UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Ré-identification efficace dans les types de données répliquées sans conflit (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

Président :	Stephan Merz	
Rapporteurs :	Le rapporteur 1	de Paris
	Le rapporteur 2	
	suite	taratata
	Le rapporteur 3	
Examinateurs :	L'examinateur 1	d'ici
	L'examinateur 2	
Membres de la famille :	Mon frère	
	Ma sœur	

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

Mis en page avec la classe thesul.

Remerciements

Les remerciements.

Je dédie cette thèse
à ma machine.
Oui, à Pandore,
qui fut la première de toutes.

Sommaire

Introduction **1**

1 Contexte 1
2 Questions de recherche 1
3 Contributions 1
4 Plan du manuscrit 1

Chapitre 1
État de l'art **3**

1.1 Systèmes distribués 3
1.2 Transformées opérationnelles 4
1.3 Séquences répliquées sans conflits 5
1.3.1 Types de données répliquées sans conflits 5
1.3.2 Approches pour les séquences répliquées sans conflits 7
1.4 LogootSplit 7
1.4.1 Identifiants 7
1.4.2 Aggrégation dynamique d'élements en blocs 8
1.4.3 Modèle de données 9
1.4.4 Modèle de livraison 11
1.4.5 Limites 13
1.5 Mitigation du surcoût des séquences répliquées sans conflits 14
1.6 Synthèse 15

Chapitre 2
Renommage dans une séquence répliquée **17**

2.1 Présentation de l'approche 18
2.1.1 Modèle du système 18
2.1.2 Définition de l'opération de renommage 18

2.2 RenamableLogootSplit 20
2.2.1 Opération de renommage proposée 20
2.2.2 Gestion des opérations concurrentes au renommage 21
2.2.3 Évolution du modèle de livraison des opérations 24
2.3 RenamableLogootSplit v2 26
2.3.1 Conflits en cas de renommages concurrents 26
2.3.2 Relation de priorité entre renommages 27
2.3.3 Algorithme d'annulation de l'opération de renommage 28
2.3.4 Processus d'intégration d'une opération 31
2.3.5 Règles de récupération de la mémoire des états précédents 35
2.4 Validation 38
2.4.1 Preuve de correction de RENAMEID 38
2.4.2 Complexité en temps des opérations 38
2.4.3 Expérimentations 42
2.4.4 Résultats 43
2.5 Discussion 49
2.5.1 Stockage des états précédents sur disque 49
2.5.2 Compression et limitation de la taille de l'opération *rename* 49
2.5.3 Définition de relations de priorité pour minimiser les traitements . . 50
2.5.4 Report de la transition vers la nouvelle epoch principale 50
2.5.5 Utilisation de l'opération de renommage comme mécanisme de compression du log d'opérations 51
2.5.6 Implémentation alternative de l'intégration de l'opération *rename* basée sur le log d'opérations 53
2.6 Comparaison avec les approches existantes 55
2.6.1 Core-Nebula 55
2.6.2 LSEQ 55
2.6.3 Eager stability determination 55
2.7 Conclusion 55

Chapitre 3
MUTE, un éditeur web collaboratif P2P temps réel 57

3.1 Présentation 57
3.1.1 Objectifs 58
3.1.2 Architecture 58

3.2 Couche interface . . . 58
3.3 Couche réplication . . . 59
3.3.1 Modèle de données du document texte . . . 59
3.3.2 Module de livraison des opérations . . . 59
3.3.3 Métadonnées . . . 62
3.3.4 Collaborateurs . . . 62
3.3.5 Curseurs . . . 63
3.4 Couche sécurité . . . 63
3.4.1 Objectifs . . . 63
3.4.2 Approche choisie . . . 64
3.4.3 Limites . . . 64
3.4.4 Perspectives . . . 64
3.5 Couche réseau . . . 65
3.5.1 Netflux . . . 65
3.5.2 Pulsar . . . 65
3.6 Pistes d'amélioration et de recherche . . . 66
3.6.1 Fusion de versions distantes d'un document collaboratif . . . 66
3.6.2 Rôles et places des bots dans systèmes collaboratifs . . . 66
3.7 Conclusion . . . 66

Chapitre 4
Conclusions et perspectives **67**

4.1 Résumé des contributions . . . 67
4.2 Perspectives . . . 67
4.2.1 Définition de relations de priorité pour minimiser les traitements . . . 67
4.2.2 Redéfinition de la sémantique du renommage en déplacement d'éléments . . . 67
4.2.3 Définition de types de données répliquées sans conflits plus complexes 67

Annexe A
Algorithmes renameId

Annexe B
Algorithmes revertRenameId

Index **73**

Bibliographie

Table des figures

1.1 Representation of a LogootSplit sequence containing the elements "HLO" . 9
1.2 TODO . . . 10
1.3 TODO . . . 11
1.4 TODO . . . 12
1.5 Insertion leading to longer identifiers . . . 14

2.1 Renaming the sequence on node A . . . 20
2.2 Concurrent update leading to inconsistency . . . 21
2.3 Main functions to rename an identifier . . . 23
2.4 Renaming concurrent update using RENAMEID before applying it to maintain intended order . . . 23
2.5 TODO . . . 24
2.6 TODO . . . 25
2.7 Concurrent *rename* operations leading to divergent states . . . 26
2.8 The *epoch tree* corresponding to the scenario of Figure 2.7 . . . 26
2.9 Selecting target epoch from execution with concurrent *rename* operations . 28
2.10 Main functions to revert an identifier renaming . . . 29
2.11 Reverting a previously applied *rename* operation . . . 29
2.12 TODO . . . 31
2.13 TODO . . . 32
2.14 TODO . . . 34
2.15 TODO . . . 35
2.16 Garbage collecting epochs and corresponding *former states* . . . 36
2.17 Evolution of the size of the document . . . 44
2.18 Integration time of standard operations . . . 45
2.19 Évolution du temps nécessaire pour rejouer le log d'opérations . . . 47

A.1 Remaining functions to rename an identifier . . . 69

B.1 Remaining functions to revert an identifier renaming . . . 72

Introduction

1 Contexte

2 Questions de recherche

3 Contributions

4 Plan du manuscrit

Chapitre 1

État de l'art

Sommaire

1.1 Systèmes distribués 3
1.2 Transformées opérationnelles 4
1.3 Séquences répliquées sans conflits 5
1.3.1 Types de données répliquées sans conflits 5
1.3.2 Approches pour les séquences répliquées sans conflits 7
1.4 LogootSplit 7
1.4.1 Identifiants 7
1.4.2 Aggrégation dynamique d'élements en blocs 8
1.4.3 Modèle de données 9
1.4.4 Modèle de livraison 11
1.4.5 Limites 13
1.5 Mitigation du surcoût des séquences répliquées sans conflits 14
1.6 Synthèse 15

1.1 Systèmes distribués

— Contexte des systèmes distribués à large échelle

— Réplique les données afin de pouvoir supporter les pannes

— Adopte le paradigme de la réplication optimiste [45]

— Autorise les noeuds à consulter et à modifier la donnée sans aucune coordination entre eux

— Autorise alors les noeuds à diverger temporairement

— Permet d'être toujours disponible, de toujours répondre aux requêtes même en cas de partition réseau

— Permet aussi, en temps normal, de réduire le temps de réponse (privilégie la latence) [1]

— Comme ce modèle autorise les noeuds à modifier la donnée sans se coordonner, possible d'effectuer des modifications concurrentes

— Généralement, un mécanisme de résolution de conflits est nécessaire afin d'assurer la convergence des noeuds dans une telle situation

— Plusieurs approches ont été proposées pour implémenter un tel mécanisme

1.2 Transformées opérationnelles

— Approche permettant de gérer des modifications concurrentes sur un type de données

— Consiste à transformer les opérations par rapport aux effets des opérations concurrentes pour rendre les rendre commutatives. Permet de rendre l'ordre d'intégration des opérations sans importance par rapport à l'état final obtenu

— Se décompose en 2 parties : algorithmes (génériques) et fonctions de transformations (spécifiques au type de données)

— Plusieurs algorithmes OT adoptent une architecture centralisée (trouver citations)

— Cette architecture pose des problèmes de performances (bottleneck), sécurité (SPOF), coût, d'utilisabilité (mode offline), pérennité (disparition du service), vie privée et de résistance à la censure.

— Pour ces raisons, des algorithmes reposant sur une architecture décentralisée ont été proposés

— Mais ne règlent qu'en partie ces limites

— Notamment, ne sont pas adaptés à des systèmes P2P dynamiques

— Besoin de vector clocks sur chaque opération pour détecter la concurrence. Vector clocks adaptés dans systèmes à nombre de pairs fixe, mais pas aux systèmes dynamiques (revoir causal barrier pour p-e nuancer ce propos).

— Néanmoins, cette approche a permis de démocratiser les systèmes collaboratifs via son adoption par des services tels que Google Docs, Overleaf, Framapad

— De plus, dans le cadre de ces travaux, ont été définies les propriétés CCI [49].

— Remettre en question la propriété Causalité des CCI. Généralement, confond causalité et happen-before et exprime en finalité une contrainte trop forte. Cette contrainte peut réduire la réactivité du système (exemple avec 2 insertions sans liens mais qui force d'attendre la 1ère pour intégrer la 2nde). Causalité pose aussi des problèmes de passage à l'échelle car repose sur vector clocks. IMO, doit relaxer cette propriété pour pouvoir construire systèmes à large échelle.

Matthieu: TODO : Mentionner TP1 et TP2
Matthieu: TODO : Spécification faible et forte des séquences répliquées

1.3 Séquences répliquées sans conflits

1.3.1 Types de données répliquées sans conflits

Principes

- Nouvelles spécifications des types de données existants
- Structures conçues pour être répliquées au sein d'un système
- Et être modifiées sans coordination par ses différents noeuds
- Doivent donc supporter de nouveaux scénarios uniquement possible dans des exécutions parallèles
- Et définir une sémantique pour ces scénarios inédits
  - Exemple du Registre avec LWW-Register et MV-Register ?
- Pour gérer ces scénarios, intègrent un mécanisme de résolution de conflits directement au sein de leur spécification
- Garantissent la cohérence forte à terme

Matthieu: Faire le lien avec les travaux de Burckhardt [14] et les MRDTs [26]

Familles de types de données répliquées sans conflits

- Une catégorisation des CRDTs a été proposée
- Propose de répartir les CRDTs en différentes familles en fonction de la méthode de synchronisation utilisée
- Chacune de ces méthodes de synchronisation implique des contraintes sur la couche réseau du système et entraîne des répercussions sur la structure de données elle-même
- Types de données répliquées sans conflits à base d'états [47, 46]
  - Les noeuds partagent leur état de manière périodique
  - Une fonction *merge* permet aux noeuds de fusionner leur état courant avec un autre état reçu
  - Aucune hypothèse sur la partie réseau autre que les noeuds arrivent à communiquer à terme
  - Pas un problème si états perdus, les prochains intégreront les informations de ces derniers
  - Pas un problème si états reçus dans le désordre, la fonction *merge* est commutative
  - Pas un problème si états reçus plusieurs fois, *merge* est idempotent
  - Mais nécessite de conserver au sein de la structure de données assez d'informations pour proposer une telle fonction de *merge*
  - Par exemple, besoin de conserver une trace des éléments supprimés pour empêcher leur réapparition suite à une fusion d'états

  - *Matthieu: TODO : Ajouter forces, faiblesses et cas d'utilisation de cette approche*
- Types de données répliquées sans conflits à base d'opérations [47, 46, 8, 7]
  - Les noeuds partagent uniquement des opérations représentant leurs modifications
  - Une modification peut se formaliser en deux étapes
  - *prepare*, qui permet de générer une opération correspondant à une modification
  - *effect*, qui permet d'appliquer l'effet de la modification à un état
  - Les opérations concurrentes doivent être commutatives pour assurer la convergence
  - Mais pas de contraintes sur les opérations causalement liées
  - Pas de contraintes non plus sur l'idempotence des opérations
  - Nécessite donc généralement d'ajouter une couche *livraison* pour faire le lien entre le réseau et le CRDT
  - Permet d'attacher des informations de causalité aux opérations locales avant de les envoyer
  - Permet de ré-ordonner et filtrer les opérations distantes reçues avant de les fournir au CRDT
  - Besoin d'un mécanisme d'anti-entropie [41] pour assurer que l'ensemble des noeuds observent l'ensemble des opérations et ainsi garantir la convergence *Matthieu: TODO : Ajouter référence mécanisme d'anti-entropie basé sur Merkle Tree*
  - Permet de lisser la consommation réseau
  - Offre des temps d'intégration et de propagation des modifications rapides
  - Mais accumule des métadonnées puisque les noeuds doivent conserver les opérations passées pour permettre à un nouveau noeud de rejoindre la collaboration et de se synchroniser
  - Possible de tronquer le log des opérations en se basant sur la stabilité causale [9] afin de limiter cette accumulation de métadonnées
- Types de données répliquées sans conflits à base de différences [4, 3]

