époques*, h : hauteur de l'*arbre des époques*

Malgré sa complexité en temps, le mécanisme de récupération de mémoire des époques devrait avoir un impact limité sur les performances de l'application. En effet, ce mécanisme n'appartient pas au chemin critique de l'application, c.-à-d. l'intégration des modifications. Il peut être déclenché occasionnellement, en tâche de fond. Nous pouvons même viser des fenêtres spécifiques pour le déclencher, e.g. pendant les périodes d'inactivité. Ainsi, nous avons pas étudié plus en détails cette partie de RenamableLogootSplit dans le cadre de cette thèse. Des améliorations de ce mécanisme doivent donc être possibles.

3.4.2 Expérimentations

Afin de valider l'approche que nous proposons, nous avons procédé à une évaluation expérimentale. Les objectifs de cette évaluation étaient de mesurer (i) le surcoût mémoire de la séquence répliquée (ii) le surcoût en calculs ajouté aux opérations *insert* et *remove* par le mécanisme de renommage (iii) le coût d'intégration des opérations *rename*.

Par le biais de simulations, nous avons généré le jeu de données utilisé par nos benchmarks. Ces simulations suivent le scénario suivant.

Scénario d'expérimentation

Le scénario reproduit la rédaction d'un article par plusieurs pairs de manière collaborative, en temps réel. La collaboration ainsi décrite se décompose en 2 phases.

Dans un premier temps, les pairs spécifient principalement le contenu de l'article. Quelques opérations *remove* sont tout même générées pour simuler des fautes de frappes. Une fois que le document atteint une taille critique (définie de manière arbitraire), les pairs passent à la seconde phase de la collaboration. Lors de cette seconde phase, les pairs

arrêtent d'ajouter du nouveau contenu mais se concentre à la place sur la reformulation et l'amélioration du contenu existant. Ceci est simulé en équilibrant le ratio entre les opérations *insert* et *remove*.

Chaque pair doit émettre un nombre donné d'opérations *insert* et *remove*. La simulation prend fin une fois que tous les pairs ont reçu toutes les opérations. Pour suivre l'évolution de l'état des pairs, nous prenons des instantanés de leur état à plusieurs points donnés de la simulation.

Implémentation des simulations

Nous avons effectué nos simulations avec les paramètres expérimentaux suivants : nous avons déployé 10 bots à l'aide de conteneurs Docker sur une même machine. Chaque conteneur correspond à un processus Node.js mono-threadé et permet de simuler un pair. Les bots sont connectés entre eux par le biais d'un réseau P2P maillé entièrement connecté. Enfin, ils partagent et éditent le document de manière collaborative en utilisant soit LogootSplit soit RenamableLogootSplit en fonction des paramètres de la session.

Toutes les 200 ± 50 ms, chaque bot génère localement une opération *insert* ou *remove* et la diffuse immédiatement aux autres noeuds. Au cours de la première phase, la probabilité d'émettre une opération *insert* (resp. *remove*) est de 80% (resp. 20%). Une fois que leur copie locale du document atteint 60k caractères (environ 15 pages), les bots basculent à la seconde phase et redéfinissent chaque probabilité à 50%. De plus, tout au long de la collaboration, les bots ont une probabilité de 5% de déplacer leur curseur à une position aléatoire dans le document après chaque opération locale.

Chaque bot doit générer 15k opérations *insert* ou *remove*, et s'arrête donc une fois qu'il a intégré les 150k opérations. Pour chaque bot, nous enregistrons un instantané de son état toutes les 10k opérations intégrées. Nous enregistrons aussi son log des opérations à l'issue de la simulation.

De plus, dans le cas de RenamableLogootSplit, 1 à 4 bots sont désignés de façon arbitraire comme des *renaming bots* en fonction de la session. Les *renaming bots* génèrent des opérations *rename* toutes les 7.5k ou toutes les 30k opérations qu'ils observent, en fonction des paramètres de la simulation. Ces opérations *rename* sont générées de manière à assurer qu'elles soient concurrentes.

Dans un but de reproductibilité, nous avons mis à disposition notre code, nos benchmarks et les résultats à l'adresse suivante : <https://github.com/coast-team/mute-bot-random/>.

3.4.3 Résultats

En utilisant les instantanés et les logs d'opérations générés, nous avons effectué plusieurs benchmarks. Ces benchmarks évaluent les performances de RenamableLogootSplit et les comparent à celles de LogootSplit. Sauf mention contraire, les benchmarks utilisent les données issues des simulations au cours desquelles les opérations *rename* étaient générées toutes les 30k opérations. Les résultats sont présentés et analysés ci-dessous.

Convergence

Nous avons tout d'abord vérifié la convergence de l'état des noeuds à l'issue des simulations. Pour chaque simulation, nous avons comparé l'état final de chaque noeud à l'aide de leur instantanés respectifs. Nous avons pu confirmer que les noeuds convergaient sans aucune autre forme de communication que les opérations, satisfaisant donc le modèle de la SEC.

Ce résultat établit un premier jalon dans la validation de la correction de Renamable-LogootSplit. Il n'est cependant qu'empirique. Des travaux supplémentaires pour prouver formellement sa correction doivent être entrepris.

Consommation mémoire

Nous avons ensuite procédé à l'évaluation de l'évolution de la consommation mémoire du document au cours des simulations, en fonction du CRDT utilisé et du nombre de *renaming bots*. Nous présentons les résultats obtenus dans la Figure 3.14.

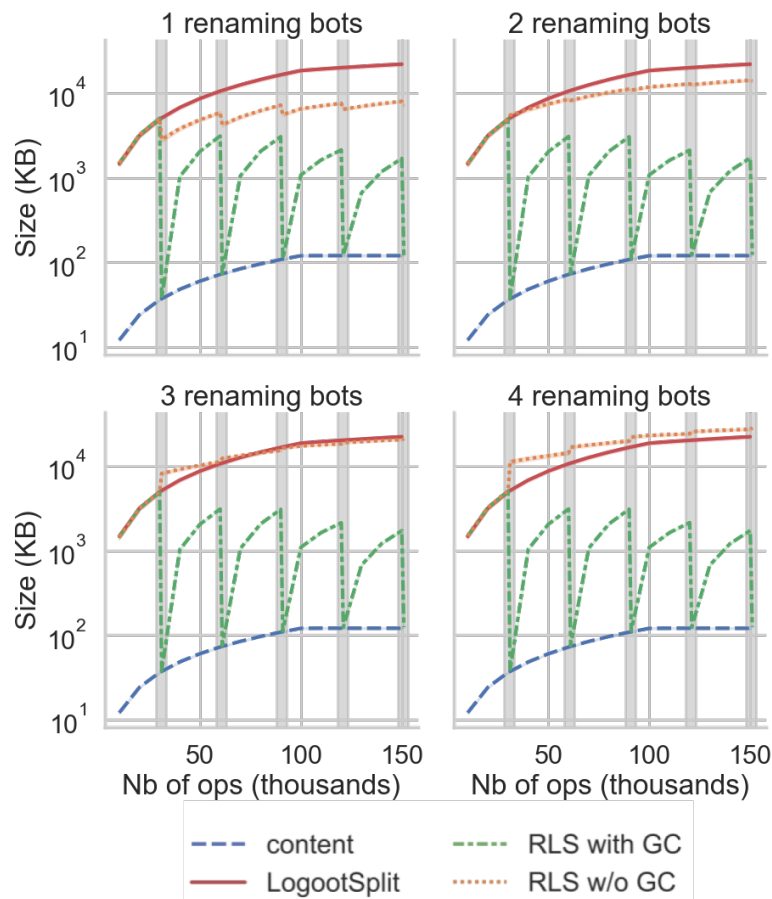


FIGURE 3.14 – Évolution de la taille du document en fonction du CRDT utilisé et du nombre de *renaming bots* dans la collaboration

Pour chaque graphique dans la Figure 3.14, nous représentons 4 données différentes.

La ligne tiretée bleue correspond à la taille du contenu du document, c.-à-d. du texte, tandis que la ligne continue rouge représente la taille complète du document LogootSplit.

La ligne verte pointillée-tiretée représente la taille du document RenamableLogootSplit dans son meilleur cas. Dans ce scénario, les noeuds considèrent que les opérations *rename* sont causalement stables dès qu'ils les reçoivent. Les noeuds peuvent alors bénéficier des effets du mécanisme de renommage tout en supprimant les métadonnées qu'il introduit : les *anciens états* et époques. Ce faisant, les noeuds peuvent minimiser de manière périodique le surcoût en métadonnées de la structure de données, indépendamment du nombre de *renaming bots* et d'opérations *rename* concurrentes générées.

La ligne pointillée orange représente la taille du document RenamableLogootSplit dans son pire cas. Dans ce scénario, les noeuds considèrent que les opérations *rename* ne deviennent jamais causalement stables. Les noeuds doivent alors conserver de façon permanente les métadonnées introduites par le mécanisme de renommage. Les performances de RenamableLogootSplit diminuent donc au fur et à mesure que le nombre de *renaming bots* et d'opérations *rename* générées augmente. Néanmoins, même dans ces conditions, nous observons que RenamableLogootSplit offre de meilleures performances que LogootSplit tant que le nombre de *renaming bots* reste faible (1 ou 2). Ce résultat s'explique par le fait que le mécanisme de renommage permet aux noeuds de supprimer les métadonnées de la structure de données utilisée en interne pour représenter la séquence (c.-à-d. l'arbre AVL).

Pour récapituler les résultats présentés, le mécanisme de renommage introduit un surcoût temporaire en métadonnées qui augmente avec chaque opération *rename*. Mais ce surcoût se résorbe à terme une fois que le système devient quiescent et que les opérations *rename* deviennent causalement stables. Dans la sous-section 3.5.2, nous détaillerons l'idée que les *anciens états* peuvent être déchargés sur le disque en attendant que la stabilité causale soit atteinte pour atténuer l'impact du surcoût temporaire en métadonnées.

Temps d'intégration des opérations standards

Nous avons ensuite comparé l'évolution du temps d'intégration des opérations standards, c.-à-d. les opérations *insert* et *remove*, sur des documents LogootSplit et RenamableLogootSplit. Puisque les deux types d'opérations partagent la même complexité en temps, nous avons seulement utilisé des opérations *insert* dans nos benchmarks. Nous faisons par contre la différence entre les mises à jours *locales* et *distantes*. Conceptuellement, les modifications locales peuvent être décomposées comme présenté dans [30] en les deux étapes suivantes : (i) la génération de l'opération correspondante (ii) l'application de l'opération correspondante sur l'état local. Cependant, pour des raisons de performances, nous avons fusionné ces deux étapes dans notre implémentation. Nous distinguons donc les résultats des modifications *locales* et des modifications *distantes* dans nos benchmarks. La Figure 3.15 présente les résultats obtenus.

Dans ces figures, les boxplots oranges correspondent aux temps d'intégration sur des documents LogootSplit, les boxplots bleues sur des documents RenamableLogootSplit. Bien que les temps d'intégration soient initialement équivalents, les temps d'intégration sur des documents RenamableLogootSplit sont ensuite réduits par rapport à ceux de LogootSplit une fois que des opérations *rename* ont été intégrées. Cette amélioration

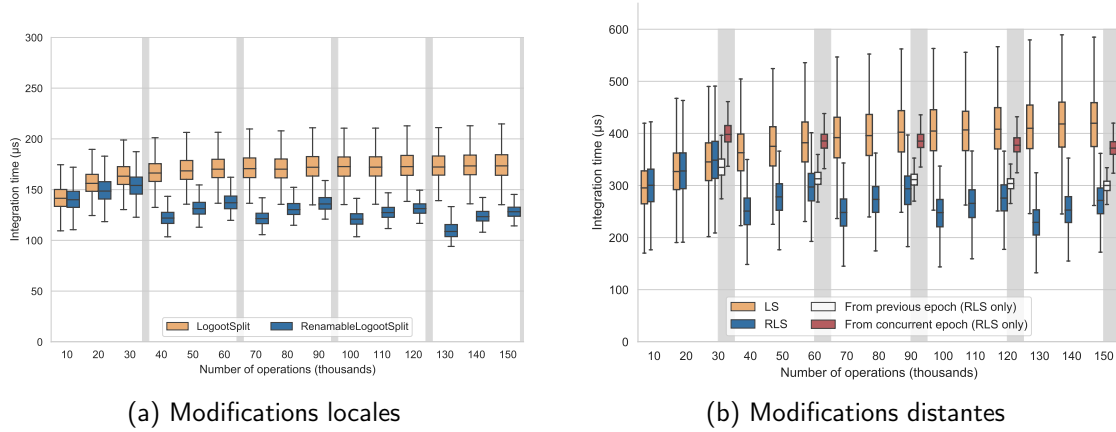


FIGURE 3.15 – Temps d'intégration des opérations standards

s'explique par le fait que l'opération *rename* optimise la représentation interne de la séquence (c.-à-d. elle réduit le nombre de blocs stockés dans l'arbre AVL).

Dans le cadre des opérations distantes, nous avons mesuré des temps d'intégration spécifiques à RenamableLogootSplit : le temps d'intégration d'opérations distantes provenant d'époques *parentes* et d'époques *soeurs*, respectivement affiché sous la forme de boxplots blanches et rouges dans la Figure 3.15b.

Les opérations distantes provenant d'époques *parentes* sont des opérations générées de manière concurrente à l'opération *rename* mais appliquées après cette dernière. Puisque l'opération doit être transformée au préalable en utilisant `RENAMEID`, nous observons un surcoût computationnel par rapport aux autres opérations. Mais ce surcoût est compensé par l'optimisation de la représentation interne de la séquence effectuée par l'opération *rename*.

Concernant les opérations provenant d'époques *soeurs*, nous observons de nouveau un surcoût puisque les noeuds doivent tout d'abord annuler les effets de l'opération *rename* concurrente en utilisant `REVERTRENAMEID`. À cause de cette étape supplémentaire, les performances de RenamableLogootSplit pour ces opérations sont comparables à celles de LogootSplit.

Pour récapituler, les fonctions de transformation ajoutent un surcoût aux temps d'intégration des opérations concurrentes aux opérations *rename*. Malgré ce surcoût, RenamableLogootSplit offre de meilleures performances que LogootSplit pour intégrer ces opérations grâce aux réductions de la taille de l'état effectuées par les opérations *rename*. Cependant, cette affirmation n'est vraie que tant que la distance entre l'époque de génération de l'opération et l'époque courante du noeud reste limitée, puisque les performances de RenamableLogootSplit dépendent linéairement de cette dernière (cf. Tableau 3.1). Néanmoins, ce surcoût ne concerne que les opérations concurrentes aux opérations *rename*. Il ne concerne pas la majorité des opérations, c.-à-d. les opérations générées entre deux séries d'opérations *rename*. Ces opérations, elles, ne souffrent d'aucun surcoût tout en bénéficiant des réductions de taille de l'état.

Temps d'intégration de l'opération de renommage

Finalement, nous avons mesuré l'évolution du temps d'intégration de l'opération *rename* en fonction du nombre d'opérations émises précédemment, c.-à-d. en fonction de la taille de l'état. Comme précédemment, nous distinguons les performances des modifications *locales* et *distantes*.

Nous rappelons que le traitement d'une opération *rename* dépend de l'ordre défini par la relation *priority* entre l'époque qu'elle introduit et l'époque courante du noeud qui intègre l'opération. Le cas des opérations *rename* distantes se décompose donc en trois catégories. Les opérations *distantes directes* désignent les opérations *rename* distantes qui introduisent une nouvelle époque *enfant* de l'époque courante du noeud. Les opérations *concurrentes introduisant une plus grande* (resp. *petite*) *époque* désignent les opérations *rename* qui introduisent une époque *soeur* de l'époque courante du noeud. D'après la relation *priority*, l'époque introduite est plus grande (resp. petite) que l'époque courante du noeud. Les résultats obtenus sont présentés dans le Tableau 3.3.