Adoption dans la littérature et l'industrie

- Conception et développement de librairies mettant à disposition des développeurs d'applications des types de données composés [36, 35, 57, 27, 6]
- Conception de langages de programmation intégrant des CRDTs comme types primitifs, destinés au développement d'applications distribuées [29, 20]
- Conception et implémentation de bases de données distribuées, relationnelles ou non, privilégiant la disponibilité et la minimisation de la latence à l'aide des CRDTs [43, 17, 55, 16, 59]

— Conception d'un nouveau paradigme d'applications, Local-First Software, dont une des fondations est les CRDTs [28, 24]
— Éditeurs collaboratifs temps réel à large échelle et offrant de nouveaux scénarios de collaboration grâce aux CRDTs [31, 39]

1.3.2 Approches pour les séquences répliquées sans conflits

Approche à pierres tombales

— WOOT [40, 54, 2]
— RGA [44]
— RGASplit [13]

Approche à identifiants densément ordonnés

— Treedoc [42]
— Logoot [52, 53]

Matthieu: NOTE : Ajouter LogootSplit de manière sommaire aussi à cet endroit ?
Matthieu: TODO : Autres Sequence CRDTs à considérer : String-wise CRDT [58], Chronofold [23]

1.4 LogootSplit

LogootSplit [5] est l'état de l'art des séquences répliquées à identifiants densément ordonnés. Comme expliqué précédemment, LogootSplit utilise des identifiants provenant d'un ordre total dense pour positionner les éléments dans la séquence répliquée.

1.4.1 Identifiants

Pour ce faire, LogootSplit assigne des identifiants composés d'une liste de tuples aux éléments. Les tuples sont définis de la manière suivante :

Définition 1 (Tuple) *Un* Tuple *est un quadruplet* $\langle position, nodeId, nodeSeq, offset \rangle$ *où*

— *position incarne la position souhaitée de l'élément.*
— *nodeId est l'identifiant unique du noeud qui a généré le tuple.*
— *nodeSeq est le numéro de séquence courant du noeud à la génération du tuple.*
— *offset indique la position de l'élément au sein d'un bloc. Nous reviendrons plus en détails sur ce composant dans la sous-section 1.4.2.*

Matthieu: TODO : Ajouter une relation d'ordre sur les tuples

Dans ce manuscrit, nous représentons les tuples par le biais de la notation suivante : $position_{offset}^{nodeId\ nodeSeq}$ où *position* est une lettre minuscule, *nodeId* une lettre majuscule et *nodeSeq* et *offset* des entiers, e.g. i_0^{B0}.

À partir de là, les identifiants LogootSplit sont définis de la manière suivante :

Définition 2 (Identifiant) *Un* Identifiant *est une liste de* Tuples.

Matthieu: TODO : Définir la notion de base (et autres fonctions utiles sur les identifiants ? genre isPrefix, concat, getTail...)

Nous représentons les identifiants en listant les tuples qui les composent. Par exemple, l'identifiant composé des tuples $\langle\langle$i, B, 0, 0$\rangle\langle$f, A, 0, 0$\rangle\rangle$ est présenté de la manière suivante : $i_0^{B0} f_0^{A0}$.

Les identifiants ont pour rôle d'ordonner les éléments relativement les uns par rapport aux autres. Pour ce faire, une relation d'ordre totale aux identifiants est associée à l'ensemble des identifiants :

Définition 3 (Relation $<_{id}$) *La relation $<_{id}$ est un ordre total strict sur l'ensemble des identifiants. Elle permet aux noeuds de comparer n'importe quelle paire d'identifiants. Elle est définie en utilisant l'ordre lexicographique sur les composants des différents tuples des identifiants comparés.*

- En utilisant cette relation d'ordre, les noeuds peuvent ordonner les éléments grâce à leur identifiant.
- Par exemple, déterminent que $i_0^{A1} <_{id} i_0^{B0}$ car les positions sont identiques et que le *nodeId* (A) du premier est plus petit que le *nodeId* (B) du second
- et que $i_0^{B0} <_{id} i_0^{B0} f_0^{A0}$ car le premier est un préfixe du second

Matthieu: TODO : Montrer que cet ensemble d'identifiants est un ensemble dense

1.4.2 Aggrégation dynamique d'élements en blocs

Au lieu de stocker les identifiants de chaque élément de la séquence, LogootSplit propose d'aggréger de façon dynamique les éléments dans des blocs. Pour cela, LogootSplit introduit la notion d'interval d'identifiants :

Définition 4 (IdInterval) *Un* IdInterval *est un couple $\langle$idBegin, offsetEnd$\rangle$ où*

- *idBegin est l'identifiant du premier élément de l'interval.*
- *offsetEnd est l'offset du dernier identifiant de l'interval.*

Les intervals d'identifiants permettent à LogootSplit d'assigner logiquement un identifiant à un ensemble d'éléments, tout en ne stockant réellement que l'identifiant de son premier élément et le dernier offset de son dernier élément.

LogootSplit regroupe les éléments avec des identifiants *contigus* dans un interval. Nous appelons *contigus* deux identifiants qui partagent une même base (c.-à-d. qui sont identiques à l'exception de leur dernier offset) et dont les *offsets* sont consécutifs. Nous représentons un interval d'identifiants à l'aide du formalisme suivant : $position_{begin..end}^{nodeId\ \ nodeSeq}$ où *begin* est l'offset du premier identifiant de l'interval et *end* du dernier.

Les blocs permettent d'associer un interval d'identifiants aux éléments correspondant. Les blocs sont définis de la manière suivante :

Définition 5 (Bloc) *Un* Bloc *est un quadruplet* ⟨*idInterval, elts, isAppendable, isPrependable*⟩ *où*

- *idInterval est l'interval d'identifiants formant le bloc*
- *elts sont les éléments contenus dans le bloc*
- *isAppendable (resp. isPrependable) est un booléen indiquant si l'auteur du bloc peut ajouter un nouvel élément en fin (resp. début) de bloc*

La Figure 1.1 présente un exemple de séquence LogootSplit : dans la 1.1a, les identifiants i_0^{B0}, i_1^{B0}, i_2^{B0} forment une chaîne d'identifiants contigus. LogootSplit est donc capable de regrouper ces éléments en un bloc représentant l'interval d'identifiants $i_{0..2}^{B0}$ pour minimiser les métadonnées stockées, comme montré dans la 1.1b.

(a) Elements with their corresponding identifiers (b) Elements grouped into a block

FIGURE 1.1 – Representation of a LogootSplit sequence containing the elements "HLO"

Cette fonctionnalité réduit le nombre d'identifiants stockés au sein de la structure de données, puisque les identifiants sont conservés à l'échelle des blocs plutôt qu'à l'échelle de chaque élément. Ceci permet de réduire de manière significative le surcoût en métadonnées de la structure de données. L'utilisation de blocs améliore aussi les performances de la structure de données. En effet, l'utilisation de blocs permet de parcourir plus efficacement la structure de données. Les blocs permettent aussi d'effectuer des modifications à l'échelle de la chaîne de caractères et non plus seulement caractère par caractère.

Matthieu: TODO : indiquer que le couple ⟨nodeId, nodeSeq⟩ permet d'identifier de manière unique la base d'un bloc ou d'un identifiant

Notons que pour une séquence donnée, nous pouvons identifier chacun de ses identifiants par le triplet ⟨nodeId, nodeSeq, offset⟩ issue de leur dernier Tuple. Par exemple, le triplet ⟨B, 0, 2⟩ désigne de manière unique l'identifiant i_2^{B0} dans Figure 1.1.

1.4.3 Modèle de données

ANDRÉ et al. [5] définissent une séquence LogootSplit de la manière suivante :

Définition 6 (Séquence LogootSplit) *Une séquence* Séquence LogootSplit *est un triplet* ⟨*nodeId, nodeSeq, blocks*⟩ *où*

- *nodeId est l'identifiant du noeud.*
- *nodeSeq est le numéro de séquence courant du noeud.*
- *blocks est une liste de Blocs correspondant à l'état actuel de la séquence répliquée.*

Plusieurs fonctions sont définies sur cette structure de données et permettent de l'interroger et de la modifier :

— ins(S, index, elts) permet d'insérer les éléments elts à la position index dans la séquence S. Cette fonction génère et associe un interval d'identifiants valide aux éléments insérés Elle retourne une opération *insert* permettant aux autres noeuds d'intégrer la modification à leur état.

Définition 7 (insert) *Une opération* insert *est un couple* $\langle id, elts \rangle$ *où*

— *id est l'identifiant du premier élément inséré par cette opération.*

— *elts est la liste des éléments insérés par cette opération.*

— rem(S, index, length) permet de supprimer length éléments à partir la position index dans la séquence S. Cette fonction répertorie les éléments supprimés sous la forme d'intervals d'identifiants. Elle retourne une opération *remove* permettant aux autres noeuds d'intégrer la modification à leur état.

Définition 8 (remove) *Une opération* remove *est une liste de IdInterval où chaque idInterval désigne un ensemble d'éléments à supprimer.*

Nous présentons dans la Figure 1.2 un exemple d'utilisation de cette séquence répliquée.

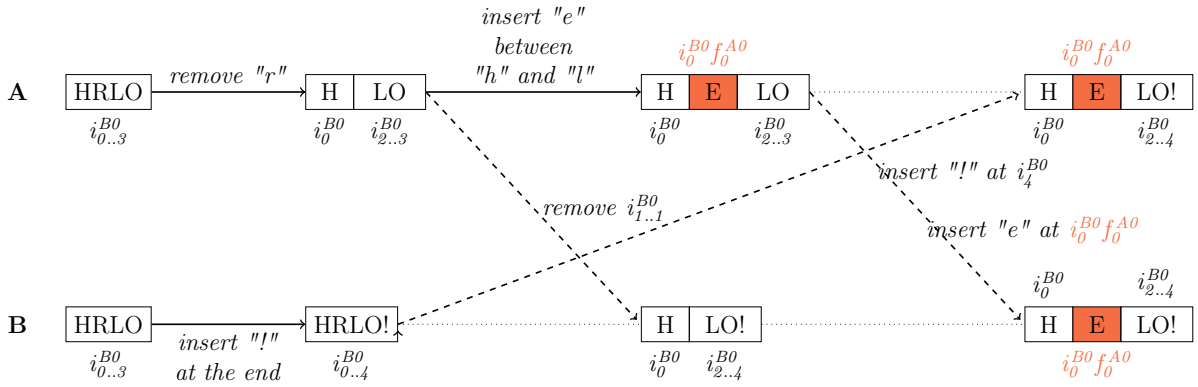


FIGURE 1.2 – TODO

Dans cet exemple, deux noeuds A et B répliquent et éditent collaborativement un document texte en utilisant LogootSplit. Ils partagent initialement le même état : une séquence composée d'un seul bloc associant les identifiants $i_{0..3}^{B0}$ aux éléments "HRLO". Les noeuds se mettent ensuite à éditer le document.

Le noeud A commence par supprimer l'élément "r" de la séquence. LogootSplit génère l'opération *remove* correspondante en utilisant l'identifiant de l'élément supprimé (i_1^{B0}). Cette opération est envoyée au noeud B pour qu'il intègre cette modification.

Le noeud A insère ensuite un élément "e" dans la séquence, entre le "h" et le "l". LogootSplit doit alors générer un identifiant *id* à associer à ce nouvel élément. Ce nouvel identifiant *id* doit respecter la contrainte suivante : $i_0^{B0} <_{id} id <_{id} i_2^{B0}$. Cependant, LogootSplit ne peut pas générer un identifiant composé d'un seul tuple respectant cet ordre. LogootSplit génère alors *id* en recopiant le premier tuple (i_0^{B0}) et en y ajoutant un nouveau

tuple (f_0^{A0}). LogootSplit génère l'opération *insert* correspondante, indiquant l'élément à insérer et sa position grâce à son identifiant. Cette opération est ensuite diffusée sur le réseau.

En parallèle, le noeud B insère un élément "!" à la fin de la séquence. Comme le noeud B est l'auteur du bloc $i_{0..3}^{B0}$, il peut y ajouter de nouveaux éléments. LogootSplit associe donc l'identifiant i_4^{B0} à l'élément "!" et l'ajoute au bloc existant.

Les noeuds se synchronisent ensuite. Le noeud A reçoit l'opération *insert* de l'élément "!" à la position i_4^{B0}. Le noeud A détermine que cet élément doit être inséré à la fin de la séquence (puisque $i_3^{B0} <_{id} i_4^{B0}$) et qu'il peut être ajouté au bloc $i_{2..3}^{B0}$ (puisque i_3^{B0} et i_4^{B0} sont contigus).

De son côté, le noeud B reçoit tout d'abord l'opération *remove* des éléments identifiés par l'interval $i_{1..1}^{B0}$, c.-à-d. l'élément attaché à l'identifiant i_1^{B0}. Le noeud B supprime donc l'élément "r" de son état.

Il reçoit ensuite l'opération *insert* de l'élément "e" à la position $i_0^{B0} f_0^{A0}$. Le noeud B insère cet élément entre les éléments "h" et "l" (puisque $i_0^{B0} <_{id} i_0^{B0} f_0^{A0} <_{id} i_2^{B0}$), respectant ainsi l'intention du noeud A.

Matthieu: NOTE : Pourrait définir dans cette sous-section la notion de séquence bien-formée

1.4.4 Modèle de livraison

Afin de garantir son bon fonctionnement, LogootSplit doit être associé à une couche de livraison de messages garantissant plusieurs propriétés.

Livraison des opérations en exactement un exemplaire

Tout d'abord, la couche de livraison de messages doit assurer que toutes les opérations soient délivrées aux noeuds, mais qu'une seule et unique fois. La Figure 1.3 représente un exemple illustrant la nécessité de cette contrainte.

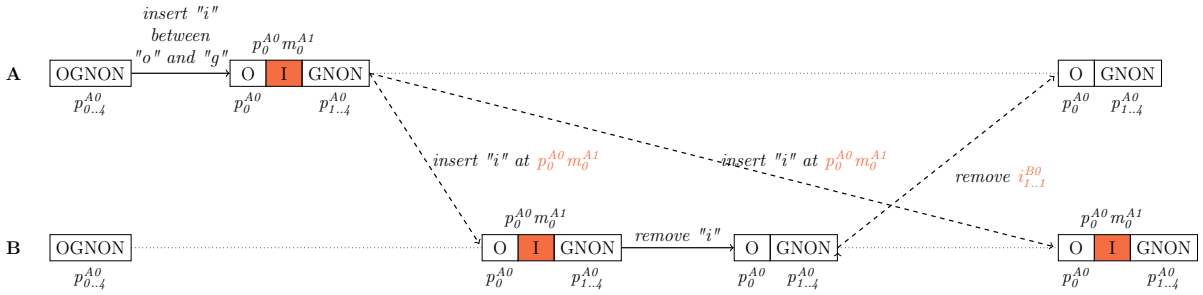


FIGURE 1.3 – TODO

Dans cet exemple, deux noeuds A et B répliquent et éditent collaborativement une séquence. La séquence répliquée contient initialement les éléments "ognon", qui sont associés à l'interval d'identifiants $p_{0..4}^{A0}$.

Le noeud A commence par insérer un nouvel élément, "i", dans la séquence entre les éléments "o" et "g". L'opération *insert* résultante, insérant l'élément "i" à la position $p_0^{A0} m_0^{A1}$, est diffusée au noeud B.

À la réception de l'opération *insert*, le noeud B l'intègre à son état. Puis il supprime dans la foulée ce nouvel élément. L'opération *remove* générée est envoyée au noeud A.

Le noeud A intègre l'opération *remove*, ce qui a pour effet de supprimer l'élément "i" associé à l'identifiant $p_0^{A0} m_0^{A1}$. Il obtient alors un état équivalent à celui du noeud B.

Cependant, l'opération *insert* insérant l'élément "i" à la position $p_0^{A0} m_0^{A1}$ est de nouveau délivrée au noeud B. De multiples raisons peuvent être à l'origine de cette nouvelle livraison : perte du message d'*acknowledgment*, utilisation d'un protocole de diffusion épidémique des messages, déclenchement du mécanisme d'anti-entropie en concurrence... Le noeud B ré-intègre alors l'opération *insert*, ce qui fait revenir l'élément "i" et l'identifiant associé. L'état du noeud B diverge désormais de celui-ci du noeud A.

Pour se prémunir de ce type de scénarios, LogootSplit requiert que la couche de livraison des messages assure une livraison en exactement un exemplaire des opérations. Cette contrainte permet d'éviter que d'anciens éléments et identifiants ressurgissent après leur suppression chez certains noeuds uniquement à cause d'une livraison multiple de l'opération *insert* correspondante.

Matthieu: QUESTION : Ajouter quelques lignes ici sur comment faire ça en pratique (Ajout d'un dot aux opérations, maintien d'un dot store au niveau de la couche livraison, vérification que dot pas encore présent dans dot store avant de passer opération à la structure de données) ? Ou je garde ça pour le chapitre sur MUTE ?

Livraison de l'opération *remove* après l'opération *insert*

Une autre propriété que doit assurer la couche de livraison de messages est que les opérations *remove* doivent être livrées au Conflict-free Replicated Data Type (CRDT) après les opérations *insert* correspondantes. La Figure 1.4 présente un exemple justifiant cette contrainte.

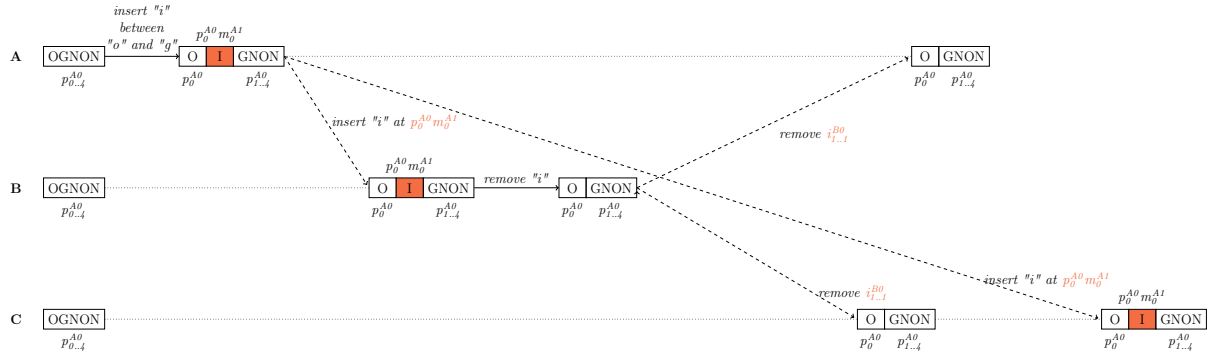


FIGURE 1.4 – TODO

Dans cet exemple, trois noeuds A, B et C répliquent et éditent collaborativement une séquence. Le noeud A commence par insérer un nouvel élément, "i", dans la séquence entre les éléments "o" et "g". L'opération *insert* résultante, insérant l'élément "i" à la position $p_0^{A0} m_0^{A1}$, est diffusée aux autres noeuds.

À la réception de l'opération *insert*, le noeud B l'intègre à son état. Cependant, le noeud B supprime dans la foulée l'élément nouvellement ajouté. Il diffuse ensuite l'opération *remove* générée.

Toutefois, suite à un aléa du réseau, l'opération *remove* supprimant l'élément "i" est livrée au noeud C avant l'opération *insert* l'ajoutant à son état. Lorsque le noeud C reçoit l'opération *remove*, il parcourt son état à la recherche de l'élément "i" pour le supprimer. Cependant, celui-ci n'est pas présent dans son état courant. L'intégration de l'opération s'achève donc sans effectuer de modification.

Le noeud C reçoit ensuite l'opération *insert*. Le noeud C intègre ce nouvel élément dans la séquence en utilisant son identifiant ($p_0^{A0} <_{id} p_0^{A0} m_0^{A1} <_{id} p_1^{A0}$).

Ainsi, l'état du noeud C diverge de celui-ci des autres noeuds à terme, et cela malgré que les noeuds A, B et C aient intégré le même ensemble d'opérations. Ce résultat transgresse la propriété de Cohérence forte à terme (SEC) que doivent assurer les CRDTs. Afin d'empêcher ce scénario de se produire, LogootSplit impose donc la livraison causale des opérations *remove* par rapport aux opérations *insert* correspondantes.

Matthieu: QUESTION : Même que pour la exactly-once delivery, est-ce que j'explique ici comment assurer cette contrainte plus en détails (Ajout des dots des opérations insert *en dépendances de l'opération* remove*, vérification que dots présents dans dot store avant de passer l'opération* remove *à la structure de données) ou je garde ça pour le chapitre sur MUTE ?*

Définition du modèle de livraison

Pour résumer, la couche de livraison des opérations associée à LogootSplit doit respecter le modèle de livraison suivant :

Définition 9 (Exactly-once + Causal remove) *Le modèle de livraison* Exactly-once + Causal remove *définit les 3 règles suivantes sur la livraison des opérations :*

1. *Une opération doit être délivrée à l'ensemble des noeuds à terme,*
2. *Une opération doit être délivrée qu'une seule et unique fois aux noeuds,*
3. *Une opération* remove *doit être délivrée à un noeud une fois que les opérations* insert *des éléments concernés par la suppression ont été délivrées à ce dernier.*

Il est à noter que Elvinger [21] a récemment proposé dans ses travaux de thèse Dotted LogootSplit, un nouveau Sequence CRDT basée sur les différences. Inspiré de Logoot et LogootSplit, ce nouveau CRDT associe une séquence à identifiants densément ordonnés à un contexte causal. Le contexte causal est une structure de données permettant à Dotted LogootSplit de représenter et de maintenir efficacement les informations des modifications déjà intégrées à l'état courant. Cette association permet à Dotted LogootSplit de fonctionner de manière autonome, sans imposer de contraintes particulières à la couche livraison autres que la livraison à terme.

1.4.5 Limites

Comme indiqué précédemment, la taille des identifiants provenant d'un ordre total dense est variable. Quand les noeuds insèrent de nouveaux éléments entre deux autres

ayant la même valeur de *position*, LogootSplit n'a pas d'autre choix que d'augmenter la taille de l'identifiant résultant. La Figure 1.5 illustre de tels cas. Dans cet exemple, puisque le noeud A insère un nouvel élément entre deux identifiants contigus i_0^{B0} et i_1^{B0}, LogootSplit ne peut pas générer un identifiant adapté de la même taille. Pour respecter l'ordre souhaité, LogootSplit génère un identifiant en ajoutant un nouveau tuple à l'identifiant du prédecesseur : $i_0^{B0} f_0^{A0}$.

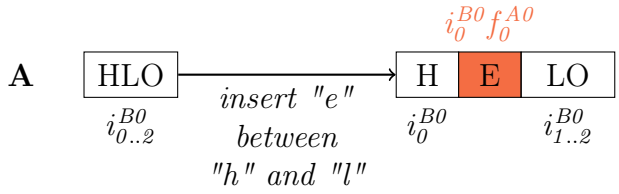


FIGURE 1.5 – Insertion leading to longer identifiers

Par conséquent, la taille des identifiants a tendance à croître alors que le système progresse. Cette croissance impacte négativement les performances de la structure de données sur plusieurs aspects. Puisque les identifiants attachés aux éléments deviennent plus long, le surcoût en métadonnées augmente. Ceci augmente aussi la consommation en bande-passante puisque les noeuds doivent diffuser les identifiants aux autres.