TABLE 3.3 – Temps d'intégration de l'opération *rename*

Paramètres		Temps d'intégration (ms)				
Type	Nb Ops (k)	Moyenne	Médiane	IQR	1 ^{er} Percent.	99 ^{ème} Percent.
Locale	30	41.8	38.7	5.66	37.3	71.7
	60	78.3	78.2	1.58	76.2	81.4
	90	119	119	2.17	116	124
	120	144	144	3.24	139	149
	150	158	158	3.71	153	164
Distante directe	30	481	477	15.2	454	537
	60	982	978	28.9	926	1073
	90	1491	1482	58.8	1396	1658
	120	1670	1664	41	1568	1814
	150	1694	1676	60.6	1591	1853
Cc. int. plus grande époque	30	644	644	16.6	620	683
	60	1318	1316	26.5	1263	1400
	90	1998	1994	46.6	1906	2112
	120	2240	2233	54	2144	2368
	150	2242	2234	63.5	2139	2351
Cc. int. plus petite époque	30	1.36	1.3	0.038	1.22	3.53
	60	2.82	2.69	0.476	2.43	4.85
	90	4.45	4.23	1.1	3.69	5.81
	120	5.33	5.1	1.34	4.42	8.78
	150	5.53	5.26	1.05	4.84	8.7

Le principal résultat de ces mesures est que les opérations *rename* sont particulièrement coûteuses quand comparées aux autres types d'opérations. Les opérations *rename* locales s'intègrent en centaines de millisecondes tandis que les opérations *distantes directes* et *concurrentes introduisant une plus grande époque* s'intègrent en secondes lorsque la taille du document dépasse 40k éléments. Ces résultats s'expliquent facilement par la complexité en temps de l'opération *rename* qui dépend supra-linéairement du nombre de blocs et d'éléments stockés dans l'état (cf. Tableau 3.1). Il est donc nécessaire de prendre en compte ce résultat et de (i) concevoir des stratégies de génération des opérations *rename* pour éviter d'impacter négativement l'expérience utilisateur (ii) proposer des versions améliorées des algorithmes `RENAMEID` et `REVERTRENAMEID` pour réduire

ces temps d'intégration :

- Au lieu d'utiliser `RENAMEID`, qui renomme l'état identifiant par identifiant, nous pourrions définir et utiliser `RENAMEBLOCK`. Cette fonction permettrait de renommer l'état bloc par bloc, offrant ainsi une meilleure complexité en temps. De plus, puisque les opérations *rename* fusionnent les blocs existants en un unique bloc, `RENAMEBLOCK` permettrait de mettre en place un cercle vertueux où chaque opération *rename* réduirait le temps d'exécution de la suivante.
- Puisque chaque appel à `REVERTRENAMEID` et `REVERTRENAMEID` est indépendant des autres, ces fonctions sont adaptées à la programmation parallèle. Au lieu de renommer les identifiants (ou blocs) de manière séquentielle, nous pourrions diviser la séquence en plusieurs parties et les renommer en parallèle.

Un autre résultat intéressant de ces benchmarks est que les opérations *concurrentes introduisant une plus petite époque* sont rapides à intégrer. Puisque ces opérations introduisent une époque qui n'est pas sélectionnée comme nouvelle époque cible, les noeuds ne procèdent pas au renommage de leur état. L'intégration des opérations *concurrentes introduisant une plus petite époque* consiste simplement à ajouter l'époque introduite et l'*ancien état* correspondant à l'*arbre des époques*. Les noeuds peuvent donc réduire de manière significative le coût d'intégration d'un ensemble d'opérations *rename* concurrentes en les appliquant dans l'ordre le plus adapté en fonction du contexte. Nous développons ce sujet dans la sous-section 3.5.5.

Temps pour rejouer le log d'opérations

Afin de comparer les performances de `RenamableLogootSplit` et de `LogootSplit` de manière globale, nous avons mesuré le temps nécessaire pour un nouveau noeud pour rejouer l'entièreté du log d'opérations d'une session de collaboration, en fonction du nombre de *renaming bots* de la session. Nous présentons les résultats obtenus dans la Figure 3.16.

Nous observons que le gain sur le temps d'intégration des opérations *insert* et *remove* permet initialement de contrebalancer le surcoût des opérations *rename*. Mais au fur et à mesure que la collaboration progresse, le temps nécessaire pour intégrer les opérations *rename* augmente car plus d'éléments sont impliqués. Cette tendance est accentuée dans les scénarios avec des opérations *rename* concurrentes.

Dans un cas réel d'utilisation, ce scénario (c.-à-d. rejouer l'entièreté du log) ne correspond pas au scénario principal et peut être mitigé, par exemple en utilisant un mécanisme de compression du log d'opérations. Dans la sous-section 3.5.6, nous présentons comment mettre en place un tel mécanisme en se basant justement sur les possibilités offertes par l'opération *rename*.

Impact de la fréquence de l'opération *rename* sur les performances

Pour évaluer l'impact de la fréquence de l'opération *rename* sur les performances, nous avons réalisé un benchmark supplémentaire. Ce benchmark consiste à rejouer les logs d'opérations des simulations en utilisant divers CRDTs et configurations : `LogootSplit`, `RenamableLogootSplit` effectuant des opérations *rename* toutes les 30k opérations,

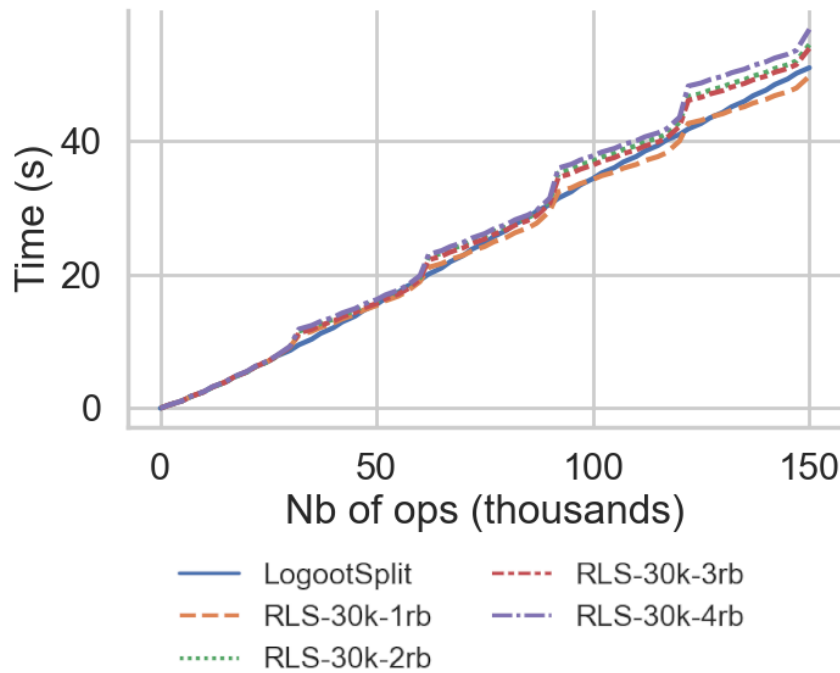


FIGURE 3.16 – Progression du nombre d'opérations du log rejouées en fonction du temps

RenamableLogootSplit effectuant des opérations *rename* toutes les 7.5k opérations. Au fur et à mesure que le benchmark rejoue le log des opérations, il mesure le temps d'intégration des opérations ainsi que leur taille. Les résultats de ce benchmark sont présentés dans le Tableau 3.4.

Paramètres		Temps d'intégration (μ s)					Taille (o)				
Type	CRDT	Moyenne	Médiane	IQR	1 ^{er} Percent.	99 ^{ème} Percent.	Moyenne	Médiane	IQR	1 ^{er} Percent.	99 ^{ème} Percent.
insert	LS	471	460	130	224	768	593	584	184	216	1136
	RLS - 30k	397	323	66.7	171	587	442	378	92	314	958
	RLS - 7.5k	393	265	54.5	133	381	389	378	0	314	590
remove	LS	280	270	71.4	140	435	632	618	184	250	1170
	RLS - 30k	247	181	39	97.9	308	434	412	0	320	900
	RLS - 7.5k	296	151	34.8	74.9	214	401	412	0	320	596
Paramètres		Temps d'intégration (ms)					Taille (Ko)				
Type	CRDT	Moyenne	Médiane	IQR	1 ^{er} Percent.	99 ^{ème} Percent.	Moyenne	Médiane	IQR	1 ^{er} Percent.	99 ^{ème} Percent.
rename	RLS - 30k	1022	1188	425	540	1276	1366	1258	514	635	3373
	RLS - 7.5k	861	974	669	123	1445	273	302	132	159	542

TABLE 3.4 – Temps d'intégration et taille des opérations par type et par fréquence d'opérations *rename*

Concernant les temps d'intégration, nous observons des opérations *rename* plus fréquentes permettent d'améliorer les temps d'intégration des opérations *insert* et *remove*. Cela confirme les résultats attendus puisque l'opération *rename* réduit la taille des identifiants de la structure ainsi que le nombre de blocs composant la séquence.

Nous remarquons aussi que la fréquence n'a aucun impact significatif sur le temps d'intégration des opérations *rename*. Il s'agit là aussi d'un résultat attendu puisque la

complexité en temps de l'implémentation de l'opération *rename* dépend du nombre d'éléments dans la séquence, un facteur qui n'est pas impacté par les opérations *rename*.

Concernant la taille des opérations, nous observons que les opérations *insert* et *remove* de *RenamableLogootSplit* sont initialement plus lourdes que les opérations correspondantes de *LogootSplit*, notamment car elles intègrent leur époque de génération comme donnée additionnelle. Mais alors que la taille des opérations de *LogootSplit* augmentent indéfiniment, celle des opérations de *RenamableLogootSplit* est bornée. La valeur de cette borne est définie par la fréquence de l'opération *rename*. Cela permet à *RenamableLogootSplit* d'atteindre un coût moindre par opération.

D'un autre côté, le coût des opérations *rename* est bien plus important (1000x) que celui des autres types d'opérations. Ceci s'explique par le fait que l'opération *rename* intègre l'*ancien état*, c.-à-d. la liste de tous les blocs composant l'état de la séquence au moment de la génération de l'opération. Cependant, nous observons le même phénomène pour les opérations *rename* que pour les autres opérations : la fréquence des opérations *rename* permet d'établir une borne pour la taille des opérations *rename*. Nous pouvons donc choisir d'émettre fréquemment des opérations *rename* pour limiter leur taille respective. Ceci implique néanmoins un surcoût en computations pour chaque opération *rename* dans l'implémentation actuelle. Nous présentons une autre approche possible pour limiter la taille des opérations *rename* dans la sous-section 3.5.3. Cette approche consiste à implémenter un mécanisme de compression pour les opérations *rename* pour ne transmettre que les composants nécessaires à l'identifiant de chaque bloc de l'*ancien état*.

3.5 Discussion

Matthieu: TODO : Ajouter une partie sur la discussion qu'on a pu avoir avec les reviewers sur la présence de pierres tombales dans RenamableLogootSplit, et comment ces pierres tombales diffèrent de celles présentes dans WOOT et RGA.

3.5.1 Stratégie de génération des opérations *rename*

Comme indiqué dans la sous-section 3.1.1, les opérations *rename* sont des opérations systèmes. C'est donc aux concepteurs de systèmes qu'incombe la responsabilité de déterminer quand les noeuds devraient générer des opérations *rename* et de définir une stratégie correspondante. Il n'existe cependant pas de solution universelle, chaque système ayant ses particularités et contraintes.

Plusieurs aspects doivent être pris en compte lors de la définition de la stratégie de génération des opérations *rename*. Le premier porte sur la taille de la structure de données. Comme illustré dans la Figure 3.14, les métadonnées augmentent de manière progressive jusqu'à représenter 99% de la structure de données. En utilisant les opérations *rename*, les noeuds peuvent supprimer les métadonnées et ainsi réduire la taille de la structure à un niveau acceptable. Pour déterminer quand générer des opérations *rename*, les noeuds peuvent donc monitorer le nombre d'opérations effectuées depuis la dernière opération *rename*, le nombre de blocs qui composent la séquence ou encore la taille des identifiants.

Un second aspect à prendre en compte est le temps d'intégration des opérations *rename*. Comme indiqué dans le Tableau 3.3, l'intégration des opérations *rename* distantes peut nécessiter des secondes si elles sont retardées trop longtemps. Bien que les opérations *rename* travaillent en coulisses, elles peuvent néanmoins impacter négativement l'expérience utilisateur. Notamment, les noeuds ne peuvent pas intégrer d'autres opérations *distantes* tant qu'ils sont en train de traiter des opérations *rename*. Du point de vue des utilisateurs, les opérations *rename* peuvent alors être perçues comme des pics de latence. Dans le domaine de l'édition collaborative temps réel, IGNAT et al. [31, 32] ont montré que le délai dégradait la qualité des collaborations. Il est donc important de générer fréquemment des opérations *rename* pour conserver leur temps d'intégration sous une limite perceptible.

Finalement, le dernier aspect à considérer est le nombre d'opérations *rename* concurrentes. La Figure 3.14 montre que les opérations *rename* concurrentes accroissent la taille de la structure de données tandis que la Figure 3.16 illustre qu'elles augmentent le temps nécessaire pour rejouer le log d'opérations. La stratégie proposée doit donc viser à minimiser le nombre d'opérations *rename* concurrentes générées. Cependant, elle doit éviter d'utiliser des coordinations synchrones entre les noeuds pour cela (e.g. algorithmes de consensus), pour des raisons de performances. Pour réduire la probabilité de générer des opérations *rename* concurrentes, plusieurs méthodes peuvent être proposées. Par exemple, les noeuds peuvent monitorer à quels autres noeuds ils sont connectés actuellement et déléguer au noeud ayant le plus grand *identifiant de noeud* la responsabilité de générer les opérations *rename*.

Pour récapituler, nous pouvons proposer plusieurs stratégies de génération des opérations *rename*, pour minimiser de manière individuelle chacun des paramètres présentés. Mais bien que certaines de ces stratégies convergent (minimiser la taille de la structure de données et minimiser le temps d'intégration des opérations *rename*), d'autres entrent en conflit (générer une opération *rename* dès qu'un seuil est atteint vs. minimiser le nombre d'opérations *rename* concurrentes générées). Les concepteurs de systèmes doivent proposer un compromis entre les différents paramètres en fonction des contraintes du système concerné (application temps réel ou asynchrone, limitations matérielles des noeuds...). Il est donc nécessaire d'analyser le système pour évaluer ses performances sur chaque aspect, ses usages et trouver le bon compromis entre tous les paramètres de la stratégie de renommage. Par exemple, dans le contexte des systèmes d'édition collaborative temps réel, [31] a montré que le délai diminue la qualité de la collaboration. Dans de tels systèmes, nous viserions donc à conserver le temps d'intégration des opérations (en incluant les opérations *rename*) en dessous du temps limite correspondant à leur perception par les utilisateurs.

3.5.2 Stockage des états précédents sur disque

Les noeuds doivent conserver les *anciens états* associés aux opérations *rename* pour transformer les opérations issues d'époques précédentes ou concurrentes. Les noeuds peuvent recevoir de telles opérations dans deux cas précis : (i) des noeuds ont émis récemment des opérations *rename* (ii) des noeuds se sont récemment reconnectés. Entre deux de ces événements spécifiques, les *anciens états* ne sont pas nécessaires pour traiter les opérations.

Nous pouvons donc proposer l'optimisation suivante : décharger les *anciens états* sur le disque jusqu'à leur prochaine utilisation ou jusqu'à ce qu'ils puissent être supprimés de manière sûre. Décharger les *anciens états* sur le disque permet de mitiger le surcoût en mémoire introduit par le mécanisme de renommage. En échange, cela augmente le temps d'intégration des opérations nécessitant un *ancien état* qui a été déchargé précédemment.

Les noeuds peuvent adopter différentes stratégies, en fonction de leurs contraintes, pour déterminer les *anciens états* comme déchargeables et pour les récupérer de manière préemptive. La conception de ces stratégies peut reposer sur différentes heuristiques : les époques des noeuds actuellement connectés, le nombre de noeuds pouvant toujours émettre des opérations concurrentes, le temps écoulé depuis la dernière utilisation de l'*ancien état*...