Matthieu: TODO : Ajouter une phrase pour expliquer que la croissance des identifiants impacte aussi le temps d'intégration des modifications

De plus, le nombre de blocs composant la séquence répliquée augmente au fil du temps. En effet, plusieurs contraintes sur la génération d'identifiants empêchent les noeuds d'ajouter des nouveaux éléments aux blocs existants. Par exemple, seul le noeud qui a généré un bloc peut ajouter un élément à ce dernier. Ces limitations provoquent la génération de nouveau blocs. La séquence se retrouve finalement fragmentée en de nombreux blocs de seulement quelques caractères chacun. Cependant, aucun mécanisme pour fusionner les blocs à posteriori n'est fourni. L'efficacité de la structure décroît donc puisque chaque bloc entraîne un surcoût.

Comme illustré plus loin, nous avons mesuré au cours de nos évaluations que le contenu représente à terme moins de 1% de taille de la structure de données. Les 99% restants correspondent aux métadonnées utilisées par la séquence répliquée. Il est donc nécessaire de proposer des mécanismes et techniques afin de mitiger les problèmes soulignés précédemments.

1.5 Mitigation du surcoût des séquences répliquées sans conflits

— Plusieurs approches ont été proposées pour réduire croissance des métadonnées dans Sequence CRDTs

— RGA (et RGASplit) propose un mécanisme de GC des pierres tombales. Nécessite cependant stabilité causale des opérations de suppression. S'agit d'une contrainte

forte, peu adaptée aux systèmes dynamiques à large échelle. *Matthieu: TODO : Trouver référence sur la stabilité causale dans systèmes dynamiques*

— Core & Nebula propose un mécanisme de ré-équilibrage de l'arbre pour Treedoc. Le ré-équilibrage a pour effet de supprimer des potentielles pierres tombales et de réduire la taille des identifiants. Repose sur un algorithme de consensus. S'agit de nouveau d'une contrainte forte pour systèmes dynamique à large échelle. Pour y pallier, propose de séparer les pairs entre deux ensembles : Core et Nebula. Permet de limiter le nombre participant au consensus. Un protocole de rattrapage permet aux noeuds de la Nebula de mettre à jour leurs modifications concurrentes à un ré-équilibrage.

— LSEQ adopte une autre approche. Part du constat que les identifiants dans Logoot croissent de manière linéaire. Vise une croissance logarithmique des identifiants. Pour cela, propose de nouvelles fonctions d'allocation des identifiants visant à maximiser le nombre d'identifiants insérés avant de devoir augmenter la taille de l'identifiant. Propose aussi d'utiliser une base exponentielle pour la valeur *position* des identifiants. Atteint ainsi la croissance polylogarithmique des identifiants, sans coordination requise entre les noeuds et mécanisme supplémentaire. Solution adaptée aux systèmes distribués à large échelle. Conjecture cependant que cette approche se marie mal avec les Sequence CRDTs utilisant des blocs. En effet, ajoute une raison supplémentaire à la croissance des identifiants : l'insertion entre identifiants contigus. Force alors la croissance des identifiants.

1.6 Synthèse

Chapitre 2

Renommage dans une séquence répliquée

Sommaire

2.1 Présentation de l'approche 18
2.1.1 Modèle du système 18
2.1.2 Définition de l'opération de renommage 18
2.2 RenamableLogootSplit 20
2.2.1 Opération de renommage proposée 20
2.2.2 Gestion des opérations concurrentes au renommage 21
2.2.3 Évolution du modèle de livraison des opérations 24
2.3 RenamableLogootSplit v2 26
2.3.1 Conflits en cas de renommages concurrents 26
2.3.2 Relation de priorité entre renommages 27
2.3.3 Algorithme d'annulation de l'opération de renommage 28
2.3.4 Processus d'intégration d'une opération 31
2.3.5 Règles de récupération de la mémoire des états précédents . . . 35
2.4 Validation 38
2.4.1 Preuve de correction de RENAMEID 38
2.4.2 Complexité en temps des opérations 38
2.4.3 Expérimentations 42
2.4.4 Résultats 43
2.5 Discussion 49
2.5.1 Stockage des états précédents sur disque 49
2.5.2 Compression et limitation de la taille de l'opération *rename* . . 49
2.5.3 Définition de relations de priorité pour minimiser les traitements 50
2.5.4 Report de la transition vers la nouvelle epoch principale 50
2.5.5 Utilisation de l'opération de renommage comme mécanisme de compression du log d'opérations 51
2.5.6 Implémentation alternative de l'intégration de l'opération *rename* basée sur le log d'opérations 53

2.6 Comparaison avec les approches existantes 55
2.6.1 Core-Nebula 55
2.6.2 LSEQ 55
2.6.3 Eager stability determination 55
2.7 Conclusion 55

2.1 Présentation de l'approche

Nous proposons un nouveau CRDT pour la *Sequence* appartenant à l'approche des identifiants densément ordonnées : RenamableLogootSplit [37, 38]. 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.

2.1.1 Modèle du système

Le système est composé d'un ensemble dynamique de noeuds, les noeuds pouvant rejoindre puis quitter la collaboration tout au long de sa durée. Les noeuds collaborent afin de construire et maintenir une séquence à l'aide de RenamableLogootSplit. Chaque noeud possède une copie de la séquence et peut l'éditer sans se coordonner avec les autres. Les modifications des noeuds prennent la forme d'opérations qui sont appliquées immédiatement à leur copie locale. Les opérations sont ensuite transmises de manière asynchrone aux autres noeuds pour qu'ils puissent à leur tour appliquer les modifications à leur copie.

Les noeuds communiquement par l'intermédiaire d'un réseau Pair-à-Pair (P2P). Ce réseau est non-fiable : les messages peuvent être perdus, ré-ordonnés ou même livrés à plusieurs reprises. Le réseau peut aussi être sujet à des partitions, qui séparent alors les noeuds en des sous-groupes disjoints. Afin de compenser les limitations du réseau, les noeuds reposent sur une couche de livraison de messages.

Puisque RenamableLogootSplit est une extension de LogootSplit, il partage les mêmes contraintes sur la livraison de messages. La couche de livraison de messages sert donc à livrer les messages à l'application exactement une fois. La couche de livraison de messages a aussi pour tâche de garantir la livraison des opérations de suppression après les opérations d'insertion correspondantes. Aucune autre contrainte n'existe sur l'ordre de livraison des opérations. Finalement, la couche de livraison intègre aussi un mécanisme d'anti-entropie [41]. Ce mécanisme permet aux noeuds de se synchroniser par paires, en détectant et ré-échangeant les messages perdus.

2.1.2 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 SEC, nous définissons plusieurs propriétés de sécurité que l'opération *rename* doit respecter. Ces propriétés sont inspirées principalement par celles proposées dans [60].

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.*

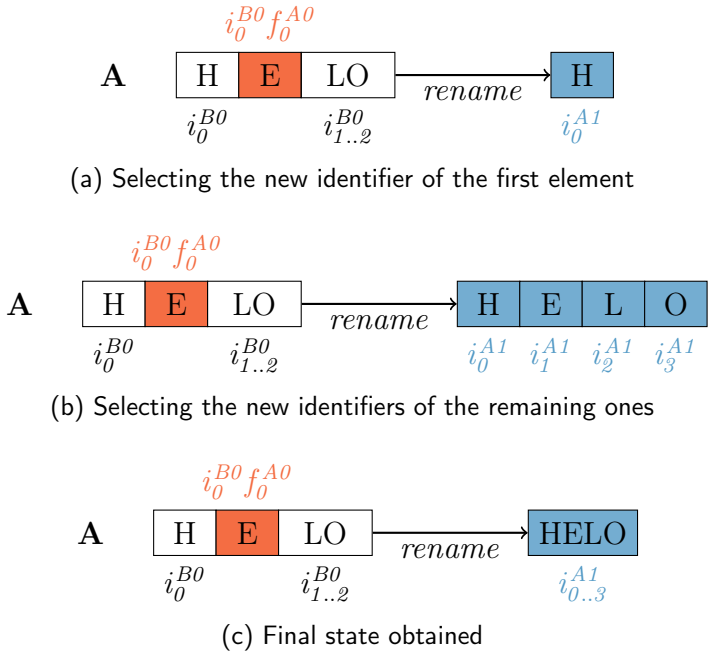
Propriété 3 *(Séquence bien formée) La séquence répliquée doit être bien formée. Appliquée une opération* rename *sur une séquence bien formée doit produire une nouvelle séquence bien formée. Une séquence bien formée doit respecter les propriétés suivantes :*

> ***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 délivrées dans des ordres différents à chaque noeud. Afin de garantir la convergence des réplicas, 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 4 est particulièrement difficile à assurer. Cette difficulté est dûe 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 intrinséquement 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 4.

Dans un souci de simplicité, la présentation de l'opération *rename* est divisée en deux parties. Dans le section 2.2, nous présentons l'opération *rename* proposée avec l'hypothèse



(a) Selecting the new identifier of the first element

(b) Selecting the new identifiers of the remaining ones

(c) Final state obtained

FIGURE 2.1 – Renaming the sequence on node A

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 le section 2.3, nous supprimons cette hypothèse. Nous présentons alors notre approche pour gérer les scénarios avec des opérations *rename* concurrentes.

2.2 RenamableLogootSplit

2.2.1 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 2.1. 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 2.1a. 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 2.1b. Comme nous assignons des identifiants consécutifs à tous les éléments de la séquence, nous pouvons au final aggréger ces éléments en un seul bloc, comme illustré en 2.1c. Ceci permet aux noeuds de bénéficier au mieux

de la fonctionnalité des blocs et de minimiser le surcoût en métadonnés 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 10 (Ancien état) *Un* ancien état *est la liste des idInterval 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 11 (rename) *Une opération* rename *est un triplet* ⟨*nodeId, nodeSeq, formerState*⟩ *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 sous-section 2.2.2 comment les noeuds peuvent les renommer de manière déterministe.

2.2.2 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 2.2 illustre de tels cas.

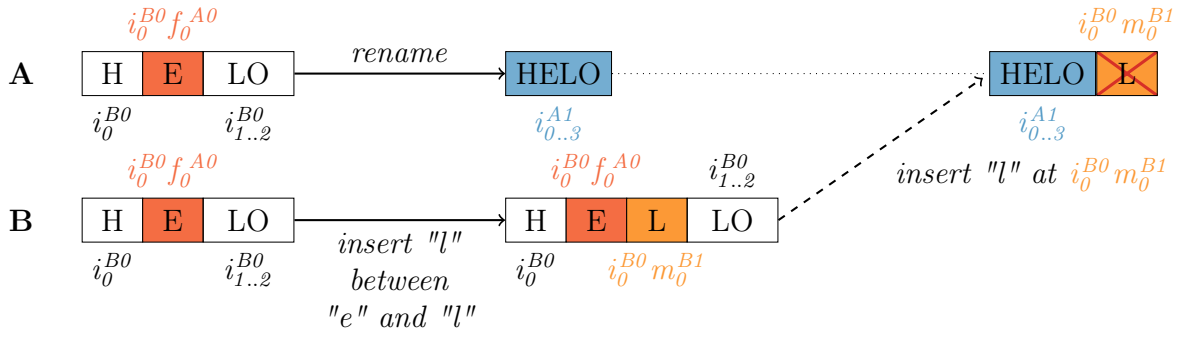


FIGURE 2.2 – Concurrent update leading to inconsistency

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 2.2. 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 d'à sa position prévue. Comme décrit par cet exemple, appliquer naivement 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 2.2 permet aux noeuds de faire progresser leur état de ε_0 à ε_{A1}. Nous définissons les époques de la manière suivante :

Définition 12 (Époque) *Une époque est un triplet ⟨nodeId, nodeSeq, formerState⟩ 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.*
- *formerState est l'ancien état du noeud de la génération de cette époque.*

Notons que l'époque générée est caractérisée et identifiée de manière unique par son couple ⟨nodeId, nodeSeq⟩.

Au fur et à mesure que les noeuds reçoivent des opérations *rename*, ils construisent et maintiennent localement la *chaîne des époques*, une structure de données ordonnant les époques en fonction de leur relation *parent-enfant*. De plus, les noeuds marquent chaque opération avec l'identifiant de 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 Figure 2.3, 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 prédecesseur. Un exemple est présenté dans la Figure 2.4. Cette figure décrit le même scénario que la Figure 2.2, à l'exception que le noeud A utilise RENAMEID pour renommer les identifiants générés de façon concurrente avant de les insérer dans son état.