3.5.3 Compression et limitation de la taille de l'opération *rename*

Pour limiter la consommation en bande passante des opérations *rename*, nous proposons la technique de compression suivante. Au lieu de diffuser les identifiants complets formant l'*ancien état*, les noeuds peuvent diffuser seulement les éléments nécessaires pour identifier de manière unique les intervalles d'identifiants. En effet, un identifiant peut être caractérisé de manière unique par le triplet composé de l'*identifiant de noeud*, du *numéro de séquence* et de l'*offset* de son dernier tuple. Par conséquent, un intervalle d'identifiants peut être identifié de manière unique à partir du triplet signature de son identifiant de début et de sa longueur, c.-à-d. du quadruplet $\langle nodeId, nodeSeq, offsetBegin, offsetEnd \rangle$. Cette méthode nous permet de réduire les données à diffuser dans le cadre de l'opération *rename* à un montant fixe par intervalle.

Pour décompresser l'opération reçue, les noeuds doivent reformer les intervalles d'identifiants correspondant aux quadruplets reçus. Pour cela, ils parcourent leur état. Lorsqu'ils rencontrent un identifiant partageant le même couple $\langle nodeId, nodeSeq \rangle$ qu'un des intervalles de l'opération *rename*, les noeuds disposent de l'ensemble des informations requises pour le reconstruire. Cependant, certains couples $\langle nodeId, nodeSeq \rangle$ peuvent avoir été supprimés en concurrence et ne plus être présents dans la séquence. Dans ce cas, il est nécessaire de parcourir le log des opérations *remove* concurrentes pour retrouver les identifiants correspondants et reconstruire l'opération *rename* originale.

Grâce à cette méthode de compression, nous pouvons instaurer une taille maximale à l'opération *rename*. En effet, les noeuds peuvent émettre une opération *rename* dès que leur état courant atteint un nombre donné d'intervalles d'identifiants, bornant ainsi la taille du message à diffuser.

3.5.4 Définition de relations de priorité pour minimiser les traitements

Bien que la relation *priority* proposée dans la sous-section 3.3.2 est simple et garantit que tous les noeuds désignent la même époque comme époque cible, elle introduit un surcoût computationnel significatif dans certains cas. La Figure 3.17 présente un tel cas.

Dans cet exemple, les noeuds A et B éditent en collaboration un document. Au fur et à mesure de leur collaboration, ils effectuent plusieurs opérations *rename*. Cependant,

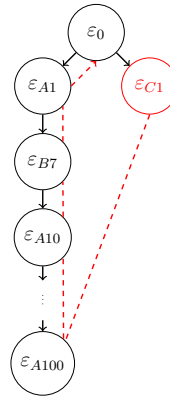
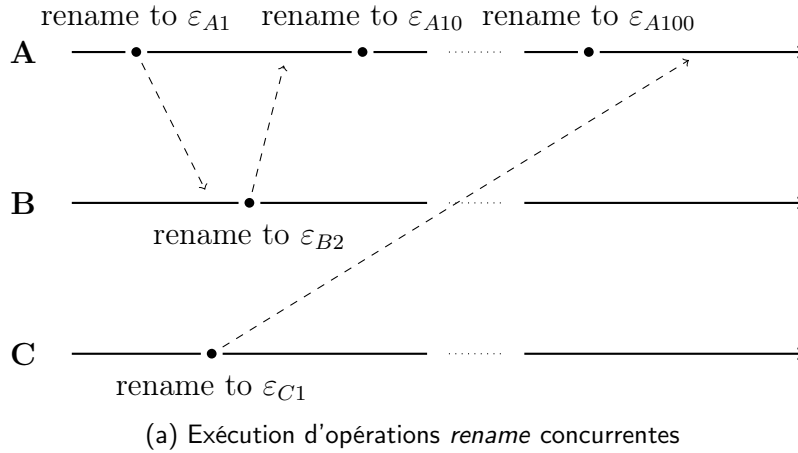


FIGURE 3.17 – Livraison d'une opération *rename* d'un noeud

après un nombre conséquent de modifications de leur part, un autre noeud C se reconnecte. Celui-ci leur transmet sa propre opération *rename*, concurrente à toutes leurs opérations. D'après la relation *priority*, nous avons $\varepsilon_0 <_{\varepsilon} \varepsilon_{A1} <_{\varepsilon} \dots <_{\varepsilon} \varepsilon_{A100} <_{\varepsilon} \varepsilon_{C1}$. La nouvelle époque cible étant ε_{C1} , les noeuds A et B doivent pour l'atteindre annuler successivement l'ensemble des opérations *rename* composant leur branche de l'*arbre des époques*. Ainsi, un noeud isolé peut forcer l'ensemble des noeuds à effectuer un lourd calcul. Il serait plus efficace que, dans cette situation, ce soit seulement le noeud isolé qui doive se mettre à jour.

La relation *priority* devrait donc être conçue pour garantir la convergence des noeuds, mais aussi pour minimiser les calculs effectués globalement par les noeuds du système. Pour concevoir une relation *priority* efficace, nous pourrions incorporer dans les opérations *rename* des métriques qui représentent l'état du système et le travail accumulé sur le document (nombre de noeuds actuellement à l'époque *parente*, nombre d'opérations générées depuis l'époque *parente*, taille du document...). De cette manière, nous pourrions favoriser la branche de l'*arbre des époques* regroupant les collaborateurs les plus actifs et empêcher les noeuds isolés d'imposer leurs opérations *rename*.

Afin d'offrir une plus grande flexibilité dans la conception de la relation *priority*, il est nécessaire de retirer la contrainte interdisant aux noeuds de rejouer une opération

rename. Pour cela, un couple de fonctions réciproques doit être proposée pour `RENAMEID` et `REVERTRENAMEID`. Une solution alternative est de proposer une implémentation du mécanisme de renommage qui repose sur les identifiants originaux plutôt que sur ceux transformés, par exemple en utilisant le log des opérations.

3.5.5 Report de la transition vers la nouvelle époque cible

Comme illustré par le Tableau 3.3, intégrer des opérations *rename* distantes est généralement coûteux. Ce traitement peut générer un surcoût computationnel significatif en cas de multiples opérations *rename* concurrentes. En particulier, un noeud peut recevoir et intégrer les opérations *rename* concurrentes dans l'ordre inverse défini par la relation *priority* sur leur époques. Dans ce scénario, le noeud considérerait chaque nouvelle époque introduite comme la nouvelle époque cible et renommerait son état en conséquence à chaque fois.

En cas d'un grand nombre d'opérations *rename* concurrentes, nous proposons que les noeuds délaient le renommage de leur état vers l'époque cible jusqu'à ce qu'ils aient obtenu un niveau de confiance donné en l'époque cible. Ce délai réduit la probabilité que les noeuds effectuent des traitements inutiles. Plusieurs stratégies peuvent être proposées pour calculer le niveau de confiance en l'époque cible. Ces stratégies peuvent reposer sur une variété de métriques pour produire le niveau de confiance, tel que le temps écoulé depuis que le noeud a reçu une opération *rename* concurrente et le nombre de noeuds en ligne qui n'ont pas encore reçu l'opération *rename*.

Durant cette période d'incertitude introduite par le report, les noeuds peuvent recevoir des opérations provenant d'époques différentes, notamment de l'époque cible. Néanmoins, les noeuds peuvent toujours intégrer les opérations *insert* et *remove* en utilisant `RENAMEID` et `REVERTRENAMEID` au prix d'un surcoût computationnel pour chaque identifiant. Cependant, ce coût est négligeable (plusieurs centaines de microsecondes par identifiant d'après la Figure 3.15b) comparé au coût de renommer, de manière inutile, complètement l'état (plusieurs centaines de millisecondes à des secondes complètes d'après le Tableau 3.3).

Notons que ce mécanisme nécessite que `RENAMEID` et `REVERTRENAMEID` soient des fonctions réciproques. En effet, au cours de la période d'incertitude, un noeud peut avoir à utiliser `REVERTRENAMEID` pour intégrer les identifiants d'opérations *insert* distantes provenant de l'époque cible. Ensuite, le noeud peut devoir renommer son état vers l'époque cible une fois que celle-ci a obtenu le niveau de confiance requis. Il s'ensuit que `RENAMEID` doit restaurer les identifiants précédemment transformés par `REVERTRENAMEID` à leur valeur initiale pour garantir la convergence.

3.5.6 Utilisation de l'opération de renommage comme mécanisme de compression du log d'opérations

Lorsqu'un nouveau pair rejoint la collaboration, il doit tout d'abord récupérer l'état courant du document avant de pouvoir participer. Le nouveau pair utilise un mécanisme d'anti-entropie [33] pour récupérer l'ensemble des opérations via un autre pair. Puis il reconstruit l'état courant en appliquant successivement chacune des opérations. Ce pro-

cessus peut néanmoins s'avérer coûteux pour les documents comprenant des milliers d'opérations.

Pour pallier ce problème, des mécanismes de compression du log ont été proposés dans la littérature. Les approches présentées dans [34, 35] consistent à remplacer un sous-ensemble des opérations du log par une opération équivalente, par exemple en agrégeant les opérations *insert* adjacentes. Une autre approche, présentée dans [36], définit une relation *obsolete* sur les opérations. La relation *obsolete* permet de spécifier qu'une nouvelle opération rend obsolètes des opérations précédentes et permet de les retirer du log. Pour donner un exemple, une opération d'ajout d'un élément donné dans un OR-Set CRDT rend obsolètes toutes les opérations précédentes d'ajout et de suppression de cet élément.

Dans notre contexte, il est intéressant de noter que l'opération *rename* peut endosser un rôle comparable à ces mécanismes de compression du log. En effet, l'opération *rename* prend un état donné, somme des opérations passées, et génère en retour un nouvel état équivalent et compacté. Une opération *rename* rend donc obsolète l'ensemble des opérations dont elle dépend causalement, et peut être utilisée pour les remplacer. En partant de cette observation, nous proposons le mécanisme de compression du log suivant.

Le mécanisme consiste à réduire le nombre d'opérations transmises à un nouveau pair rejoignant la collaboration grâce à l'opération *rename* de l'époque courante. L'opération *rename* ayant introduite l'époque courante fournit un état initial au nouveau pair. À partir de cet état initial, le nouveau pair peut obtenir l'état courant en intégrant les opérations *insert* et *remove* qui ont été générées de manière concurrente ou causale par rapport à l'opération *rename*. En réponse à une demande de synchronisation d'un nouveau pair, un pair peut donc simplement lui envoyer un sous-ensemble de son log composé de : (i) l'opération *rename* ayant introduite son époque courante (ii) les opérations *insert* et *remove* dont l'opération *rename* courante ne dépend pas causalement.

Notons que les données contenues dans l'opération *rename* telle que nous l'avons définie précédemment (cf. Définition 14) sont insuffisantes pour cette utilisation. En effet, les données incluses (*ancien état* au moment du renommage, identifiant du noeud auteur de l'opération *rename* et son numéro de séquence au moment de la génération) nous permettent seulement de recréer la structure de la séquence après le renommage. Mais le contenu de la séquence est omis, celui-ci n'étant jusqu'ici d'aucune utilité pour l'opération *rename*. Afin de pouvoir utiliser l'opération *rename* comme état initial, il est nécessaire d'y inclure cette information.

De plus, des informations de causalité doivent être intégrées à l'opération *rename*. Ces informations doivent permettre aux noeuds d'identifier les opérations supplémentaires nécessaires pour obtenir l'état courant, c.-à-d. toutes les opérations desquelles l'opération *rename* ne dépend pas causalement. L'ajout à l'opération *rename* d'un *vecteur de version*, structure représentant l'ensemble des opérations intégrées par l'auteur de l'opération *rename* au moment de sa génération, permettrait cela.

Nous définissons donc de la manière suivante l'opération *rename* enrichie compatible avec ce mécanisme de compression du log :

Définition 21 (*rename enrichie*). Une opération *rename enrichie* est un quintuplet $\langle \text{nodeId}, \text{nodeSeq}, \text{formerState}, \text{versionVector}, \text{content} \rangle$ où

- *nodeId* est l'identifiant du noeud qui a générée l'opération *rename*.

- `nodeSeq` est le numéro de séquence du noeud au moment de la génération de l'opération *rename*.
- `formerState` est l'ancien état du noeud au moment du renommage.
- `versionVector` est le vecteur de version représentant l'ancien état du noeud au moment du renommage.
- `content` est le contenu du document au moment du renommage.

Ce mécanisme de compression du log introduit néanmoins le problème suivant. Un nouveau pair synchronisé de cette manière ne possède qu'un sous-ensemble du log des opérations. Si ce pair reçoit ensuite une demande de synchronisation d'un second pair, il est possible qu'il ne puisse répondre à la requête. Par exemple, le pair ne peut pas fournir des opérations faisant partie des dépendances causales de l'opération *rename* qui lui a servi d'état initial.

Une solution possible dans ce cas de figure est de rediriger le second pair vers un troisième pour qu'il se synchronise avec lui. Cependant, cette solution pose des problèmes de latence/temps de réponse si le troisième pair s'avère indisponible à ce moment. Une autre approche possible est de généraliser le processus de synchronisation que nous avons présenté ici (opération *rename* comme état initial puis application des autres opérations) à l'ensemble des pairs, et non plus seulement aux nouveaux pairs. Nous présentons les avantages et inconvénients de cette approche dans la sous-section suivante.

3.5.7 Implémentation alternative de l'intégration de l'opération *rename* basée sur le journal d'opérations

Nous avons décrit précédemment dans la section 3.3.4, et plus précisément dans l'Algorithme 4, le processus d'intégration de l'opération *rename* évaluée dans ce manuscrit. Pour rappel, le processus consiste à (i) identifier le chemin entre l'époque courante et l'époque cible (ii) appliquer les fonctions de transformations `REVERTRENAMEID` et `RENAMEID` à l'ensemble des identifiants composant l'état courant (iii) re-crée une séquence à partir des nouveaux identifiants calculés et du contenu courant.

Dans cette section, nous abordons une implémentation alternative de l'intégration de l'opération *rename*. Cette implémentation repose sur le log des opérations.

Cette implémentation se base sur les observations suivantes : (i) L'état courant est obtenu en intégrant successivement l'ensemble des opérations. (ii) L'opération *rename* est une opération subsumant les opérations passées : elle prend un état donné (l'*ancien état*), somme des opérations précédentes, et génère un nouvel état équivalent compacté. (iii) L'ordre d'intégration des opérations concurrentes n'a pas d'importance sur l'état final obtenu.

Ainsi, pour intégrer une opération *rename* distante, un noeud peut (i) générer l'état correspondant au renommage de l'*ancien état* (ii) identifier le chemin entre l'époque courante et l'époque cible (iii) identifier les opérations concurrentes à l'opération *rename* présentes dans son log (iv) transformer et intégrer successivement les opérations concurrentes à l'opération *rename* à ce nouvel état

Cet algorithme est équivalent à ré-ordonner le log des opérations de façon à intégrer les opérations précédant l'opération *rename*, puis à intégrer l'opération *rename* elle-même, puis à intégrer les opérations concurrentes à cette dernière.

Cette approche présente plusieurs avantages par rapport à l'implémentation décrite dans la section 3.3.4. Tout d'abord, elle modifie le facteur du nombre de transformations à effectuer. La version décrite dans la section 3.3.4 transforme de l'époque courante vers l'époque cible chaque identifiant (ou chaque bloc si nous disposons de `RENAMEBLOCK`) de l'état courant. La version présentée ici effectue une transformation pour chaque opération du log concurrente à l'opération *rename* à intégrer. Le nombre de transformation peut donc être réduit de plusieurs ordres de grandeur avec cette approche, notamment si les opérations sont propagées aux pairs du réseau rapidement.