L'algorithme procède de la manière suivante. Tout d'abord, le noeud récupère le prédecesseur 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 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.

```
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 ∈ renameIds 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 ← findIndexOfPred(id, renamedIds)
    newPredId ← new Id(pos, nId, nSeq, index)

    return concat(newPredId, id)
end function
```

FIGURE 2.3 – Main functions to rename an identifier

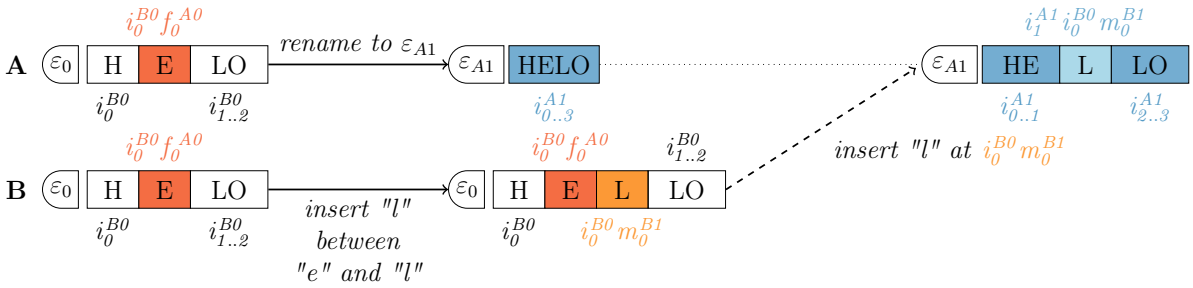


FIGURE 2.4 – Renaming concurrent update using RENAMEID before applying it to maintain intended order

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 modifications concurrentes. Ce cas correspond à celui du noeud B dans la Figure 2.4. 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.

Figure 2.3 présente seulement le cas principal de RENAMEID, c.-à-d. le cas où l'identifiant à renommer appartient à l'interval 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 interval ($id <_{id} firstId$ ou $lastId <_{id} id$), sont présentées dans A.

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.

2.2.3 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 :

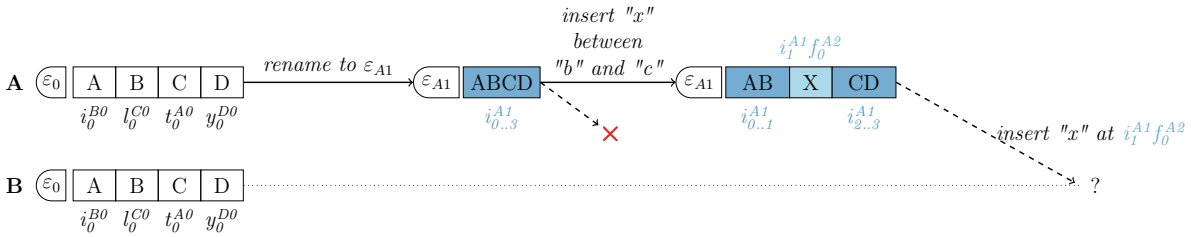


FIGURE 2.5 – TODO

Dans la Figure 2.5, 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 2.2, 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 avions défini dans la 9. 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 supprimés.

Pour prévenir les scénarios tels que celui illustré par la Figure 2.5 nous y ajoutons la règle suivante : les opérations *rename* doivent être livrées à la structure de données avant les opérations dépendant causalement dessus. Nous obtenons donc le modèle de livraison suivant :

Définition 13 (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 :*

1. *Une opération doit être délivrée à l'ensemble des noeuds à terme,*
2. *Une opération doit être délivrée qu'une seule et unique fois aux noeuds,*
3. *Une opération* remove *doit être délivrée à un noeud une fois que les opérations* insert *des éléments concernés par la suppression ont été délivrées à ce dernier.*
4. *Une opération doit être délivrée à un noeud une fois que l'opération* rename *une fois que l'opération* rename *qui introduit son époque de génération a été delivrée 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 2.6 présente un exemple de ce cas figure.

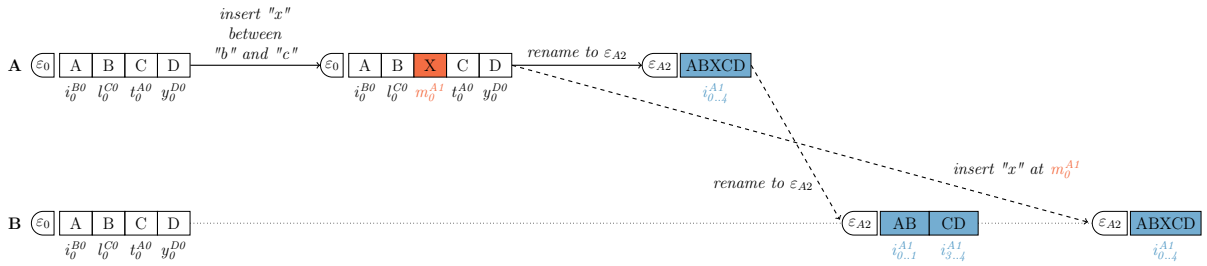


FIGURE 2.6 – TODO

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.

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 renomme 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.

2.3 RenamableLogootSplit v2

2.3.1 Conflits en cas de renommages concurrents

Nous considérons à présent les scénarios avec des opérations *rename* concurrentes. Figure 2.7 développe le scénario décrit précédemment dans Figure 2.4.

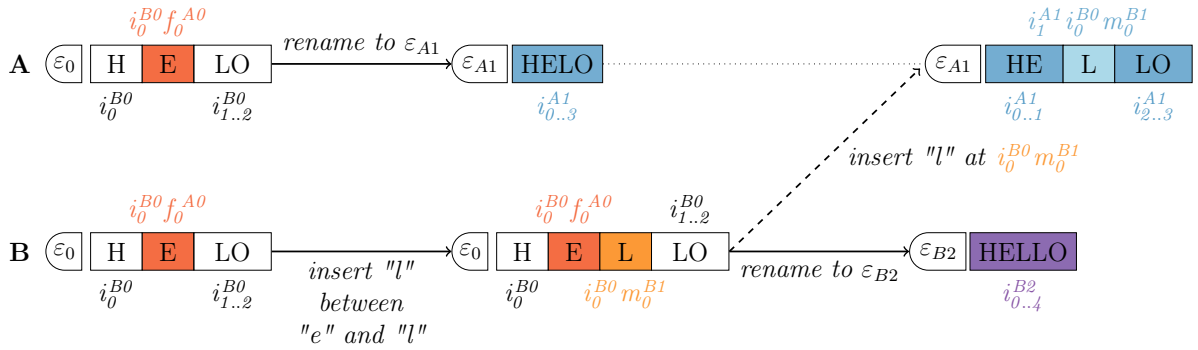


FIGURE 2.7 – Concurrent *rename* operations leading to divergent states

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 délivrée 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 2.8 l'*arbre des époques* que les noeuds obtiennent une fois qu'ils se sont synchronisés à terme. Les époques sont representées sous la forme de noeuds de l'arbre et la relation *parent-enfant* entre elles est illustrée sous la forme de flèches noires.

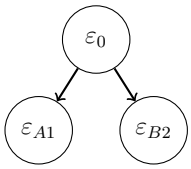


FIGURE 2.8 – The *epoch tree* corresponding to the scenario of Figure 2.7

À l'issue du scénario décrit dans la Figure 2.7, 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 Figure 2.3 permet seulement au 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 dûs 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 sous-section 2.3.2.

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 qui remplit cet objectif dans la sous-section 2.3.3.

2.3.2 Relation de priorité entre renommages

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

Définition 14 (Relation *priority* $<_\varepsilon$) *La relation* priority $<_\varepsilon$ *est un ordre total strict 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 :

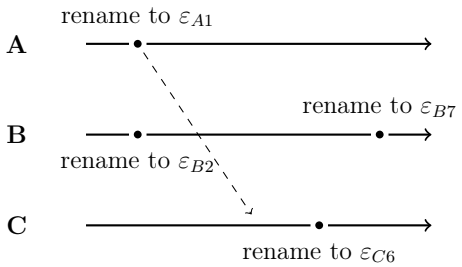
Définition 15 (Époque cible) *L'époque de l'ensemble des époques vers laquelle les noeuds doivent progresser. Les noeuds sélectionnent comme époque cible l'époque maximale d'après l'ordre établit par* priority*.*

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 2.9 fournit un exemple.

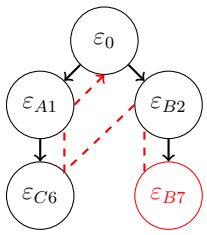
La 2.9a décrit une exécution dans laquelle trois noeuds A, B et C génèrent plusieurs opérations avant de se synchroniser à terme. Comme seules les opérations *rename* sont pertinentes pour le problème qui nous occupe, seules ces opérations sont représentées dans cette figure. Initialement, le noeud A génère une opération *rename* qui introduit l'époque ε_{A1}. Cette opération est délivrée 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 2.9b. Dans cette figure, la flèche pointillé 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



(a) Example of execution with concurrent *rename* operations



(b) Corresponding epoch tree with *priority* relation shown

FIGURE 2.9 – Selecting target epoch from execution with concurrent *rename* operations

$\varepsilon_0 < \varepsilon_0\varepsilon_{A1} < \varepsilon_0\varepsilon_{A1}\varepsilon_{C6} < \varepsilon_0\varepsilon_{B2} < \varepsilon_0\varepsilon_{B2}\varepsilon_{B7}$. 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.

2.3.3 Algorithme d'annulation de l'opération de renommage

Nous présentons maintenant la fonction REVERTRENAMEID. Décrite dans Figure 2.10, cette fonction permet aux noeuds d'annuler une opération *rename* appliquée précédemment. Pour ce faire, REVERTRENAMEID associe les identifiants de l'époque *enfant* aux identifiants correspondant dans l'époque *parente*.

Les objectifs de REVERTRENAMEID sont les suivants : (i) Restaurer à leur ancienne valeur les identifiants générés causalement avant ou de manière concurrente à l'opération *rename* annulée (ii) Assigner de nouveaux identifiants respectant l'ordre souhaité aux éléments qui ont été insérés causalement après l'opération *rename* annulée. Nous illustrons son comportement à l'aide de la Figure 2.11.

Cette figure reprend le scénario de la Figure 2.2. Le noeud A reçoit l'opération *rename* du noeud B, qui est concurrente à l'opération *rename* que le noeud A a appliqué précédemment. Selon la relation *priority* proposée, le noeud A sélectionne l'époque introduite ε_{B2} comme l'époque cible ($\varepsilon_{A1} <_\varepsilon \varepsilon_{B2}$). Il procède donc à ramener son état à un état équivalent à l'époque ε_0, le PPAC de son époque courante ε_{A1} et de l'époque cible ε_{B2}. Pour ce faire, il applique REVERTRENAMEID à chaque identifiant de son état courant.