Un autre avantage de cette approche est qu'elle permet de récupérer et de réutiliser les identifiants originaux des opérations. Lorsqu'une suite de transformations est appliquée sur les identifiants d'une opération, elle est appliquée sur les identifiants originaux et non plus sur leur équivalents présents dans l'état courant. Ceci permet de réinitialiser les transformations appliquées à un identifiant et d'éviter le cas de figure mentionné dans la sous-section 3.3.3 : le cas où `REVERTRENAMEID` est utilisé pour retirer l'effet d'une opération *rename* sur un identifiant, avant d'utiliser `RENAMEID` pour ré-intégrer l'effet de la même opération *rename*. Cette implémentation supprime donc la contrainte de définir un couple de fonctions réciproques `RENAMEID` et `REVERTRENAMEID`, ce qui nous offre une plus grande flexibilité dans le choix de la relation $<_{\varepsilon}$ et du couple de fonctions `RENAMEID` et `REVERTRENAMEID`.

Cette implémentation dispose néanmoins de plusieurs limites. Tout d'abord, elle nécessite que chaque noeud maintienne localement le log des opérations. Les métadonnées accumulées par la structure de données répliquées vont alors croître avec le nombre d'opérations effectuées. Cependant, ce défaut est à nuancer. En effet, les noeuds doivent déjà maintenir le log des opérations pour le mécanisme d'anti-entropie, afin de renvoyer une opération passée à un noeud l'ayant manquée. Plus globalement, les noeuds doivent aussi conserver le log des opérations pour permettre à un nouveau noeud de rejoindre la collaboration et de calculer l'état courant en rejouant l'ensemble des opérations. Il s'agit donc d'une contrainte déjà imposée aux noeuds pour d'autres fonctionnalités du système.

Un autre défaut de cette implémentation est qu'elle nécessite de détecter les opérations concurrentes à l'opération *rename* à intégrer. Cela implique d'ajouter des informations de causalité à l'opération *rename*, tel qu'un vecteur de version. Cependant, la taille des vecteurs de version croît de façon monotone avec le nombre de noeuds qui participent à la collaboration. Diffuser cette information à l'ensemble des noeuds peut donc représenter un coût significatif dans les collaborations à large échelle. Néanmoins, il faut rappeler que les noeuds échangent déjà régulièrement des vecteurs de version dans le cadre du fonctionnement du mécanisme d'anti-entropie. Les opérations *rename* étant rares en comparaison, ce surcoût nous paraît acceptable.

Finalement, cette approche implique aussi de parcourir le log des opérations à la recherche d'opérations concurrentes. Comme dit précédemment, la taille du log croît de façon monotone au fur et à mesure que les noeuds émettent des opérations. Cette étape du nouvel algorithme d'intégration de l'opération *rename* devient donc de plus en plus coûteuse. Des méthodes permettent néanmoins de réduire son coût computationnel. No-

tamment, chaque noeud traquent les informations de progression des autres noeuds afin de supprimer les métadonnées du mécanisme de renommage (cf. sous-section 3.3.5). Ces informations permettent de déterminer la stabilité causale des opérations et donc d'identifier les opérations qui ne peuvent plus être concurrentes à une nouvelle opération *rename*. Les noeuds peuvent ainsi maintenir, en plus du log complet des opérations, un log composé uniquement des opérations non stables causalement. Lors du traitement d'une nouvelle opération *rename*, les noeuds peuvent alors parcourir ce log réduit à la recherche des opérations concurrentes.

Matthieu: TODO : Ajouter conclusion à cette sous-section

3.6 Comparaison avec les approches existantes

3.6.1 Core-Nebula

L'approche *core-nebula* [7, 8] a été proposée pour réduire la taille des identifiants dans Treedoc [37]. Dans ces travaux, les auteurs définissent l'opération *rebalance* qui permet aux noeuds de réassigner des identifiants plus courts aux éléments du document. Cependant, cette opération *rebalance* n'est ni commutative avec les opérations *insert* et *remove*, ni avec elle-même. Pour assurer la convergence à terme [4], l'approche *core-nebula* empêche la génération d'opérations *rebalance* concurrentes. Pour ce faire, l'approche requiert un consensus entre les noeuds pour générer les opérations *rebalance*. Des opérations *insert* et *remove* sont elles toujours générées sans coordination entre les noeuds et peuvent donc être concurrentes aux opérations *rebalance*. Pour gérer les opérations concurrentes aux opérations *rebalance*, les auteurs proposent de transformer les opérations concernées par rapport aux effets des opérations *rebalance*, à l'aide un mécanisme de *catch-up*, avant de les appliquer.

Cependant, les protocoles de consensus ne passent pas à l'échelle et ne sont pas adaptés aux systèmes distribués à large échelle. Pour pallier ce problème, l'approche *core-nebula* propose de répartir les noeuds dans deux groupes : le *core* et la *nebula*. Le *core* est un ensemble, de taille réduite, de noeuds stables et hautement connectés tandis que la *nebula* est un ensemble, de taille non-bornée, de noeuds. Seuls les noeuds du *core* participent à l'exécution du protocole de consensus. Les noeuds de la *nebula* contribuent toujours au document par le biais des opérations *insert* et *remove*.

Notre travail peut être vu comme une extension de celui présenté dans *core-nebula*. Avec RenamableLogootSplit, nous adaptons l'opération *rebalance* et le mécanisme de *catch-up* à LogootSplit pour tirer partie de la fonctionnalité offerte par les blocs. De plus, nous proposons un mécanisme pour supporter les opérations *rename* concurrentes, ce qui supprime la nécessité de l'utilisation d'un protocole de consensus. Notre contribution est donc une approche plus générique puisque RenamableLogootSplit est utilisable dans des systèmes composés d'un *core* et d'une *nebula*, ainsi que dans les systèmes ne disposant pas de noeuds stables pour former un *core*.

Dans les systèmes disposant d'un *core*, nous pouvons donc combiner RenamableLogootSplit avec un protocole de consensus pour éviter la génération d'opérations *rename* concurrentes. Cette approche offre plusieurs avantages. Elle permet de se passer de tout

ce qui à attrait au support d'opérations *rename* concurrentes, c.-à-d. la définition d'une relation *priority* et l'implémentation de REVERTRENAMEID. Elle permet aussi de simplifier l'implémentation du mécanisme de récupération de mémoire des époques et *anciens états* pour reposer seulement sur la stabilité causale des opérations. Concernant ses performances, cette approche se comporte de manière similaire à RenamableLogootSplit avec un seul *renaming bot* (cf. sous-section 3.4.3), mais avec un surcoût correspondant au coût du protocole de consensus sélectionné.

3.6.2 LSEQ

L'approche LSEQ [38, 39] est une approche visant à réduire la croissance des identifiants dans les Séquences CRDTs à identifiants densément ordonnés. Au lieu de réduire périodiquement la taille des métadonnées liées aux identifiants à l'aide d'un mécanisme de renommage coûteux, les auteurs définissent de nouvelles stratégies d'allocation des identifiants pour réduire leur vitesse de croissance. Dans ces travaux, les auteurs notent que la stratégie d'allocation des identifiants proposée dans Logoot [40] n'est adaptée qu'à un seul comportement d'édition : de gauche à droite, de haut en bas. Si les insertions sont effectuées en suivant d'autres comportements, les identifiants générés saturent rapidement l'espace des identifiants pour une taille donnée. Les insertions suivantes déclenchent alors une augmentation de la taille des identifiants. En conséquent, la taille des identifiants dans Logoot augmente de façon linéaire au nombre d'insertions, au lieu de suivre la progression logarithmique attendue.

LSEQ définit donc plusieurs stratégies d'allocation d'identifiants adaptées à différents comportements d'édition. Les noeuds choisissent aléatoirement une de ces stratégies pour chaque taille d'identifiants. De plus, LSEQ adopte une structure d'arbre exponentiel pour allouer les identifiants : l'intervalle des identifiants possibles double à chaque fois que la taille des identifiants augmente. Cela permet à LSEQ de choisir avec soin la taille des identifiants et la stratégie d'allocation en fonction des besoins. En combinant les différentes stratégies d'allocation avec la structure d'arbre exponentiel, LSEQ offre une croissance polylogarithmique de la taille des identifiants en fonction du nombre d'insertions.

Bien que l'approche LSEQ réduit la vitesse de croissance des identifiants dans les Séquences CRDTs à identifiants densément ordonnés, le surcoût de la séquence reste proportionnel à son nombre d'éléments. À l'inverse, le mécanisme de renommage de RenamableLogootSplit permet de réduire les métadonnées à une quantité fixe, indépendamment du nombre d'éléments.

Ces deux approches sont néanmoins orthogonales et peuvent, comme avec l'approche précédente, être combinées. Le système résultant réinitialiserait périodiquement les métadonnées de la séquence répliquée à l'aide de l'opération *rename* tandis que les stratégies d'allocation d'identifiants de LSEQ réduiraient leur croissance entretemps. Cela permettrait aussi de réduire la fréquence de l'opération *rename*, réduisant ainsi les calculs effectués par le système de manière globale.

3.7 Conclusion

Dans ce chapitre, nous avons présenté un nouvel Sequence CRDT : RenamableLogootSplit. Ce nouveau type de données répliquées associe à LogootSplit un mécanisme de renommage optimiste permettant de réduire périodiquement les métadonnées stockées et d'optimiser l'état interne de la structure de données.

Ce mécanisme prend la forme d'une nouvelle opération, l'opération *rename*, qui peut être émise à tout moment par n'importe quel noeud. Cette opération génère une nouvelle séquence LogootSplit, équivalente à l'état précédent, avec une empreinte minimale en métadonnées. L'opération *rename* transporte aussi suffisamment d'informations pour que les noeuds puissent intégrer les opérations concurrentes à l'opération *rename* dans le nouvel état.

En cas d'opérations *rename* concurrentes, la relation d'ordre strict total $<_{\epsilon}$ permet aux noeuds de décider quelle opération *rename* utiliser, sans coordination. Les autres opérations *rename* sont quant à elles ignorées. Seules leurs informations sont stockées par RenamableLogootSplit, afin de gérer les opérations concurrentes potentielles.

Une fois qu'une opération *rename* a été propagée à l'ensemble des noeuds, elle devient causalement stable. À partir de ce point, il n'est plus possible qu'un noeud émette une opération concurrente à cette dernière. Les informations incluses dans l'opération *rename* pour intégrer les opérations concurrentes potentielles peuvent donc être supprimées par l'ensemble des noeuds.

Ainsi, le mécanisme de renommage permet à RenamableLogootSplit d'offrir de meilleures performances que LogootSplit. La génération du nouvel état minimal et la suppression à terme des métadonnées du mécanisme de renommage divisent par 100 la taille de la structure de données répliquée. L'optimisation de l'état interne représentant la séquence réduit aussi le coût d'intégration des opérations suivantes, amortissant ainsi le coût de transformation et d'intégration des opérations concurrentes à l'opération *rename*.

RenamableLogootSplit souffre néanmoins de plusieurs limitations. La première d'entre elles est le besoin d'observer la stabilité causale des opérations *rename* pour supprimer de manière définitive les métadonnées associées. Il s'agit d'une contrainte forte, notamment dans les systèmes dynamiques à grande échelle dans lesquels nous n'avons aucune garantie et aucun contrôle sur les noeuds. Il est donc possible qu'un noeud déconnecté ne se reconnecte jamais, bloquant ainsi la progression de la stabilité causale pour l'ensemble des opérations. Il s'agit toutefois d'une limite partagée avec les autres mécanismes de réduction des métadonnées pour Sequence CRDTs proposés dans la littérature [41, 8], à l'exception de l'approche LSEQ [39]. En pratique, il serait intéressant d'étudier la mise en place d'un mécanisme d'éviction des noeuds inactifs pour répondre à ce problème.

Matthieu: TODO : Revoir ce point pour indiquer que stabilité causale n'est pas une condition raisonnable dans systèmes sujets au churn. Qu'à notre sens, rend cette solution inadéquate par rapport au modèle du système. Mais préciser que majorité des noeuds dans ce type de système se connectent de manière éphémère, c.-à-d. ne reviennent jamais. Mais principe d'une collaboration est de collaborer. Si noeuds ne collaborent pas ou mal, e.g. font un truc dans leur coin pendant X mois (dépendant du cas d'application), nous paraît justifier de les retirer de la collaboration.

La seconde limitation de RenamableLogootSplit concerne la génération d'opérations

rename concurrentes. Chaque opération *rename* est coûteuse, aussi bien en terme de métadonnées à stocker et diffuser qu'en terme de traitements à effectuer. Il est donc important de chercher à minimiser le nombre d'opérations *rename* concurrentes émises par les noeuds. Une approche possible est d'adopter une architecture du type *core-nebula*[8]. Mais pour les systèmes incompatibles avec ce type d'architecture système, il serait intéressant de proposer d'autres approches ne nécessitant aucune coordination entre les noeuds. Mais par définition, ces approches ne pourraient offrir de garanties fortes sur le nombre d'opérations concurrentes possibles.

Chapitre 4

Conclusions et perspectives

Sommaire

4.1	Résumés des contributions	63
4.1.1	Ré-identification sans coordination pour les CRDTs pour Séquence	63
4.1.2	Éditeur de texte collaboratif P2P chiffré de bout en bout	65
4.2	Perspectives	67
4.2.1	Définition de relations de priorité pour minimiser les traitements	67
4.2.2	Détection et fusion manuelle de versions distantes	68
4.2.3	Étude comparative des différents modèles de synchronisation pour CRDTs	71
4.2.4	Approfondissement du patron de conception de Pure Operation- Based CRDTs	73

Dans ce chapitre, nous revenons sur les contributions présentées dans cette thèse. Nous rappelons le contexte dans lequel elles s’inscrivent, récapitulons leurs spécificités et apports, et finalement présentons leurs limites que nous identifions. Puis, nous concluons ce manuscrit en présentant plusieurs pistes de recherche qui nous restent à explorer à l’issue de cette thèse. Les premières s’inscrivent dans la continuité directe de nos travaux sur un mécanisme de ré-identification pour CRDTs pour Séquence dans un système P2P à large échelle sujet au churn. Les dernières traduisent quant à elles notre volonté de recentrer nos travaux sur le domaine plus général des CRDTs.

4.1 Résumés des contributions

4.1.1 Ré-identification sans coordination pour les CRDTs pour Séquence

Pour privilégier leur disponibilité, latence et tolérance aux pannes, les systèmes distribués peuvent adopter le paradigme de la réplication optimiste [3]. Ce paradigme consiste à relaxer la cohérence de données entre les noeuds du système pour leur permettre de consulter et modifier leur copie locale sans se coordonner. Leur copies peuvent alors temporairement diverger avant de converger de nouveau une fois les modifications de chacun

propagées. Cependant, cette approche nécessite l'emploi d'un mécanisme de résolution pour assurer la convergence même en cas de modifications concurrentes. Pour cela, l'approche des CRDTs [5, 6] propose d'utiliser des types de données dont les modifications sont nativement commutatives.

Depuis la spécification des CRDTs, la littérature a proposé plusieurs de ces mécanismes résolution de conflits automatiques pour le type de données Séquence [42, 41, 37, 40]. Cependant, ces approches souffrent toutes d'un surcoût croissant de manière monotone. Ce problème a été identifié par la communauté, et celle-ci a proposé pour y répondre des mécanismes permettant soit de réduire la croissance du surcoût [38, 39], soit d'effectuer une GC du surcoût [41, 7, 8]. Nous avons cependant déterminé que ces mécanismes ne sont pas adaptés aux systèmes P2P à large échelle souffrant de churn et utilisant des CRDTs pour Séquence à granularité variable.

Dans le cadre de cette thèse, nous avons donc souhaité proposer un nouveau mécanisme adapté à ce type de systèmes. Pour cela, nous avons suivi l'approche proposée par [7, 8] : l'utilisation d'un mécanisme pour ré-assigner de nouveaux identifiants aux éléments stockés dans la séquence. Nous avons donc proposé un nouveau mécanisme appartenant à cette approche pour le CRDT LogootSplit [11].

Notre proposition prend la forme d'un nouvel CRDT pour Séquence à granularité variable : RenamableLogootSplit. Ce nouveau CRDT associe à LogootSplit un nouveau type de modification, *ren*, permettant de produire une nouvelle séquence équivalente à son état précédent. Cette nouvelle modification tire profit de la granularité variable de la séquence pour produire un état de taille minimale : elle assigne à tous les éléments des identifiants de position issus d'un même intervalle. Ceci nous permet de minimiser les métadonnées que la séquence doit stocker de manière effective. De plus, le passage à une représentation interne minimale de la séquence nous permet d'améliorer le coût des modifications suivantes en termes de calculs.