REVERTRENAMEID détermine quelle stratégie appliquer pour restaurer 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
```

FIGURE 2.10 – Main functions to revert an identifier renaming

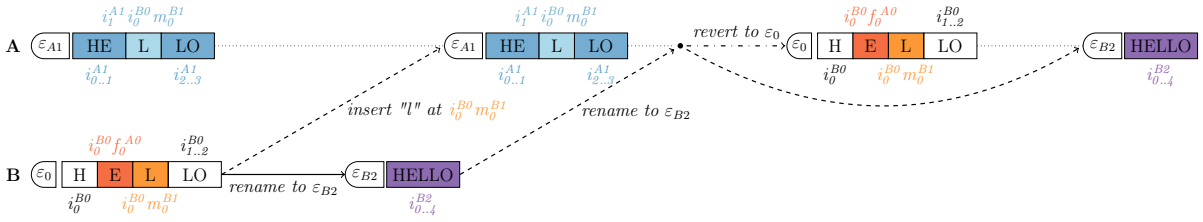


FIGURE 2.11 – Reverting a previously applied *rename* operation

donné en utilisant des motifs. Par exemple, les identifiants de la forme $pos_{offset}^{nId\ nSeq}$ (i_{offset}^{A1} dans l'exemple courant) correspondent aux nouvelles valeurs des identifiants qui composent l'*ancien état*. Pour retrouver les identifiants d'origine, REVERTRENAMEID utilise simplement leur offset puisqu'il correspond à leur index dans l'*ancien état*. Par exemple, l'identifiant ayant pour offset 0 correspond au premier identifiant dans l'*ancien état* (i_0^{B0}), celui ayant pour offset 1 au second identifiant ($i_0^{B0} f_0^{A0}$), et ainsi de suite.

Les identifiants de la forme $pos_{offset}^{nId\ nSeq} tail$ (e.g. $i_1^{A1} i_0^{B0} m_0^{B1}$) correspondent à des identifiants qui ont été soit insérés de façon concurrente à l'opération *rename*, soit causalement après. Pour traiter ces identifiant, REVERTRENAMEID retire tout d'abord le premier tuple (i_1^{A1}) pour isoler la queue de l'identifiant ($i_0^{B0} m_0^{B1}$). En faisant cela, REVERTRENAMEID annule la transformation appliquée à l'identifiant par RENIDFROMPREDID si l'identifiant a été inséré de manière concurrente. L'algorithme compare ensuite la queue de l'identifiant aux identifiants de l'élément précédant et de l'élément suivant dans l'*ancien état*. Dans cet exemple, nous avons $i_0^{B0} f_0^{A0} <_{id} i_0^{B0} m_0^{B1} <_{id} i_1^{B0}$. L'algorithme peut alors retourner la queue comme identifiant résultant tout en préservant l'ordre souhaité, puisque sa valeur est comprise entre celles des identifiants du prédécesseur et du successeur.

Sinon, cela signifie que l'identifiant donné a été inséré de manière causale après l'opération *rename*. Puisqu'aucun identifiant correspondant n'existe encore à l'époque *parente*, REVERTRENAMEID peut retourner n'importe quel identifiant tant qu'il préserve l'ordre souhaité. Pour ce faire, REVERTRENAMEID génère l'identifiant à partir de l'identifiant du prédecesseur ou du successeur, et en utilisant des tuples exclusifs au mécanisme de renommage : MIN_TUPLE et MAX_TUPLE.

Matthieu: TODO : Modifier exemple pour illustrer le cas de figure où on a besoin de MIN/MAX_ TUPLE

Une fois que le noeud A a converti son état à un état équivalent à l'époque ε_0 en utilisant REVERTRENAMEID, il peut appliquer RENAMEID pour calculer l'état correspondant à ε_{B2}.

Comme pour Figure 2.3, Figure 2.10 ne présente seulement que le cas principal de REVERTRENAMEID. Il s'agit du cas où l'identifiant à restaurer appartient à l'interval des identifiants renommés $newFirstId \leq_{id} id \leq_{id} newLastId$). Les fonctions pour gérer les cas restants sont présentées dans B.

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.

2.3.4 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*.

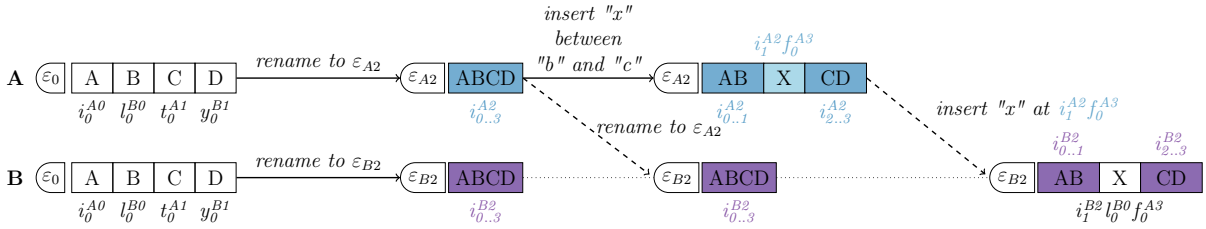
Intégration d'une opération *insert* ou *remove* distante

Dans Figure 2.12, nous présentons l'algorithme d'intégration d'une opération *insert* distante dans RenamableLogootSplit.

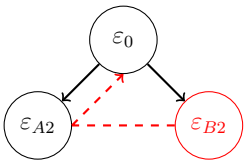
```
    function insRemote(seq, epochTree, currentEpoch, insOp)
        if currentEpoch = opEpoch then
            insert(seq, getIdBegin(insertOp), getContent(insertOp))
        else
 5:         insertedIdInterval ← getInsertedIdInterval(insOp)
            ids ← expand(insertedIdInterval)

            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:             revertRenameId_partial ← papply(revertRenameId, renamedIds, nId, nSeq)
                ids ← map(ids, revertRenameId_partial)
            end for

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

Figure 2.12 – TODO

Cet algorithme se décompose en de multiples étapes. Afin d'illustrer chacune d'entre elles, nous utilisons l'exemple représenté par la Figure 2.13.



(a) Execution



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

FIGURE 2.13 – TODO

Matthieu: TODO : Remplacer Figure 2.13 par un exemple avec plus d'opérations rename *pour mieux faire apparaître les calculs et manipulations effectués sur les chemins dans l'*arbre des époques

Dans la 2.13a, 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 2.13b.

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 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*

([ε_{B2}, ε_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 ([ε_{A2}] et [ε_{B2}]) (v) elle inverse l'ordre des époques du chemin entre l'époque courante et la racine ([ε_{B2}]) (vi) elle retourne les deux chemins obtenus ($\langle$[ε_{A2}], [ε_{B2}]$\rangle$).

Le chemin entre l'époque de l'opération et l'époque PPAC ([ε_{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 ([ε_{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 aggrège les identifiants en intervals d'identifiants au préalable à l'aide de la fonction AGGREGATE (ligne 28). Cette fonction regroupe simplement les identifiants contigus en intervals d'identifiants et retourne la liste des intervals obtenus.

À partir des intervals d'identifiants obtenus et du contenu initial de l'opération *insert*, le noeud regénère 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 la Figure 2.14.

Comme précédemment, nous utilisons l'exemple illustré dans la Figure 2.15 pour présenter le fonctionnement de cet algorithme.

La Figure 2.15 reprend le scénario décrit précédemment dans la Figure 2.13. 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 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 5).

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 (ligne 8).

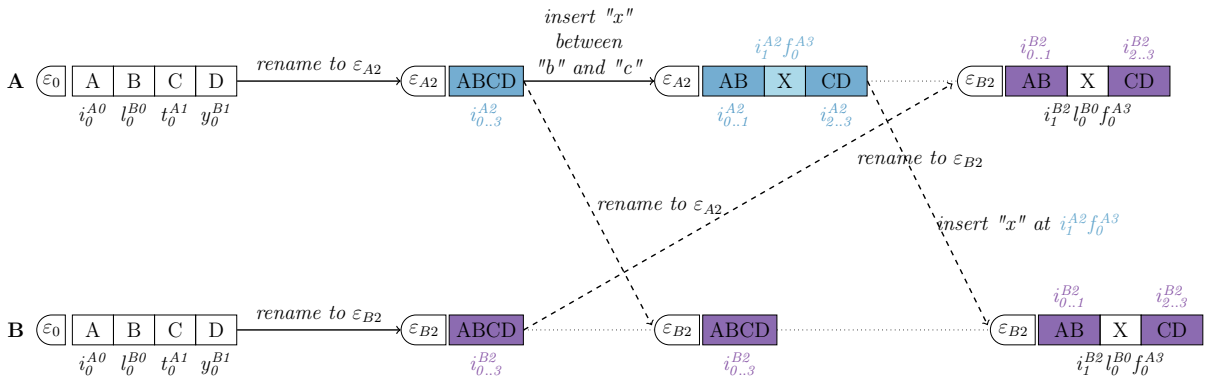
```
    function renRemote(seq, epochTree, currentEpoch, renOp)
        opEpoch ← getEpoch(renOp)
        introducedEpoch ← getIntroducedpoch(renOp)

 5:     addEpoch(epochTree, introducedEpoch, opEpoch)

        if introducedEpoch <_ε currentEpoch then
            return ⟨seq, currentEpoch⟩
        else
10:         idIntervals ← getIdIntervals(seq)
            ids ← flatMap(idIntervals, expand)

            ⟨epochsToRevert, epochsToApply⟩ ← getPathBetweenEpochs(epochTree, currentEpoch, introducedEpoch)

15:         for epoch in epochsToRevert do
                renamedIds ← getRenamedIds(epochTree, epoch)
                nId ← getNodeId(epochTree, epoch)
                nSeq ← getNodeSeq(epochTree, epoch)
                revertRenameId_partial ← papply(revertRenameId, renamedIds, nId, nSeq)
20:             ids ← map(ids, revertRenameId_partial)
            end for

            for epoch in epochsToApply do
                renamedIds ← getRenamedIds(epochTree, epoch)
25:             nId ← getNodeId(epochTree, epoch)
                nSeq ← getNodeSeq(epochTree, epoch)
                renameId_partial ← papply(renameId, renamedIds, nId, nSeq)
                ids ← map(ids, renameId_partial)
            end for
30:
            nId ← getNodeId(seq)
            nSeq ← getNodeSeq(seq)
            newIdIntervals ← aggregate(ids)
            content ← getContent(seq)
35:         blocks ← generateBlocks(newIdIntervals, content)
            newSeq ← new LogootSplit(nId, nSeq, blocks)

            return ⟨newSeq, introducedEpoch⟩
        end if
40: end function
```

FIGURE 2.14 – TODO

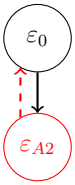
Dans le cas présent, on a $\varepsilon_{A2} <_\varepsilon \varepsilon_{B2}$. ε_{B2} devient donc la nouvelle époque courante. Le noeud A procède au renommage de son état vers cette nouvelle époque.

Pour cela, le noeud récupère l'ensemble des identifiants formant son état courant (lignes 10-11). 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 15-29).

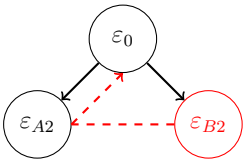
Le noeud obtient alors la liste des identifiants courant, à la nouvelle époque cible. Il ne lui reste plus qu'à construire une nouvelle séquence à partir de ces identifiants. Pour cela,



(a) Execution



(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 2.15 – TODO

le noeud regénère des blocs à partir des intervals 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 36). Finalement, RENREMOTE renvoie cette nouvelle séquence ainsi que la nouvelle époque courante.

2.3.5 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* [9]. Une opération est causalement stable une fois qu'elle a été délivrée 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* :

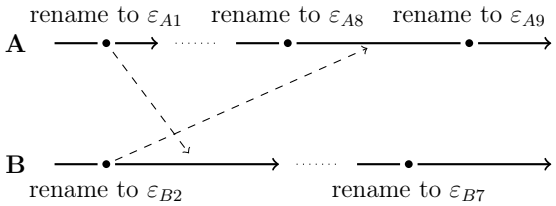
Définition 16 (Potentielles époques courantes) *L'ensemble des époques auxquelles les noeuds peuvent se trouver actuellement et à partir desquelles ils peuvent émettre des opérations, du point de vue du noeud courant. Il s'agit d'un sous-ensemble de l'ensemble*

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.

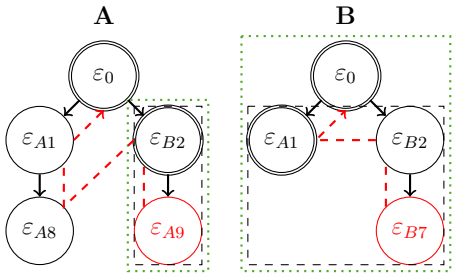
Pour traiter les prochaines opérations, les noeuds doivent maintenir les chemins entre toutes les époques de l'ensemble des *potentielles époques courantes*. Nous appelons *époques requises* l'ensemble des époques correspondant.

Définition 17 (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 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 2.16 illustre un cas d'utilisation du mécanisme de récupération de mémoire proposé.



(a) Execution



(b) States of respective epoch trees with *potential current epochs* and *required epochs* displayed

FIGURE 2.16 – Garbage collecting epochs and corresponding *former states*

Dans la 2.16a, nous représentons une exécution au cours de laquelle deux noeuds A et B génère respectivement plusieurs opérations *rename*. Dans la 2.16b, nous représentons les *arbre des époques* respectifs de chaque noeud. Les époques introduites par des opérations *rename* causalement stables sont representées en utilisant des doubles cercles. L'ensemble des *potentielles époques courantes* est montré sous la forme d'un rectangle noir pointillé, tandis que l'ensemble des *époques requises* est représenté par un rectangle vert pointillé.