Afin de gérer les opérations concurrentes aux opérations *ren*, nous définissons pour ces dernières un algorithme de transformation. Pour cela, nous définissons un mécanisme d'époques nous permettant d'identifier la concurrence entre opérations. De plus, nous introduisons une relation d'ordre strict total, *priority*, pour résoudre de manière déterministe le conflit provoqué par deux opérations *ren*, c.-à-d. pour déterminer quelle opération *ren* privilégier. Finalement, nous définissons deux algorithmes, **renameId** et **revertRenameId**, qui permettent de transformer les opérations concurrentes à une opération *ren* pour prendre en compte l'effet de cette dernière. Ainsi, notre algorithme permet de détecter et de transformer les opérations concurrentes aux opérations *ren*, sans nécessiter une coordination synchrone entre les noeuds. Le surcoût induit par ce mécanisme, notamment en termes de calculs, est toutefois contrebalancé par l'amélioration précisée précédemment, c.-à-d. la réduction de la taille de la séquence.

Finalement, le mécanisme que nous proposons est partiellement générique : il peut être adapté à d'autres CRDTs pour Séquence à granularité variable, e.g. un CRDT pour Séquence appartenant à l'approche à pierres tombales. Dans le cadre d'une telle démarche, nous pourrions réutiliser le système d'époques, la relation *priority* et l'algorithme de contrôle qui identifie les transformations à effectuer. Pour compléter une telle adaptation, nous devrions cependant concevoir de nouveaux algorithmes **renameId** et **revertRenameId** spécifiques et adaptés au CRDT choisi.

Le mécanisme de renommage que nous présentons souffre néanmoins de plusieurs limites. La première d'entre elles concerne ses performances. En effet, notre évaluation expérimentale a mis en lumière le coût important en l'état de la modification *ren* par rapport aux autres modifications en termes de calculs (cf. section 3.4.3, page 48). De plus, chaque opération *ren* comporte une représentation de l'ancien état qui doit être maintenue par les noeuds jusqu'à leur stabilité causale. Le surcoût en métadonnées introduit par un ensemble d'opérations *ren* concurrentes peut donc s'avérer important, voire pénalisant (cf. section 3.4.3, page 45). Pour répondre à ces problèmes, nous identifions trois axes d'amélioration :

- (i) La définition de stratégies de déclenchement du renommage efficaces. Le but de ces stratégies serait de déclencher le mécanisme de renommage de manière fréquente, de façon à garder son temps d'exécution acceptable, mais tout visant à minimiser la probabilité que les noeuds produisent des opérations *ren* concurrentes, de façon à minimiser le surcoût en métadonnées.
- (ii) La définition de relations *priority* efficaces. Nous développons ce point dans la sous-section 4.2.1.
- (iii) La proposition d'algorithmes de renommage efficaces. Cette amélioration peut prendre la forme de nouveaux algorithmes pour `renameId` et `revertRenameId` offrant une meilleure complexité en temps. Il peut aussi s'agir de la conception d'une nouvelle approche pour renommer l'état et gérer les modifications concurrentes, e.g. un mécanisme de renommage basé sur le journal des opérations (cf. sous-section 3.5.7, page 57).

Une seconde limite de `RenamableLogootSplit` que nous identifions concerne son mécanisme de GC des métadonnées introduites par le mécanisme de renommage. En effet, pour fonctionner, ce dernier repose sur la stabilité causale des opérations *ren*. Pour rappel, la stabilité causale représente le contexte causal commun à l'ensemble des noeuds du système. Pour le déterminer, chaque noeud doit récupérer le contexte causal de l'ensemble des noeuds du système. Ainsi, l'utilisation de la stabilité causale comme pré-requis pour la GC de métadonnées constitue une contrainte forte, voire prohibitive, dans les systèmes P2P à large échelle sujet au churn. En effet, un noeud du système déconnecté de manière définitive suffit pour empêcher la stabilité causale de progresser, son contexte causal étant alors indéterminé du point de vue des autres noeuds. Il s'agit toutefois d'une limite récurrente des mécanismes de GC distribués et asynchrones [41, 30, 43]. Nous présentons une piste de travail possible pour pallier ce problème dans la sous-section 4.2.2.

4.1.2 Éditeur de texte collaboratif P2P chiffré de bout en bout

Les applications collaboratives permettent à des utilisateur-rices de réaliser collaborativement une tâche. Elles permettent à plusieurs utilisateur-rices de consulter la version actuelle du document, de la modifier et de partager leurs modifications avec les autres. Ceci permet de mettre en place une réflexion de groupe, ce qui améliore la qualité du résultat produit [44, 45].

Cependant, les applications collaboratives sont historiquement des applications centralisées, e.g. Google Docs [9]. Ce type d'architecture induit des défauts d'un point de vue

technique, e.g. faible capacité de passage à l'échelle et faible tolérance aux pannes, mais aussi d'un point de vue utilisateur, e.g. perte de la souveraineté des données et absence de garantie de pérennité.

Les travaux de l'équipe Coast s'inscrivent dans une mouvance souhaitant résoudre ces problèmes et qui a conduit à la définition d'un nouveau paradigme d'applications : les LFS [46]. Le but de ce paradigme est la conception d'applications collaboratives, P2P, pérennes et rendant la souveraineté de leurs données aux utilisateur-rices.

Dans le cadre de cette démarche, l'équipe Coast développe depuis plusieurs années l'application Multi User Text Editor (MUTE), un éditeur de texte web collaboratif P2P temps réel chiffré de bout en bout. Cette application sert à la fois de plateforme de démonstration et de recherche pour les travaux de l'équipe, mais aussi de PoC pour les LFS.

Dans le cadre de cette thèse, nous avons implémenté dans MUTE nos travaux de recherche portant sur le nouvel CRDT pour le type Séquence : RenamableLogootSplit. MUTE a aussi servi à l'équipe pour présenter ses travaux concernant l'authentification des utilisateur-rices dans un système P2P [15]. Finalement, MUTE nous a permis de nous d'étudier et/ou de présenter les travaux de recherche existants concernant :

- (i) Les protocoles distribués d'appartenance au groupe [16].
- (ii) Les mécanismes d'anti-entropie [33].
- (iii) Les protocoles d'établissement de clés de chiffrement de groupe [19].
- (iv) Les protocoles d'établissement de topologies réseaux efficaces [18].
- (v) Les mécanismes de conscience de groupe.

MUTE offre donc, à notre connaissance, le tour d'horizon le plus complet des travaux de recherche permettant la conception d'applications LFS. Cependant, cela ne dispense pas MUTE de souffrir de plusieurs limites.

Tout d'abord, l'environnement web implique un certain nombre de contraintes, notamment au niveau des technologies et protocoles disponibles. Notamment, le protocole WebRTC repose sur l'utilisation de serveurs de signalisation, c.-à-d. de points de rendez-vous des pairs, et de serveurs de relai, c.-à-d. d'intermédiaires pour communiquer entre pairs lorsque les configurations de leur réseaux respectifs interdisent l'établissement d'une connexion directe. Ainsi, les applications P2P web doivent soit déployer et maintenir leur propre infrastructure de serveurs, soit reposer sur une infrastructure existante, e.g. celle proposée par OpenRelay [47]. Afin de minimiser l'effort requis aux applications P2P et la confiance exigée à leurs utilisateur-rices, nous devons supporter la mise en place d'une telle infrastructure transparente et pérenne.

Une autre limite de ce système que nous identifions concerne l'utilisabilité des systèmes P2P de manière générale. L'expérience vécue suivante constitue à notre avis un exemple éloquent des limites actuelles de l'application MUTE dans ce domaine. Après avoir rédigé une version initiale d'un document, nous avons envoyé le lien du document à notre collaborateur pour relecture et validation. Lorsque notre collaborateur a souhaité accéder au document, celui-ci s'est retrouvé devant une page blanche : comme nous nous étions déconnecté du système entretemps, c.-à-d. plus aucun pair n'était disponible pour effectuer une synchronisation. Notre collaborateur était donc dans l'incapacité de récupérer l'état et d'effectuer sa tâche. Afin de pallier ce problème, une solution possible est

de faire reposer MUTE sur un réseau P2P global, e.g. le réseau de InterPlanetary File System (IPFS) [48], et d'utiliser les pairs de ce dernier, potentiellement des pairs étrangers à l'application, comme pairs de stockage pour permettre une synchronisation future. Cette solution limite ainsi le risque qu'un pair ne puisse récupérer l'état du document faute de pairs disponibles. Cependant, elle nécessite de mettre en place un mécanisme de réplication de données additionnel. Ce mécanisme de réplication sur des pairs additionnels doit cependant garantir qu'il n'introduit pas de vulnérabilités, e.g. la possibilité pour les pairs de stockage sélectionnés de reconstruire et consulter le document.

4.2 Perspectives

4.2.1 Définition de relations de priorité pour minimiser les traitements

Dans la sous-section 3.3.2, nous avons spécifié la relation *priority* (cf. Définition 17, page 26). Pour rappel, cette relation doit établir un ordre strict total sur les époques de notre mécanisme de renommage.

Cette relation nous permet ainsi de résoudre le conflit provoqué par la génération de modifications *ren* concurrentes en les ordonnant. Grâce à cette relation d'ordre, les noeuds peuvent déterminer vers quelle époque de l'ensemble des époques connues progresser. Cette relation permet ainsi aux noeuds de converger à une époque commune à terme.

La convergence à terme à une époque commune présente plusieurs avantages :

- (i) Réduire la distance entre les époques courantes des noeuds, et ainsi minimiser le surcoût en calculs par opération du mécanisme de renommage. En effet, il n'est pas nécessaire de transformer une opérations livrée avant de l'intégrer si celle-ci provient de la même époque que le noeud courant.
- (ii) Définir un nouveau PPAC entre les époques courantes des noeuds. Cela permet aux noeuds d'appliquer le mécanisme de GC pour supprimer les époques devenues obsolètes et leur anciens états associés, pour ainsi minimiser le surcoût en métadonnées du mécanisme de renommage.

Il existe plusieurs manières pour définir la relation *priority* tout en satisfaisant les propriétés indiquées. Dans le cadre de ce manuscrit, nous avons utilisé l'ordre lexicographique sur les chemins des époques dans l'*arbre des époques* pour définir *priority*. Cette approche se démarque par :

- (i) Sa simplicité.
- (ii) Son surcoût limité, c.-à-d. cette approche n'introduit pas de métadonnées supplémentaires à stocker et diffuser, et l'algorithme de comparaison utilisé est simple.
- (iii) Sa propriété arrangeante sur les déplacements des noeuds dans l'arbre des époques. De manière plus précise, cette définition de *priority* impose aux noeuds de se déplacer que vers l'enfant le plus à droite de l'arbre des époques. Ceci empêche les

noeuds de faire un aller-retour entre deux époques données. Cette propriété permet de passer outre une contrainte concernant le couple de fonctions `renameId` et `revertRenameId` : leur réciprocity.

Cette définition présente cependant plusieurs limites. La limite que nous identifions est sa décorrélation avec le coût et le bénéfice de progresser vers l'époque cible désignée. En effet, l'époque cible est désignée de manière arbitraire par rapport à sa position dans l'arbre des époques. Il est ainsi possible que progresser vers cette époque détériore l'état de la séquence, c.-à-d. augmente la taille des identifiants et augmente le nombre de blocs. De plus, la transition de l'ensemble des noeuds depuis leur époque courante respective vers cette nouvelle époque cible induit un coût en calculs, potentiellement important (cf. Figure 3.17, page 54).

Pour pallier ce problème, il est nécessaire de proposer une définition de *priority* prenant l'aspect efficacité en compte. L'approche considérée consisterait à inclure dans les opérations *ren* une ou plusieurs métriques qui représente le travail accumulé sur la branche courante de l'arbre des époques, e.g. le nombre d'opérations intégrées, les noeuds actuellement sur cette branche... L'ordre strict total entre les époques serait ainsi construit à partir de la comparaison entre les valeurs de ces métriques de leur opération *ren* respective.

Il conviendra d'adjoindre à cette nouvelle définition de *priority* un nouveau couple de fonctions `renameId` et `revertRenameId` respectant la contrainte de réciprocity de ces fonctions, ou de mettre en place une autre implémentation du mécanisme de renommage ne nécessitant pas cette contrainte, telle qu'une implémentation basée sur le journal des opérations (cf. sous-section 3.5.7, page 57).

Il conviendra aussi d'étudier la possibilité de combiner l'utilisation de plusieurs relations *priority* pour minimiser le surcoût global du mécanisme de renommage, e.g. en fonction de la distance entre deux époques.

Finalement, il sera nécessaire de valider l'approche proposée par une évaluation comparative par rapport à l'approche actuelle. Cette évaluation pourrait consister à monitorer le coût du système pour observer si l'approche proposée permet de réduire les calculs de manière globale. Plusieurs configurations de paramètres pourraient aussi être utilisées pour déterminer l'impact respectif de chaque paramètre sur les résultats.

4.2.2 Détection et fusion manuelle de versions distantes

À l'issue de cette thèse, nous constatons plusieurs limites des mécanismes de résolution de conflits automatiques dans les systèmes P2P à large échelle sujet au churn. La première d'entre elles est l'utilisation d'un contexte causal. Le contexte causal est utilisé par les mécanismes de résolution de conflits pour :

- (i) Satisfaire le modèle de cohérence causale, c.-à-d. assurer que si nous avons deux modifications m_1 et m_2 telles que $m_1 \rightarrow m_2$, alors l'effet de m_2 supplantera celui de m_1 . Ceci permet d'éviter des anomalies de comportement de la part de la structure de données du point de vue des utilisateur-rices, par exemple la résurgence d'un élément supprimé au préalable.
- (ii) Permettre de préserver l'intention d'une modification malgré l'intégration préalable de modifications concurrentes.

Le contexte causal est utilisé de manière différente en fonction du mécanisme de résolution de conflit. Dans l'approche Operational Transformation (OT), le contexte causal est utilisé par l'algorithme de contrôle pour déterminer les modifications concurrentes à une modification lors de son intégration, afin de prendre en compte leurs effets. Dans l'approche CRDT, le contexte causal est utilisé par la structure de données répliquée à la génération de la modification pour en faire une modification nativement commutative avec les modifications concurrentes, c.-à-d. pour en faire un élément du sup-demi-treillis représentant la structure de données répliquée.

Le contexte causal peut être représenté de différentes manières. Par exemple, le contexte causal peut prendre la forme d'un vecteur de version [49, 50] ou d'un Directed Acyclic Graph (DAG) des modifications [51]. Cependant, de manière intrinsèque, le contexte causal ne fait que de croître au fur et à mesure que des modifications sont effectuées ou que des noeuds rejoignent le système, incrémentant son surcoût en métadonnées, calculs et bande-passante.

La stabilité causale permet cependant de réduire le surcoût lié au contexte causal. En effet, la stabilité causale permet d'établir le contexte commun à l'ensemble des noeuds, c.-à-d. l'ensemble des modifications que l'ensemble des noeuds ont intégré. Ces modifications font alors partie de l'histoire commune et n'ont plus besoin d'être considérées par les mécanismes de résolution de conflits automatiques. La stabilité causale permet donc de déterminer et de tronquer la partie commune du contexte causal pour éviter que ce dernier ne pénalise les performances du système à terme.

La stabilité causale est cependant une contrainte forte dans les systèmes P2P dynamiques à large échelle sujet au churn. Il ne suffit en effet que d'un noeud déconnecté pour empêcher la stabilité causale de progresser. Pour répondre à ce problème, nous avons dès lors tout un spectre d'approches possibles, proposant chacune un compromis entre le surcoût du contexte causal et la probabilité de rejeter des modifications. Les extrémités de ce spectre d'approches sont les suivantes :

- (i) Considérer tout noeud déconnecté comme déconnecté de manière définitive, et donc les ignorer dans le calcul de la stabilité causale. Cette première approche permet à la stabilité causale de progresser, et ainsi aux noeuds connectés de travailler dans des conditions optimales. Mais elle implique cependant que les modifications potentielles des noeuds déconnectés soient perdues, c.-à-d. de ne plus pouvoir les intégrer en l'absence d'un lien entre leur contexte causal de génération et le contexte causal actuel de chaque autre noeud. Il s'agit là de la stratégie la plus agressive en terme de GC du contexte causal.
- (ii) Assurer en toutes circonstances la capacité d'intégration des modifications des noeuds, même ceux déconnectés. Cette seconde approche permet de garantir que les modifications potentielles des noeuds déconnectés pourront être intégrées automatiquement, dans l'éventualité où ces derniers se reconnectent à terme. Mais elle implique de bloquer potentiellement de manière définitive la stabilité causale et donc le mécanisme de GC du contexte causal. Il s'agit là de la stratégie la plus timide en terme de GC du contexte causal.

La seconde limite que nous constatons est la limite des mécanismes actuels de résolution de conflits automatiques pour préserver l'intention des utilisateur-rices. Par exemple,

les mécanismes de résolution de conflits automatiques pour le type Séquence présentés dans ce manuscrit (cf. ??, page ??) définissent l'intention de la manière suivante : *l'intégration de la modification par les noeuds distants doit reproduire l'effet de la modification sur la copie d'origine*. Cette définition assure que chaque modification est porteuse d'une intention, mais limite voire ignore toute la dimension sémantique de la dite intention. Nous conjecturons que l'absence de dimension sémantique réduit les cas d'utilisation de ces mécanismes.

Considérons par exemple une édition collaborative d'un même texte par un ensemble de noeuds. Lors de la présence d'une faute de frappe dans le texte, e.g. le mot "HLLO", plusieurs utilisateur-rices peuvent la corriger en concurrence, c.-à-d. insérer l'élément "E" entre "H" et "L". Les mécanismes de résolution de conflits automatiques permettent aux noeuds d'obtenir des résultats qui convergent mais à notre sens insatisfaisant, e.g. "HEEEEEELLO". Nous considérons ce type de résultats comme des anomalies, au même titre que l'entrelacement [52]. Dans le cadre de collaborations temps réel à échelle limitée, nous conjecturons cependant qu'une granularité fine des modifications permet de pallier ce problème. En effet, les utilisateur-rices peuvent observer une anomalie produite par le mécanisme de résolution de conflits automatique, déduire l'intention initiale des modifications concernées et la restaurer par le biais d'actions supplémentaires de compensation.

Cependant, dans le cadre de collaborations asynchrones ou à large échelle, nous conjecturons que ces anomalies de résolution de conflits s'accumulent. Cette accumulation peut atteindre un seuil rendant laborieuse la déduction et le rétablissement de l'intention initiale des modifications. Le travail imposé aux utilisateur-rices pour résoudre ces anomalies par le biais d'actions de compensation peut alors entraver la collaboration. Pour reprendre l'exemple de l'édition collaborative de texte, nous pouvons constater de tels cas suite à de la duplication de contenu et/ou l'entrelacement de mots, phrases voire paragraphes nuisant à la clarté et correction du texte. Il convient alors de s'interroger sur le bien-fondé de l'utilisation de mécanismes de résolutions de conflits automatiques pour intégrer un ensemble de modifications dans l'ensemble des situations.

Ainsi, pour répondre aux limites des mécanismes de résolution conflits automatiques dans les systèmes P2P à large échelle, c.-à-d. l'augmentation de leur surcoût et la pertinence de leur résultat, nous souhaitons proposer une approche combinant un ou des mécanismes de résolution de conflits automatiques avec un ou des mécanismes de résolution de conflits manuels. L'idée derrière cette approche est de faire varier le mécanisme de résolution de conflits utilisé pour intégrer des modifications. Le choix du mécanisme de résolution de conflits utilisé peut se faire à partir de la valeur d'une distance calculée entre la version courante de la donnée répliquée et celle de la génération de la modification à intégrer, ou d'une évaluation de la qualité du résultat de l'intégration de la modification. Par exemple :

- (i) Si la distance calculée se trouve dans un intervalle de valeurs pour lequel nous disposons d'un mécanisme de résolution de conflits automatique satisfaisant, utiliser ce dernier. Ainsi, nous pouvons envisager de reposer sur plusieurs mécanismes de résolution de conflits automatiques, de plus en plus complexes et pertinents mais coûteux, sans dégrader les performances du système dans le cas de base.
- (ii) Si la distance calculée dépasse la distance seuil, c.-à-d. que nous ne disposons plus

à ce stade de mécanismes de résolution de conflits automatiques satisfaisants, faire intervenir les utilisateur-rices par le biais d'un mécanisme de résolution de conflits manuel. L'utilisation d'un mécanisme manuel n'exclut cependant pas tout pré-travail de notre part pour réduire la charge de travail des utilisateur-rices dans le processus de fusion.

Dans un premier temps, cette approche pourrait se focaliser sur un type d'application spécifique, e.g. l'édition collaborative de texte.

Cette approche nous permettrait de répondre aux limites soulevées précédemment. En effet, elle permettrait de limiter la génération d'anomalies par le mécanisme de résolution de conflits automatique en faisant intervenir les utilisateur-rices. Puis, puisque nous déléguons aux utilisateur-rices l'intégration des modifications à partir d'une distance seuil, nous pouvons dès lors reconsidérer les métadonnées conservées par les noeuds pour les mécanismes de résolution de conflits automatiques. Notamment, nous pouvons identifier les noeuds se trouvant au-delà de cette distance seuil d'après leur dernier contexte causal connu et ne plus les prendre en compte pour le calcul de la stabilité causale. Cette approche permettrait donc de réduire le surcoût lié au contexte causal et limiter la perte de modifications, tout en prenant en considération l'ajout de travail aux utilisateur-rices.

Pour mener à bien ce travail, il conviendra tout d'abord de définir la notion de distance entre versions de la donnée répliquée. Nous envisageons de baser cette dernière sur les deux aspects temporel et spatial, c.-à-d. en utilisant la distance entre contextes causaux et la distance entre contenus. Dans le cadre de l'édition collaborative, nous pourrions pour cela nous baser sur les travaux existants pour évaluer la distance entre deux textes. *Matthieu: TODO : Insérer refs distance de Hamming, Levenshtein, String-to-string correction problem (Tichy et al)*

Il conviendra ensuite de déterminer comment établir la valeur seuil à partir de laquelle la distance entre versions est jugée trop importante. Les approches d'évaluation de la qualité du résultat [53] pourront être utilisées pour déterminer un couple \langle méthode de calcul de la distance, valeur de distance \rangle spécifiant les cas pour lesquels les méthodes de résolution de conflits automatiques ne produisent plus un résultat satisfaisant. *Matthieu: TODO : Insérer refs travaux Claudia et Vinh* Le couple obtenu pourra ensuite être confirmé par le biais d'expériences utilisateurs inspirées de [31, 32].

Finalement, il conviendra de proposer un mécanisme de résolution de conflits adapté pour gérer les éventuelles fusions d'une même modification de façon concurrente par un mécanisme automatique et par un mécanisme manuel, ou à défaut un mécanisme de conscience de groupe invitant les utilisateur-rices à effectuer des actions de compensation.

4.2.3 Étude comparative des différents modèles de synchronisation pour CRDTs

Comme évoqué dans l'état de l'art (cf. ??, page ??), un nouveau modèle de synchronisation pour CRDT fut proposé récemment [54]. Ce dernier propose une synchronisation des noeuds par le biais de différences d'états.

Pour rappel, ce nouveau modèle de synchronisation se base sur le modèle de synchronisation par états. Il partage les mêmes pré-requis, à savoir la nécessité d'une fonction

`merge` associative, commutative et idempotente. Cette dernière doit permettre de la fusion toute paire d'états possible en calculant leur borne supérieure, c.-à-d. leur LUB [24].

La spécificité de ce nouveau modèle de synchronisation est de calculer pour chaque modification la différence d'état correspondante. Cette différence correspond à un élément irréductible du sup-demi-treillis du CRDT [55], c.-à-d. un état particulier de ce dernier. Cet élément irréductible peut donc être diffusé et intégré par les autres noeuds, toujours à l'aide de la fonction `merge`.

Ce modèle de synchronisation permet alors d'adopter une variété de stratégies de synchronisation, e.g. diffusion des différences de manière atomique, fusion de plusieurs différences puis diffusion du résultat..., et donc de répondre à une grande variété de cas d'utilisation.

Dans notre comparaison des modèles de synchronisation (cf. ??, page ??), nous avons justifié que les CRDTs synchronisés par différences d'états peuvent être utilisés dans les mêmes contextes que les CRDTs synchronisés par états et que les CRDTs synchronisés par opérations. Cette conclusion nous mène à reconsidérer l'intérêt des autres modèles de synchronisation de nos jours.

Par exemple, un CRDT synchronisé par différences d'états correspond à un CRDT synchronisé par états dont nous avons identifié les éléments irréductibles. La différence entre ces deux modèles de synchronisation semble reposer seulement sur la possibilité d'utiliser ces éléments irréductibles pour propager les modifications, en place et lieu des états complets. Nous conjecturons donc que le modèle de synchronisation par états est rendu obsolète par celui par différences d'états. Il serait intéressant de confirmer cette supposition.

En revanche, l'utilisation du modèle de synchronisation par opérations conduit généralement à une spécification différente du CRDT, les opérations permettant d'encoder plus librement les modifications. Notamment, l'utilisation d'opérations peut mener à des algorithmes d'intégration des modifications différents que ceux de la fonction `merge`. Il convient de comparer ces algorithmes pour déterminer si le modèle de synchronisation par opérations peut présenter un intérêt en termes de surcoût.

Au-delà de ce premier aspect, il convient d'explorer d'autres pistes pouvant induire des avantages et inconvénients pour chacun de ces modèles de synchronisation. À l'issue de cette thèse, nous identifions les pistes suivantes :

- (i) La composition de CRDTs, c.-à-d. la capacité de combiner et de mettre en relation plusieurs CRDTs au sein d'un même système, afin d'offrir des fonctionnalités plus complexes. Par exemple, une composition de CRDTs peut se traduire par l'ajout de dépendances entre les modifications des différents CRDTs composés. Le modèle de synchronisation par opérations nous apparaît plus adapté pour cette utilisation, de par le découplage qu'il induit entre les CRDTs et la couche de livraison de messages.
- (ii) L'utilisation de CRDTs au sein de systèmes non-sûrs, c.-à-d. pouvant compter un ou plusieurs adversaires byzantins [56]. Dans de tels systèmes, les adversaires byzantins peuvent générer des modifications différentes mais qui sont perçues comme identiques par les mécanismes de résolution de conflits. Cette attaque, nommée *équivoque*, peut provoquer la divergence définitive des copies. [43] propose une solution adaptée aux systèmes P2P à large échelle. Celle-ci se base notamment sur l'utili-

sation de journaux infalsifiables. *Matthieu: TODO : Ajouter refs* Il convient alors d'étudier si l'utilisation de ces structures ne limite pas le potentiel du modèle de synchronisation par différences d'états, e.g. en interdisant la diffusion des modifications par états complets.

Un premier objectif de notre travail serait de proposer des directives sur le modèle de synchronisation à privilégier en fonction du contexte d'utilisation du CRDT.

Ce travail permettrait aussi d'étudier la combinaison des modèles de synchronisation par opérations et par différences d'états au sein d'un même CRDT. Le but serait notamment d'identifier les paramètres conduisant à privilégier un modèle de synchronisation par rapport à l'autre, de façon à permettre aux noeuds de basculer dynamiquement entre les deux.

4.2.4 Approfondissement du patron de conception de Pure Operation-Based CRDTs

BAQUERO et al. [30] proposent un framework pour concevoir des CRDTs synchronisés par opérations : Pure Operation-Based CRDTs. Ce framework a plusieurs objectifs :

- (i) Proposer une approche partiellement générique pour définir un CRDT synchronisé par opérations.
- (ii) Factoriser les métadonnées utilisées par le CRDT pour le mécanisme de résolution de conflits, notamment pour identifier les éléments, et celles utilisées par la couche livraison, notamment pour identifier les opérations.
- (iii) Inclure des mécanismes de GC de ces métadonnées pour réduire la taille de l'état.

Pour cela, les auteurs se limitent aux CRDTs purs synchronisés par opérations, c.-à-d. les CRDTs dont les modifications enrichies de leurs arguments et d'une estampille fournie par la couche de livraison des messages sont commutatives. Pour ces CRDTs, les auteurs proposent un framework générique permettant leur spécification sous la forme d'un journal partiellement ordonné des opérations, nommé PO-Log. Les auteurs associent le PO-Log à une couche de livraison Reliable Causal Broadcast (RCB) des opérations.

Les auteurs définissent ensuite le concept de stabilité causale. Ce concept leur permet de retirer les métadonnées de causalité des opérations du PO-Log lorsque celles-ci sont déterminées comme étant causalement stables.

Finalement, les auteurs définissent un ensemble de relations, spécifiques à chaque CRDT, qui permettent d'exprimer la *redondance causale*. La redondance causale permet de spécifier quand retirer une opération du PO-Log, car rendue obsolète par une autre opération.

Cette approche souffre toutefois de plusieurs limites. Tout d'abord, elle repose sur l'utilisation d'une couche de livraison RCB. Cette couche satisfait le modèle de livraison causale. Mais pour rappel, ce modèle induit l'ajout de données de causalité précises à chaque opération, sous la forme d'un vecteur de version ou d'une barrière causale. Nous jugeons ce modèle trop coûteux pour un système P2P dynamique à large échelle sujet au churn.

En plus du coût induit en termes de métadonnées et de bande-passante, le modèle de livraison causale peut aussi introduire un délai superflu dans la livraison des opérations.

En effet, ce modèle impose que tous les messages précédant un nouveau message d’après la relation *happens-before* soient eux-mêmes livrés avant de livrer ce dernier. Il en résulte que des opérations peuvent être mises en attente par la couche livraison, e.g. suite à la perte d’une de leurs dépendances d’après la relation *happens-before*, alors que leurs dépendances réelles ont déjà été livrées et que les opérations sont de fait intégrables en l’état. Plusieurs travaux [57, 58] ont noté ce problème. Pour y répondre et ainsi améliorer la réactivité du framework Pure Operation-Based, ils proposent d’exposer les opérations mises en attente par la couche livraison au CRDT. Bien que fonctionnelle, cette approche induit toujours le coût d’une couche de livraison respectant le modèle de livraison causale et nous fait considérer la raison de ce coût, le modèle de livraison n’étant dès lors plus respecté.

Ensuite, ce framework impose que la modification **prepare** ne puisse pas inspecter l’état courant du noeud. Cette contrainte est compatible avec les CRDTs pour les types de données simples qui sont considérés dans [30], e.g. le Compteur ou l’Ensemble. Elle empêche cependant l’expression de CRDTs pour des types de données plus complexes, e.g. la Séquence ou le Graphe. *Matthieu: TODO : À confirmer pour le graphe* Nous jugeons dommageable qu’un framework pour la conception de CRDTs limite de la sorte son champ d’application.

Finalement, les auteurs ne considèrent que des types de données avec des modifications à granularité fixe. Ainsi, ils définissent la notion de redondance causale en se limitant à ce type de modifications. Par exemple, ils définissent que la suppression d’un élément d’un ensemble rend obsolète les ajouts précédents de cet élément. Cependant, dans le cadre d’autres types de données, e.g. la Séquence, une modification peut concerner un ensemble d’éléments de taille variable. Une opération peut donc être rendue obsolète non pas par une opération, mais par un ensemble d’opérations. Par exemple, les suppressions d’éléments formant une sous-chaîne rendent obsolète l’insertion de cette sous-chaîne. Ainsi, la notion de redondance causale est incomplète et souffre de l’absence d’une notion d’obsolescence partielle d’une opération.