Matthieu: TODO : Trouver un autre terme que pointillé pour dotted

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}.

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 [19, 18] est donc requis.

La stabilité causale peut prendre un certain temps à atteinte. En attendant, les noeuds peuvent en fait décharger les anciens états sur le disque dur puis qu'ils sont seulement nécessaires pour traiter les opérations concurrentes aux opérations *rename*. Nous approfondissons ce sujet dans la sous-section 2.5.1.

2.4 Validation

2.4.1 Preuve de correction de RENAMEID

2.4.2 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, structure de données utilisée...

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 qu'un arbre AVL est utilisé comme structure de données pour représenter l'état interne de la séquence et que des tableaux sont utilisés pour représenter les *anciens états*.

*Matthieu: TODO : Trouver structure de données adaptée à l'*arbre des époques*. Besoin de pouvoir établir rapidement le chemin entre la racine et une époque. Pour cela, le mieux serait d'avoir accès directement à l'époque et qu'elle référencie l'époque parente.*

Matthieu: NOTE : Une table de hachage correspond bien (ce que j'utilise dans l'implem). Mais pas le plus adapté pour la garbage collection des époques obsolètes (besoin de parcourir l'ensemble des clés et de supprimer celles n'appartenant plus aux époques requises*).*

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

À partir de ces hypothèses, nous établissons les complexités 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 voulu puis à insérer le bloc dans l'arbre AVL. D'après ANDRÉ et al. [5], 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 section 2.3.4, 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 prédecesseur 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 genère pour un identifiant donné un nouvel identifiant équivalent à l'époque suivante en se basant sur son prédecesseur. Cependant, il est nécessaire ici de faire une recherche pour déterminer le prédecesseur 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 prédecesseur 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*

- Concernant l'opération *rename*
- L'opération *rename* locale consiste à parcourir et à linéariser la structure actuelle pour ne garder que les idIntervals courant
- Et à créer une nouvelle séquence équivalente à l'ancienne, composée seulement du nouveau bloc

— Sa complexité est donc de $O(b)$
— Pour l'opération *rename* distante, une implémentation naïve consiste à générer le nouvel état en renommant et insérant chaque identifiant de l'état courant d'une manière similaire à l'opération *insert* distante
— On obtient dans ce cas une complexité de $O(h + n\ (k + l \cdot \log(n) + \log(b)))$
— Mais peut améliorer
— Plutôt que d'effectuer une recherche binaire sur $renamedIds$ pour trouver le prédecesseur de id
— Peut tirer parti du fait qu'on va parcourir séquentiellement l'état, qui est une liste triée d'identifiants
— Et parcourir en parallèle, au fur et à mesure, $renamedIds$ pour trouver le prédecesseur du id courant
— Permet de ramener la complexité du renommage des n identifiants de l'époque PPAC à l'époque courante à $O(l\ (n + n))$
— Plutôt que de générer une nouvelle séquence et d'y insérer un par un les n éléments
— Peut parcourir les n éléments pour reformer b blocs
— Peut ensuite construire l'AVL de manière récursive en parcourant les b blocs obtenus
— Peut donc obtenir une complexité de $O(h + k \cdot n + l\ (n + n) + n + b)$ pour l'intégration de l'opération *rename* distante

Récapitulatif

TABLE 2.1 – Complexité en temps des différentes opérations

Type d'opération	Complexité en temps	
	Locale	Distante
insert	$\log b$	$h + s(k + l \cdot \log n + \log b)$
remove	$\log b$	$h + s(k + l \cdot \log n + \log b)$
naive rename	n	$h + n(k + l \cdot \log n + \log b)$
rename	n	$b + n(k + 2 \cdot l + 1) + b$

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

Matthieu: NOTE : Dans sous-section 2.3.5, je présente le principe du mécanisme de GC. Dans cette partie, je décris l'algo correspondant et l'évalue. Rédiger l'algo ? Dans quelle partie l'insérer dans ce cas ?

Pour compléter notre analyse théorique des performances de RenamableLogootSplit, nous proposons une analyse en complexité en temps du mécanisme présenté en sous-section 2.3.5 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 définit par la relation *priority*. Ce parcours se poursuit 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 le PPAC. Pour ce faire, l'algorithme calcule de manière successive la dernière intersection entre le chemin de la racine jusqu'au PPAC courant et les chemins précédemment calculés. Le 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 le 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 2.2.

TABLE 2.2 – Complexité en temps du mécanisme de récupération de mémoire des époques

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

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.

2.4.3 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 reproduisent le scénario suivant.

Scénario d'expérimentation

Plusieurs auteurs rédigent de manière collaborative un article en temps réel. Dans un premier temps, les auteurs spécifie 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 arbitrairement définie comme critique, les collaborateurs passent à la seconde phase de la collaboration. Au cours de cette phase, les auteurs arrêtent d'ajouter du nouveau contenu mais se concentre à la place sur le remaniement du contenu existant. Ceci est simulé en équilibrant le ratio entre les opérations *insert* et *remove*. Chaque auteur doit émettre un nombre donné d'opérations *insert* et *remove*. La simulation prend fin une fois que tous les collaborateurs ont reçu toutes les opérations. Au cours de la simulation, nous prenons des instantanés de l'état des pairs à des points donnés pour suivre leur évolution.

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 auteur. Les bots partagent et éditent de façon collaborative le document en utilisant soit LogootSplit soit RenamableLogootSplit en fonction de la session. Dans chaque cas, chaque bot génère localement une opération *insert* ou *remove* toutes les 200 ± 50ms et la diffuse immédiatement aux autres noeuds via un réseau P2P maillé. Au cours de la première phase, la probabilité d'émetter une opération *insert* (resp. *remove*) est de 80% (resp. 20%). Une fois que le document atteint 60k caractères (environ 15 pages), les bots passent à la seconde phase et mettent chaque probabilité à 50%. Après chaque opération locale, le bot peut déplacer son curseur à une autre position aléatoire dans le document avec une probabilité de 5%. Chaque bot génère 15k opérations *insert* ou *remove* et s'arrête une fois qu'il a

observé 150k opérations. Des instantanés de l'état du bot sont pris de façon périodique, toutes les 10k opérations observées.

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 30k opérations qu'ils observent. Ces opérations *rename* sont générées de façon 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/`.

2.4.4 Résultats

En utilisant les instantanés générés, nous avons effectué plusieurs benchmarks. Ces benchmarks évaluent les performances de RenamableLogootSplit et les compare à celles de LogootSplit. 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 RenamableLogootSplit. 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 2.17.

Pour chaque graphique dans la Figure 2.17, nous représentons 4 données différentes. La ligne pointillée 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 en pointillés représente la taille du document RenamableLogootSplit dans son meilleur cas. Dans ce scénario, les noeuds considèrent que les opérations *rename* sont supprimables 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 et qu'elles ne peuvent donc jamais être supprimées. Les noeuds doivent alors conserver de façon permanente les métadonnées introduites par le

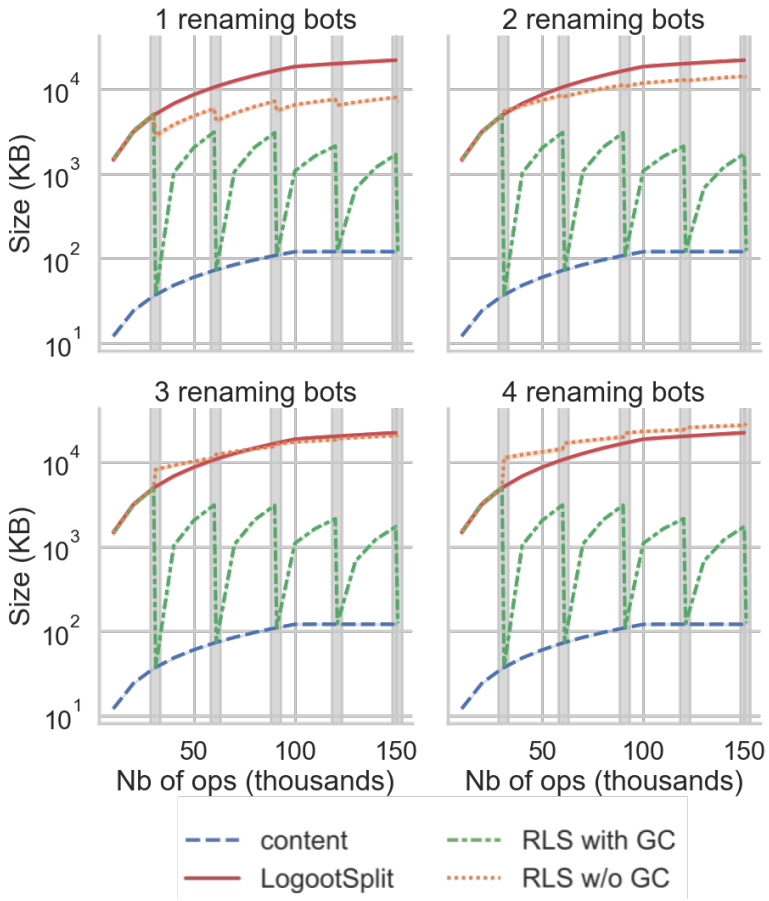


FIGURE 2.17 – Evolution of the size of the document

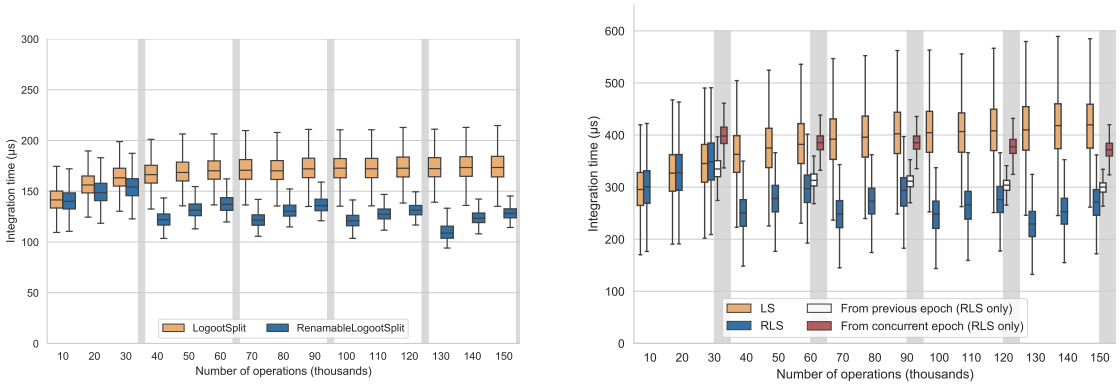
mécanisme de renommage. Les performances de RenamableLogootSplit diminuent donc à mesure que le nombre de *renaming bots* et d'opérations *rename* générées augmente. Néanmoins, nous observons que RenamableLogootSplit peut surpasser les performances de LogootSplit tant que le nombre de *renaming bots* 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.

Pour récapituler les résultats présentés, le mécanisme de renommage introduit un surcoût temporaire en métadonnées qui augmente avec chaque opération *rename*. Mais le surcoût se résorbe à terme une fois que le système devient quiescent et que les opérations *rename* deviennent causalement stables. Dans la section sous-section 2.5.1, nous détaillerons l'idée que les *anciens états* peuvent être déchargés sur le disque en attendant que la stabilité causale soit atteinte pour atténuer le 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 [7] 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 2.18 présente les résultats obtenus.



(a) Local updates (b) Remote updates

FIGURE 2.18 – Integration time of standard operations

Dans ces figures, les boxplots orange correspondent aux temps d'intégration sur des documents LogootSplit, les boxplots bleu sur des documents RenamableLogootSplit. Bien que les deux mesures soient initialement équivalentes, 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 s'explique par le fait que l'opération *rename* optimise la représentation interne de la séquence.

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 blanche et rouge dans la 2.18b.