Pour répondre aux différents problèmes soulevés, nous souhaitons proposer une extension du framework Pure Operation-Based CRDTs. Nos objectifs sont les suivants :

- (i) Proposer un framework mettant en lumière la présence et le rôle de deux modèles de livraison :
 - (i) Le modèle de livraison minimal requis par le CRDT pour assurer la convergence forte à terme [6].
 - (ii) Le modèle de livraison utilisé par le système, qui doit être égal ou plus contraint que modèle de livraison minimal du CRDT. Ce second modèle de livraison est une stratégie permettant au système de respecter un modèle de cohérence donné et régissant les règles de compaction de l’état. Il peut être amené à évoluer en fonction de l’état du système et de ses besoins. Par exemple, un système peut par défaut utiliser le modèle de livraison causale pour assurer le modèle de cohérence causal. Puis, lorsque le nombre de noeuds atteint un seuil donné, le système peut passer au modèle de livraison FIFO pour assurer le modèle de cohérence PRAM afin de réduire les coûts en bande-passante.
- (ii) Étendre la notion de redondance causale pour prendre en compte la redondance partielle des opérations. De plus, nous souhaitons rendre cette notion accessible à

la couche de livraison, pour détecter au plus tôt les opérations désormais obsolètes et prévenir leur diffusion.

- (iii) Identifier et classifier les mécanismes de résolution de conflits, pour déterminer lesquels sont indépendants de l'état courant pour la génération des opérations et lesquels nécessitent d'inspecter l'état courant dans **prepare**.

Annexe A

Entrelacement d'insertions concurrentes dans Treedoc

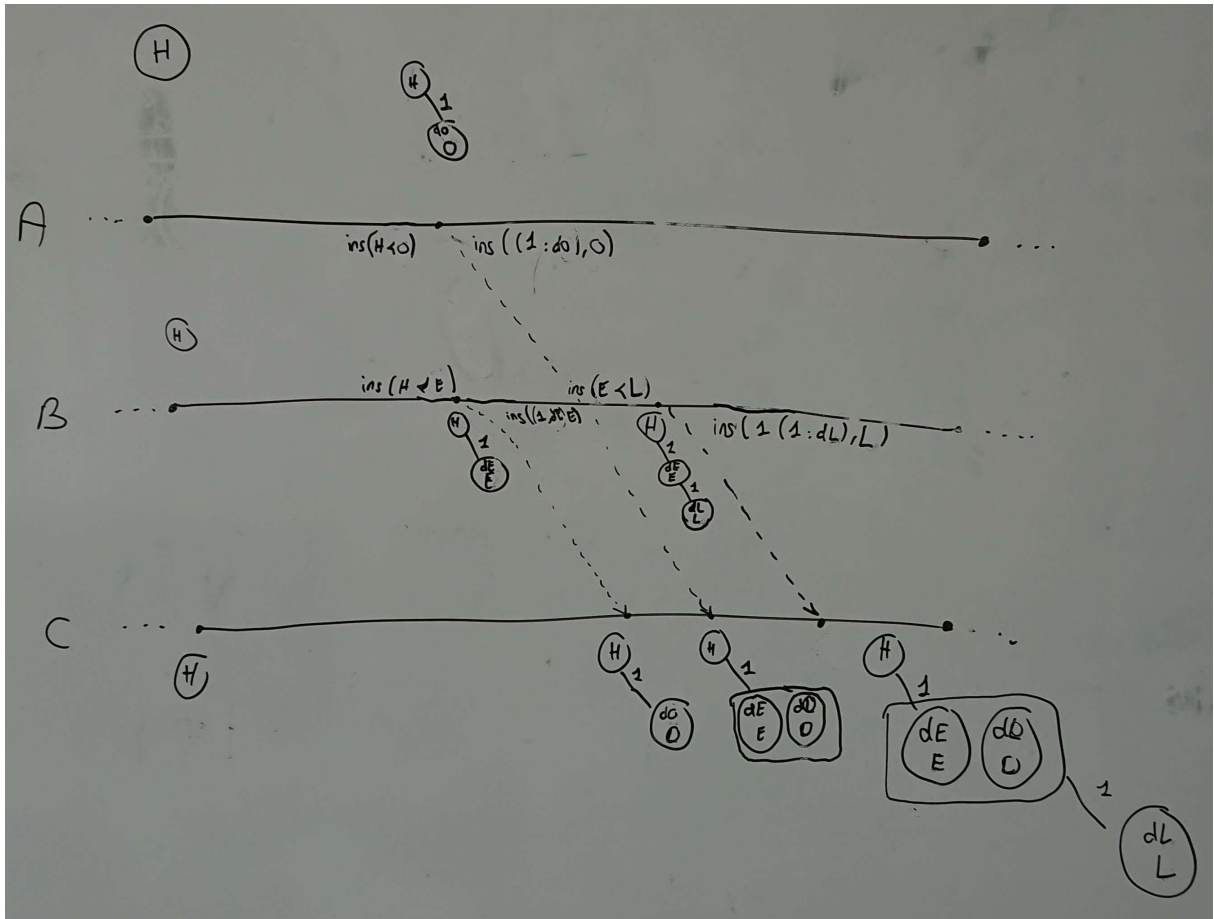


FIGURE A.1 – Modifications concurrentes d'une séquence Treedoc résultant en un entrelacement

Matthieu: TODO : Réaliser au propre contre-exemple. Nécessite que $d_E < d_O$, inverser A et B histoire d'éviter toute confusion. En soi, C pas nécessaire, à voir si le conserve.

Annexe B

Algorithmes RENAMEID

Algorithme 5 Remaining functions to rename an identifier

```
function RENIDLESTHANFIRSTID(id, newFirstId)
  if id < newFirstId then
    return id
  else
    pos ← position(newFirstId)
    nId ← nodeId(newFirstId)
    nSeq ← nodeSeq(newFirstId)
    predNewFirstId ← new Id(pos, nId, nSeq, -1)

    return concat(predNewFirstId, id)
  end if
end function

function RENIDGREATERTHANLASTID(id, newLastId)
  if id < newLastId then
    return concat(newLastId, id)
  else
    return id
  end if
end function
```

Annexe C

Algorithmes REVERTRENAMEID

Algorithme 6 Remaining functions to revert an identifier renaming

```
function REVRENIDLESTHANNEWFIRSTID(id, firstId, newFirstId)
  predNewFirstId  $\leftarrow$  createIdFromBase(newFirstId, -1)
  if isPrefix(predNewFirstId, id) then
    tail  $\leftarrow$  getTail(id, 1)
    if tail < firstId then
      return tail
    else
      ▷ id has been inserted causally after the rename op
      offset  $\leftarrow$  getLastOffset(firstId)
      predFirstId  $\leftarrow$  createIdFromBase(firstId, offset)
      return concat(predFirstId, MAX_TUPLE, tail)
    end if
  else
    return id
  end if
end function

function REVRENIDGREATERTHANNEWLASTID(id, lastId)
  if id < lastId then
    ▷ id has been inserted causally after the rename op
    return concat(lastId, MIN_TUPLE, id)
  else if isPrefix(newLastId, id) then
    tail  $\leftarrow$  getTail(id, 1)
    if tail < lastId then
      ▷ id has been inserted causally after the rename op
      return concat(lastId, MIN_TUPLE, tail)
    else if tail < newLastId then
      return tail
    else
      ▷ id has been inserted causally after the rename op
      return id
    end if
  else
    return id
  end if
end function
```

Index

Voici un index

FiXme :

Notes :

- 10 : Matthieu : TODO : Insérer refs travaux Claudia et Vinh, 71
- 11 : Matthieu : TODO : Ajouter refs, 73
- 12 : Matthieu : TODO : À confirmer pour le graphe, 74
- 13 : Matthieu : TODO : Réaliser au propre contre-exemple. Nécessite que $d_E < d_O$, inverser A et B histoire d'éviter toute confusion. En soi, C pas nécessaire, à voir si le conserve. , 77
- 1 : Matthieu : TODO : Voir si angle écologique/réduction consommation d'énergie peut être pertinent., 2
- 2 : Matthieu : NOTE : Pourrait insérer exemple de conflits de l'édition collaborative ici., 3
- 3 : Matthieu : TODO : Trouver et ajouter références, 6
- 4 : Matthieu : TODO : Vérifier du côté des applis de IPFS, 6
- 5 : Matthieu : TODO : Faire le lien avec les travaux de Burckhardt [27] et les MRDTs [28], 15
- 6 : Matthieu : TODO : Ajouter une partie sur la discussion qu'on a pu avoir avec les reviewers sur la présence de pierres tombales dans RenamableLogootSplit, et comment ces pierres tombales diffèrent de celles présentent dans WOOT et RGA. , 51

7 : Matthieu : TODO : Ajouter conclusion à cette sous-section, 59

8 : Matthieu : TODO : Revoir ce point pour indiquer que stabilité causale n'est pas une condition raisonnable dans systèmes sujets au churn. Qu'à notre sens, rend cette solution inadéquate par rapport au modèle du système. Mais préciser que majorité des noeuds dans ce type de système se connectent de manière éphémère, c.-à-d. ne reviennent jamais. Mais principe d'une collaboration est de collaborer. Si noeuds ne collaborent pas ou mal, e.g. font un truc dans leur coin pendant X mois (dépendant du cas d'application), nous paraît justifier de les retirer de la collaboration. , 61

9 : Matthieu : TODO : Insérer refs distance de Hamming, Levenstein, String-to-string correction problem (Tichy et al), 71

FiXme (Matthieu) :

Notes :

- 10 : TODO : Insérer refs travaux Claudia et Vinh, 71
- 11 : TODO : Ajouter refs, 73
- 12 : TODO : À confirmer pour le graphe, 74
- 13 : TODO : Réaliser au propre contre-exemple. Nécessite que $d_E < d_O$, inverser A et B histoire d'éviter toute confusion. En soi, C pas nécessaire, à voir si le conserve. , 77

- 1 : TODO : Voir si angle écologique/réduction consommation d'énergie peut être pertinent., 2
- 2 : NOTE : Pourrait insérer exemple de conflits de l'édition collaborative ici., 3
- 3 : TODO : Trouver et ajouter références, 6
- 4 : TODO : Vérifier du côté des applis de IPFS, 6
- 5 : TODO : Faire le lien avec les travaux de Burckhardt [27] et les MRDTs [28], 15
- 6 : TODO : Ajouter une partie sur la discussion qu'on a pu avoir avec les reviewers sur la présence de pierres tombales dans RenamableLogootSplit, et comment ces pierres tombales diffèrent de celles présentent dans WOOT et RGA. , 51
- 7 : TODO : Ajouter conclusion à cette sous-section, 59
- 8 : TODO : Revoir ce point pour indiquer que stabilité causale n'est pas une condition raisonnable dans systèmes sujets au churn. Qu'à notre sens, rend cette solution inadéquate par rapport au modèle du système. Mais préciser que majorité des noeuds dans ce type de système se connectent de manière éphémère, c.-à-d. ne reviennent jamais. Mais principe d'une collaboration est de collaborer. Si noeuds ne collaborent pas ou mal, e.g. font un truc dans leur coin pendant X mois (dépendant du cas d'application), nous paraît justifier de les retirer de la collaboration. , 61
- 9 : TODO : Insérer refs distance de Hamming, Levenstein, String-to-string correction problem (Tichy et al), 71

Bibliographie

- [1] Eric A. BREWER. « Towards Robust Distributed Systems ». In : *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA : ACM, 2000, p. 7–. ISBN : 1-58113-183-6. DOI : 10.1145/343477.343502. URL : <http://doi.acm.org/10.1145/343477.343502>.
- [2] Daniel ABADI. « Consistency Tradeoffs in Modern Distributed Database System Design : CAP is Only Part of the Story ». In : *Computer* 45.2 (2012), p. 37–42. DOI : 10.1109/MC.2012.33.
- [3] Yasushi SAITO et Marc SHAPIRO. « Optimistic Replication ». In : *ACM Comput. Surv.* 37.1 (mar. 2005), p. 42–81. ISSN : 0360-0300. DOI : 10.1145/1057977.1057980. URL : <https://doi.org/10.1145/1057977.1057980>.
- [4] Douglas B TERRY, Marvin M THEIMER, Karin PETERSEN, Alan J DEMERS, Mike J SPREITZER et Carl H HAUSER. « Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System ». In : *SIGOPS Oper. Syst. Rev.* 29.5 (déc. 1995), p. 172–182. ISSN : 0163-5980. DOI : 10.1145/224057.224070. URL : <https://doi.org/10.1145/224057.224070>.
- [5] Marc SHAPIRO et Nuno PREGUIÇA. *Designing a commutative replicated data type*. Research Report RR-6320. INRIA, 2007. URL : <https://hal.inria.fr/inria-00177693>.
- [6] Marc SHAPIRO, Nuno M. PREGUIÇA, Carlos BAQUERO et Marek ZAWIRSKI. « Conflict-Free Replicated Data Types ». In : *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. SSS 2011. 2011, p. 386–400. DOI : 10.1007/978-3-642-24550-3_29.
- [7] Mihai LETIA, Nuno PREGUIÇA et Marc SHAPIRO. « Consistency without concurrency control in large, dynamic systems ». In : *LADIS 2009 - 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*. T. 44. Operating Systems Review 2. Big Sky, MT, United States : Assoc. for Computing Machinery, oct. 2009, p. 29–34. DOI : 10.1145/1773912.1773921. URL : <https://hal.inria.fr/hal-01248270>.
- [8] Marek ZAWIRSKI, Marc SHAPIRO et Nuno PREGUIÇA. « Asynchronous rebalancing of a replicated tree ». In : *Conférence Française en Systèmes d'Exploitation (CFSE)*. Saint-Malo, France, mai 2011, p. 12. URL : <https://hal.inria.fr/hal-01248197>.
- [9] GOOGLE. *Google Docs*. URL : <https://docs.google.com/>.

- [10] Matthieu NICOLAS, Victorien ELVINGER, G  rald OSTER, Claudia-Lavinia IGNAT et Fran  ois CHAROY. « MUTE : A Peer-to-Peer Web-based Real-time Collaborative Editor ». In : *ECSCW 2017 - 15th European Conference on Computer-Supported Cooperative Work*. T. 1. Proceedings of 15th European Conference on Computer-Supported Cooperative Work - Panels, Posters and Demos 3. Sheffield, United Kingdom : EUSSET, ao  t 2017, p. 1–4. DOI : 10.18420/ecscw2017_p5. URL : <https://hal.inria.fr/hal-01655438>.
- [11] Luc ANDR  , St  phane MARTIN, G  rald OSTER et Claudia-Lavinia IGNAT. « Supporting Adaptable Granularity of Changes for Massive-Scale Collaborative Editing ». In : *International Conference on Collaborative Computing : Networking, Applications and Worksharing - CollaborateCom 2013*. Austin, TX, USA : IEEE Computer Society, oct. 2013, p. 50–59. DOI : 10.4108/icst.collaboratecom.2013.254123.
- [12] Victorien ELVINGER. « R  plication s  curis  e dans les infrastructures pair-  -pair de collaboration ». Theses. Universit   de Lorraine, juin 2021. URL : <https://hal.univ-lorraine.fr/tel-03284806>.
- [13] Matthieu NICOLAS, Gerald OSTER et Olivier PERRIN. « Efficient Renaming in Sequence CRDTs ». In : *IEEE Transactions on Parallel and Distributed Systems* 33.12 (d  c. 2022), p. 3870–3885. DOI : 10.1109/TPDS.2022.3172570. URL : <https://hal.inria.fr/hal-03772633>.
- [14] Hoang-Long NGUYEN, Claudia-Lavinia IGNAT et Olivier PERRIN. « Trusternity : Auditing Transparent Log Server with Blockchain ». In : *Companion of the The Web Conference 2018*. Lyon, France, avr. 2018. DOI : 10.1145/3184558.3186938. URL : <https://hal.inria.fr/hal-01883589>.
- [15] Hoang-Long NGUYEN, Jean-Philippe EISENBARTH, Claudia-Lavinia IGNAT et Olivier PERRIN. « Blockchain-Based Auditing of Transparent Log Servers ». In : *32th IFIP Annual Conference on Data and Applications Security and Privacy (DBSec)*. Sous la dir. de Florian KERSCHBAUM et Stefano PARABOSCHI. T. LNCS-10980. Data and Applications Security and Privacy XXXII. Part 1 : Administration. Bergamo, Italy : Springer International Publishing, juil. 2018, p. 21–37. DOI : 10.1007/978-3-319-95729-6_2. URL : <https://hal.archives-ouvertes.fr/hal-01917636>.
- [16] Abhinandan DAS, Indranil GUPTA et Ashish MOTIVALA. « SWIM : scalable weakly-consistent infection-style process group membership protocol ». In : *Proceedings International Conference on Dependable Systems and Networks*. 2002, p. 303–312. DOI : 10.1109/DSN.2002.1028914.
- [17] Armon DADGAR, James PHILLIPS et Jon CURREY. « Lifeguard : Local health awareness for more accurate failure detection ». In : *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. 2018, p. 22–25.

- [18] Brice NÉDELEC, Julian TANKE, Davide FREY, Pascal MOLLI et Achour MOSTÉFAOUI. « An adaptive peer-sampling protocol for building networks of browsers ». In : *World Wide Web* 21.3 (2018), p. 629–661.
- [19] Mike BURMESTER et Yvo DESMEDT. « A secure and efficient conference key distribution system ». In : *Advances in Cryptology — EUROCRYPT’94*. Sous la dir. d’Alfredo DE SANTIS. Berlin, Heidelberg : Springer Berlin Heidelberg, 1995, p. 275–286. ISBN : 978-3-540-44717-7.
- [20] Matthieu NICOLAS. « Efficient renaming in CRDTs ». In : *Middleware 2018 - 19th ACM/IFIP International Middleware Conference (Doctoral Symposium)*. Rennes, France, déc. 2018. URL : <https://hal.inria.fr/hal-01932552>.
- [21] Matthieu NICOLAS, Gérald OSTER et Olivier PERRIN. « Efficient Renaming in Sequence CRDTs ». In : *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC’20)*. Heraklion, Greece, avr. 2020. URL : <https://hal.inria.fr/hal-02526724>.
- [22] Rachid GUERRAOUI, Matej PAVLOVIC et Dragos-Adrian SEREDINSCHI. « Trade-offs in replicated systems ». In : *IEEE Data Engineering Bulletin* 39.ARTICLE (2016), p. 14–26.
- [23] Leslie LAMPORT. « Time, Clocks, and the Ordering of Events in a Distributed System ». In : *Commun. ACM* 21.7 (juil. 1978), p. 558–565. ISSN : 0001-0782. DOI : 10.1145/359545.359563. URL : <https://doi.org/10.1145/359545.359563>.
- [24] Nuno M. PREGUIÇA, Carlos BAQUERO et Marc SHAPIRO. « Conflict-free Replicated Data Types (CRDTs) ». In : *CoRR* abs/1805.06358 (2018). arXiv : 1805.06358. URL : <http://arxiv.org/abs/1805.06358>.
- [25] Nuno M. PREGUIÇA. « Conflict-free Replicated Data Types : An Overview ». In : *CoRR* abs/1806.10254 (2018). arXiv : 1806.10254. URL : <http://arxiv.org/abs/1806.10254>.
- [26] B. A. DAVEY et H. A. PRIESTLEY. *Introduction to Lattices and Order*. 2^e éd. Cambridge University Press, 2002. DOI : 10.1017/CB09780511809088.
- [27] Sebastian BURCKHARDT, Alexey GOTSMAN, Hongseok YANG et Marek ZAWIRSKI. « Replicated Data Types : Specification, Verification, Optimality ». In : *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’14. San Diego, California, USA : Association for Computing Machinery, 2014, p. 271–284. ISBN : 9781450325448. DOI : 10.1145/2535838.2535848. URL : <https://doi.org/10.1145/2535838.2535848>.
- [28] Gowtham KAKI, Swarn PRIYA, KC SIVARAMAKRISHNAN et Suresh JAGANNATHAN. « Mergeable Replicated Data Types ». In : *Proc. ACM Program. Lang.* 3.OOPSLA (oct. 2019). DOI : 10.1145/3360580. URL : <https://doi.org/10.1145/3360580>.
- [29] Carlos BAQUERO, Paulo Sérgio ALMEIDA et Ali SHOKER. « Making Operation-Based CRDTs Operation-Based ». In : *Distributed Applications and Interoperable Systems*. Sous la dir. de Kostas MAGOUTIS et Peter PIETZUCH. Berlin, Heidelberg : Springer Berlin Heidelberg, 2014, p. 126–140.

- [30] Carlos BAQUERO, Paulo Sergio ALMEIDA et Ali SHOKER. *Pure Operation-Based Replicated Data Types*. 2017. arXiv : 1710.04469 [cs.DC].
- [31] Claudia-Lavinia IGNAT, Gérald OSTER, Meagan NEWMAN, Valerie SHALIN et François CHAROY. « Studying the Effect of Delay on Group Performance in Collaborative Editing ». In : *Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014, Springer 2014 Lecture Notes in Computer Science*. Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014. Seattle, WA, United States, sept. 2014, p. 191–198. DOI : 10.1007/978-3-319-10831-5_29. URL : <https://hal.archives-ouvertes.fr/hal-01088815>.
- [32] Claudia-Lavinia IGNAT, Gérald OSTER, Olivia FOX, François CHAROY et Valerie SHALIN. « How Do User Groups Cope with Delay in Real-Time Collaborative Note Taking ». In : *European Conference on Computer Supported Cooperative Work 2015*. Sous la dir. de Nina BOULUS-RODJE, Gunnar ELLINGSEN, Tone BRATTETEIG, Margunn AANESTAD et Pernille BJORN. Proceedings of the 14th European Conference on Computer Supported Cooperative Work. Oslo, Norway : Springer International Publishing, sept. 2015, p. 223–242. DOI : 10.1007/978-3-319-20499-4_12. URL : <https://hal.inria.fr/hal-01238831>.
- [33] D. S. PARKER, G. J. POPEK, G. RUDISIN, A. STOUGHTON, B. J. WALKER, E. WALTON, J. M. CHOW, D. EDWARDS, S. KISER et C. KLINE. « Detection of Mutual Inconsistency in Distributed Systems ». In : *IEEE Trans. Softw. Eng.* 9.3 (mai 1983), p. 240–247. ISSN : 0098-5589. DOI : 10.1109/TSE.1983.236733. URL : <https://doi.org/10.1109/TSE.1983.236733>.
- [34] Haifeng SHEN et Chengzheng SUN. « A log compression algorithm for operation-based version control systems ». In : *Proceedings 26th Annual International Computer Software and Applications*. 2002, p. 867–872. DOI : 10.1109/CMPSAC.2002.1045115.
- [35] Claudia-Lavinia IGNAT. « Maintaining consistency in collaboration over hierarchical documents ». Thèse de doct. ETH Zurich, 2006.
- [36] Carlos BAQUERO, Paulo Sérgio ALMEIDA et Ali SHOKER. « Making Operation-Based CRDTs Operation-Based ». In : *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. PaPEC '14. Amsterdam, The Netherlands : Association for Computing Machinery, 2014. ISBN : 9781450327169. DOI : 10.1145/2596631.2596632. URL : <https://doi.org/10.1145/2596631.2596632>.
- [37] Nuno PREGUICA, Joan Manuel MARQUES, Marc SHAPIRO et Mihai LETIA. « A Commutative Replicated Data Type for Cooperative Editing ». In : *2009 29th IEEE International Conference on Distributed Computing Systems*. Juin 2009, p. 395–403. DOI : 10.1109/ICDCS.2009.20.
- [38] Brice NÉDELEC, Pascal MOLLI, Achour MOSTÉFAOUI et Emmanuel DESMONTILS. « LSEQ : an adaptive structure for sequences in distributed collaborative editing ». In : *Proceedings of the 2013 ACM Symposium on Document Engineering*. DocEng 2013. Sept. 2013, p. 37–46. DOI : 10.1145/2494266.2494278.

-
- [39] Brice NÉDELEC, Pascal MOLLI et Achour MOSTÉFAOUI. « A scalable sequence encoding for collaborative editing ». In : *Concurrency and Computation : Practice and Experience* (), e4108. DOI : 10.1002/cpe.4108. eprint : <https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4108>. URL : <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4108>.
 - [40] Stéphane WEISS, Pascal URSO et Pascal MOLLI. « Logoot : A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks ». In : *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*. Montreal, QC, Canada : IEEE Computer Society, juin 2009, p. 404–412. DOI : 10.1109/ICDCS.2009.75. URL : <http://doi.ieeeecomputersociety.org/10.1109/ICDCS.2009.75>.
 - [41] Hyun-Gul ROH, Myeongjae JEON, Jin-Soo KIM et Joonwon LEE. « Replicated abstract data types : Building blocks for collaborative applications ». In : *Journal of Parallel and Distributed Computing* 71.3 (2011), p. 354–368. ISSN : 0743-7315. DOI : <https://doi.org/10.1016/j.jpdc.2010.12.006>. URL : <http://www.sciencedirect.com/science/article/pii/S0743731510002716>.
 - [42] Gérald OSTER, Pascal URSO, Pascal MOLLI et Abdessamad IMINE. « Data Consistency for P2P Collaborative Editing ». In : *ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*. Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work. Banff, Alberta, Canada : ACM Press, nov. 2006, p. 259–268. URL : <https://hal.inria.fr/inria-00108523>.
 - [43] Victorien ELVINGER, Gérald OSTER et Francois CHAROY. « Prunable Authenticated Log and Authenticable Snapshot in Distributed Collaborative Systems ». In : *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. 2018, p. 156–165. DOI : 10.1109/CIC.2018.00031.
 - [44] Sylvie NOËL et Jean-Marc ROBERT. « Empirical study on collaborative writing : What do co-authors do, use, and like ? ». In : *Computer Supported Cooperative Work (CSCW)* 13.1 (2004), p. 63–89.
 - [45] Jim GILES. « Special Report Internet encyclopaedias go head to head ». In : *nature* 438.15 (2005), p. 900–901.
 - [46] Martin KLEPPMANN, Adam WIGGINS, Peter van HARDENBERG et Mark MCGRANAGHAN. « Local-First Software : You Own Your Data, in Spite of the Cloud ». In : *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece : Association for Computing Machinery, 2019, p. 154–178. ISBN : 9781450369954. DOI : 10.1145/3359591.3359737. URL : <https://doi.org/10.1145/3359591.3359737>.
 - [47] OPENRELAY. *OpenRelay*. URL : <https://openrelay.xyz/>.
 - [48] Protocol LABS. *IPFS*. URL : <https://ipfs.io/>.
 - [49] Friedemann MATTERN et al. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

- [50] Colin FIDGE. « Logical Time in Distributed Computing Systems ». In : *Computer* 24.8 (août 1991), p. 28–33. ISSN : 0018-9162. DOI : 10.1109/2.84874. URL : <https://doi.org/10.1109/2.84874>.
- [51] Ravi PRAKASH, Michel RAYNAL et Mukesh SINGHAL. « An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments ». In : *Journal of Parallel and Distributed Computing* 41.2 (1997), p. 190–204. ISSN : 0743-7315. DOI : <https://doi.org/10.1006/jpdc.1996.1300>. URL : <https://www.sciencedirect.com/science/article/pii/S0743731596913003>.
- [52] Martin KLEPPMANN, Victor B. F. GOMES, Dominic P. MULLIGAN et Alastair R. BERESFORD. « Interleaving Anomalies in Collaborative Text Editors ». In : *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '19. Dresden, Germany : Association for Computing Machinery, 2019. ISBN : 9781450362764. DOI : 10.1145/3301419.3323972. URL : <https://doi.org/10.1145/3301419.3323972>.
- [53] Quang Vinh DANG et Claudia-Lavinia IGNAT. « Quality Assessment of Wikipedia Articles : A Deep Learning Approach by Quang Vinh Dang and Claudia-Lavinia Ignat with Martin Vesely as Coordinator ». In : *SIGWEB Newsl.* Autumn (nov. 2016). ISSN : 1931-1745. DOI : 10.1145/2996442.2996447. URL : <https://doi.org/10.1145/2996442.2996447>.
- [54] Paulo Sérgio ALMEIDA, Ali SHOKER et Carlos BAQUERO. « Efficient State-Based CRDTs by Delta-Mutation ». In : *Networked Systems*. Sous la dir. d'Ahmed BOUAJJANI et Hugues FAUCONNIER. Cham : Springer International Publishing, 2015, p. 62–76. ISBN : 978-3-319-26850-7.
- [55] Vitor ENES, Paulo Sérgio ALMEIDA, Carlos BAQUERO et João LEITÃO. « Efficient Synchronization of State-Based CRDTs ». In : *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, p. 148–159. DOI : 10.1109/ICDE.2019.00022.
- [56] Leslie LAMPORT, Robert SHOSTAK et Marshall PEASE. « The Byzantine Generals Problem ». In : *Concurrency : The Works of Leslie Lamport*. New York, NY, USA : Association for Computing Machinery, 2019, p. 203–226. ISBN : 9781450372701. URL : <https://doi.org/10.1145/3335772.3335936>.
- [57] Jim BAUWENS et Elisa Gonzalez BOIX. « Flec : A Versatile Programming Framework for Eventually Consistent Systems ». In : *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '20. Heraklion, Greece : Association for Computing Machinery, 2020. ISBN : 9781450375245. DOI : 10.1145/3380787.3393685. URL : <https://doi.org/10.1145/3380787.3393685>.
- [58] Jim BAUWENS et Elisa Gonzalez BOIX. « Improving the Reactivity of Pure Operation-Based CRDTs ». In : *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '21. Online, United Kingdom : Association for Computing Machinery, 2021. ISBN : 9781450383387. DOI : 10.1145/3447865.3457968. URL : <https://doi.org/10.1145/3447865.3457968>.

Résumé

Afin d'assurer leur haute disponibilité, les systèmes distribués à large échelle se doivent de répliquer leurs données tout en minimisant les coordinations nécessaires entre noeuds. Pour concevoir de tels systèmes, la littérature et l'industrie adoptent de plus en plus l'utilisation de types de données répliquées sans conflits (CRDTs). Les CRDTs sont des types de données qui offrent des comportements similaires aux types existants, tel l'Ensemble ou la Séquence. Ils se distinguent cependant des types traditionnels par leur spécification, qui supporte nativement les modifications concurrentes. À cette fin, les CRDTs incorporent un mécanisme de résolution de conflits au sein de leur spécification.

Afin de résoudre les conflits de manière déterministe, les CRDTs associent généralement des identifiants aux éléments stockés au sein de la structure de données. Les identifiants doivent respecter un ensemble de contraintes en fonction du CRDT, telles que l'unicité ou l'appartenance à un ordre dense. Ces contraintes empêchent de borner la taille des identifiants. La taille des identifiants utilisés croît alors continuellement avec le nombre de modifications effectuées, aggravant le surcoût lié à l'utilisation des CRDTs par rapport aux structures de données traditionnelles. Le but de cette thèse est de proposer des solutions pour pallier ce problème.

Nous présentons dans cette thèse deux contributions visant à répondre à ce problème : (i) Un nouveau CRDT pour Séquence, *RenamableLogootSplit*, qui intègre un mécanisme de renommage à sa spécification. Ce mécanisme de renommage permet aux noeuds du système de réattribuer des identifiants de taille minimale aux éléments de la séquence. Cependant, cette première version requiert une coordination entre les noeuds pour effectuer un renommage. L'évaluation expérimentale montre que le mécanisme de renommage permet de réinitialiser à chaque renommage le surcoût lié à l'utilisation du CRDT. (ii) Une seconde version de *RenamableLogootSplit* conçue pour une utilisation dans un système distribué. Cette nouvelle version permet aux noeuds de déclencher un renommage sans coordination préalable. L'évaluation expérimentale montre que cette nouvelle version présente un surcoût temporaire en cas de renommages concurrents, mais que ce surcoût est à terme.

Mots-clés: CRDTs, édition collaborative en temps réel, cohérence à terme, optimisation mémoire, performance

Abstract

Keywords: CRDTs, real-time collaborative editing, eventual consistency, memory-wise optimisation, performance