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 un surcoût additionnel puisque les noeuds doivent tout d'abord annulé 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 obtient de meilleures performances que LogootSplit tant que la distance entre l'époque de génération de l'opération et l'époque courante du noeud reste limité. Au fur et à mesure que la distance entre les deux époques augmente, les performances de RenamableLogootSplit diminuent, jusqu'à atteindre des performances moins bonnes que celles de LogootSplit, puisque le surcoût est multiplié. Néanmoins, le mécanisme de renommage réduit le temps d'intégration de la majorité des opérations, c.-à-d. les opérations générées entre deux séries d'opérations *rename*.

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 depuis l'opération *rename* précédente. Comme précédemment, nous distinguons les performances des modifications *locales* et *distantes*. Le cas des opérations *rename distantes* se sous-divise 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* désigne 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 2.3.

Table 2.3 – Integration time of rename operations

Parameters		Integration Time (ms)			
Type	Nb Ops (k)	Mean	Median	99ème Quant.	Std
Local	30	41.75	38.74	71.68	6.84
	60	78.32	78.16	81.42	1.24
	90	119.19	118.87	124.22	2.49
	120	143.75	143.57	148.59	2.16
	150	158.04	157.95	164.38	2.49
Direct remote	30	481.32	477.13	537.30	17.11
	60	981.62	978.24	1072.83	31.54
	90	1491.28	1481.83	1657.58	51.10
	120	1670.00	1663.85	1814.38	50.29
	150	1694.17	1675.95	1852.55	59.94
Cc. int. greater epoch	30	643.53	643.57	682.80	13.42
	60	1317.66	1316.39	1399.55	28.67
	90	1998.23	1994.08	2111.98	45.37
	120	2239.71	2233.22	2368.45	50.06
	150	2241.92	2233.61	2351.02	52.20
Cc. int. lesser epoch	30	1.36	1.30	3.53	0.37
	60	2.82	2.69	4.85	0.45
	90	4.45	4.23	5.81	0.71
	120	5.33	5.10	8.78	0.90
	150	5.53	5.26	8.70	0.79

Le principal résultat de ces mesures est que les opérations *rename* sont généralement

coûteuses quand comparées aux autres types d'opérations, puisque les noeuds doivent parcourir et renommer leur état courant complet. Les opérations *rename* locales s'intègrent en plusieurs centaines de millisecondes tandis que les opérations *distantes directes* et *concurrentes introduisant une plus grande époque* peuvent prendre des secondes si retardées trop longtemps. Il est donc nécessaire de prendre en compte ce résultat pour concevoir des stratégies de génération des opérations *rename* pour éviter d'impacter négativement l'expérience utilisateur.

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 sélectionné comme nouvelle époque cible, les noeuds ne procède 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.

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. Nous présentons les résultats obtenus dans Figure 2.19.

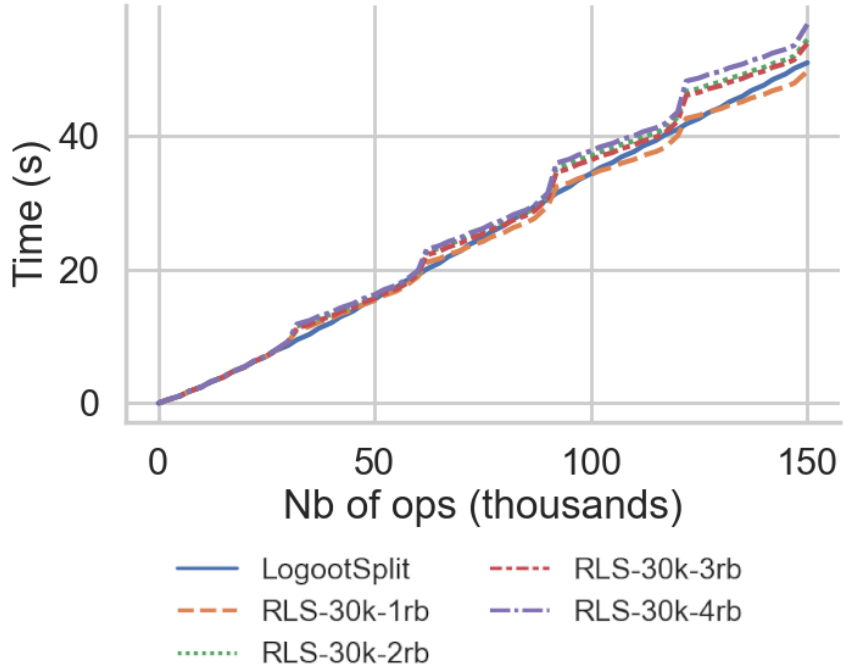


FIGURE 2.19 – Évolution du temps nécessaire pour rejouer le log d'opérations

Nous observons que le gain sur le temps d'intégration des opérations *insert* et *remove* permet initialement de contrebalancer le surcoût lié aux 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 davantage prononcé 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 correspondra au scénario principal et pourra être mitigé, par exemple en utilisant un mécanisme de compression du log d'opérations. Dans la sous-section 2.5.5, nous présentons comment mettre en place un tel mécanisme en se basant justement sur 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, 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 Tableau 2.4.

Paramètres		Temps d'intégration (μs)					Taille (B)				
Type	CRDT	Moyenne	Médiane	IQR	1er Quant.	99ème Quant.	Moyenne	Médiane	IQR	1er Quant.	99ème Quant.
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 (KB)				
Type	CRDT	Moyenne	Médiane	IQR	1er Quant.	99ème Quant.	Moyenne	Médiane	IQR	1er Quant.	99ème Quant.
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 2.4 – 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 définit 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 2.5.2. 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*.

2.5 Discussion

2.5.1 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 évènements 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*...

2.5.2 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 blocs. En effet, un identifiant peut être caractérisé de manière unique par le triplet composé de l'*identifiant de noeud*, du *numéro séquentiel* et de l'*offset* de son dernier tuple. Par conséquent, un bloc peut être identifié de manière unique à partir du triplet signature de son identifiant de début et de sa longueur. Cette méthode nous permet de réduire les données à diffuser dans le cadre de l'opération *rename*

à un montant fixe par bloc.

Pour décompresser l'opération reçue, les noeuds parcourent leur état courant ainsi que leur log des opérations *remove* concurrentes. De cette manière, ils peuvent retrouver les identifiants complets et reconstruire l'opération *rename* originale. *Matthieu: TODO : Développer ce paragraphe : noeuds doivent retrouver les blocs renommés à partir des données reçues. Pour cela, parcourent leur état. Suffit de retrouver un identifiant avec le même couple ⟨nodeId, nodeSeq⟩ pour reformer un bloc. Certains couples ⟨nodeId, nodeSeq⟩ peuvent avoir été supprimés en concurrence et ne plus être présent dans la séquence. Donc besoin d'aussi parcourir le log des opérations* remove *concurrentes.*

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 de blocs, bornant ainsi la taille du message à diffuser.

2.5.3 Définition de relations de priorité pour minimiser les traitements

Bien que la relation *priority* proposée dans la sous-section 2.3.2 est simple et garantit que tous les noeuds désignent la même époque comme époque cible, elle introduit un surcoût computationnel significatif dans certains cas. Notamment, cette relation *priority* autorise le cas où un simple noeud, déconnecté de la collaboration depuis longtemps, force l'ensemble des autres noeuds à annuler les opérations *rename* qu'ils ont effectué pendant ce temps car sa propre opération *rename* introduit la nouvelle époque cible. *Matthieu: TODO : ajouter figure d'un epoch tree où une longue branche se fait remplacer par une époque isolée*

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.

2.5.4 Report de la transition vers la nouvelle epoch principale

Comme illustré par Tableau 2.3, 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.

Matthieu: TODO : Ajouter figure où noeud reçoit successivement plusieurs opérations rename *concurrentes et procéde au renommage de son état 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 n'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 2.18b) comparé au coût de renommer, de manière inutile, complètement l'état (plusieurs centaines de milliseconds à des secondes complètes d'après Tableau 2.3).

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.

2.5.5 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 travailler. Le nouveau pair repose sur un mécanisme d'anti-entropie [41] pour récupérer l'ensemble des opérations via un autre pair. Puis il reconstruit l'état courant en appliquant successivement chacune des opérations. Ce processus 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 [48, 25] consistent à remplacer un sous-ensemble des opérations du log par une opération équivalente, par exemple en aggrégeant les opérations *insert* adjacentes. Une autre approche, présentée dans [8], définie 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 (Définition 11) 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 d'état*, structure représentant l'ensemble des opérations observées 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 18 (rename enrichie) *Une opération* rename enrichie *est un quintuplet* $\langle$*nodeId, nodeSeq, formerState, stateVector, 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.*
- *stateVector est le vecteur d'état 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.

Matthieu: TODO : Étudier si y a un intérêt à privilégier la synchronisation basée sur l'intégration successive de toutes les opérations quand on a cette méthode de synchronisation par snapshot/checkpoint de possible

2.5.6 Implémentation alternative de l'intégration de l'opération *rename* basée sur le log d'opérations

Nous avons décrit précédemment dans section 2.3.4, et plus précisément dans Figure 2.14, 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-créer 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 section 2.3.4. Tout d'abord, elle modifie le facteur du nombre de transformations

à effectuer. La version décrite dans section 2.3.4 transforme de l'époque courante vers l'époque cible chaque identifiant (ou chaque bloc si on dispose 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 sous-section 2.3.3 : 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. Notamment, chaque noeud traquent les informations de progression des autres noeuds afin de supprimer les métadonnées du mécanisme de renommage (cf. sous-section 2.3.5). 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

2.6 Comparaison avec les approches existantes

2.6.1 Core-Nebula

2.6.2 LSEQ

Matthieu: Serait intéressant d'avoir une implémentation combinant LogootSplit et LSEQ pour vérifier si les contraintes sur la création de blocs dans LogootSplit ne "sabotent" pas la croissance polylogarithmique des identifiants de LSEQ

2.6.3 Eager stability determination

Matthieu: Peut aussi aborder les travaux de Jim Bauwens et Elisa Gonzalez Boix [12, 10, 11] sur l'accélération de la stabilité causale : ne concerne pas seulement les séquences, mais les operation-based CRDTs. Permet de tronquer le log des opérations mais aussi d'accélerer le mécanisme de GC de RGA (et le mien aussi)

Matthieu: Peut aborder les travaux de Weidner, Miller et Meiklejohn [51, 50] qui combinent aussi CRDT et OT dans une certaine mesure. Pas vraiment dans le but de réduire les métadonnées du CRDT. Mais reste intéressant à présenter pour se différencier (eux proposent d'utiliser OT pour fusionner 2 CRDTs, moi pour ajouter une action qui est incompatible nativement avec les autres actions du CRDT)

2.7 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 ponctuellement 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 total stricte $<_\varepsilon$ 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 [44, 60], à l'exception de l'approche LSEQ [32]. En pratique, il serait intéressant d'étudier la mise en place d'un mécanisme d'éviction des noeuds inactifs pour répondre à ce problème.

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 à la Core et Nebula [60]. 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 3

MUTE, un éditeur web collaboratif P2P temps réel

Sommaire

3.1 Présentation 57
3.1.1 Objectifs 58
3.1.2 Architecture 58
3.2 Couche interface 58
3.3 Couche réplication 59
3.3.1 Modèle de données du document texte 59
3.3.2 Module de livraison des opérations 59
3.3.3 Métadonnées 62
3.3.4 Collaborateurs 62
3.3.5 Curseurs 63
3.4 Couche sécurité 63
3.4.1 Objectifs 63
3.4.2 Approche choisie 64
3.4.3 Limites 64
3.4.4 Perspectives 64
3.5 Couche réseau 65
3.5.1 Netflux 65
3.5.2 Pulsar 65
3.6 Pistes d'amélioration et de recherche 66
3.6.1 Fusion de versions distantes d'un document collaboratif 66
3.6.2 Rôles et places des bots dans systèmes collaboratifs 66
3.7 Conclusion 66

3.1 Présentation

— Plateforme d'expérimentation et de démonstration de l'équipe

3.1.1 Objectifs

- Éditeur collaboratif
- Permettre collaboration synchrone (temps réel) et asynchrone (mode offline)
- À grande échelle
- Respecter privacy, limiter au maximum la confiance qu'on demande aux utilisateurs d'avoir dans l'outil
- Facile d'accès
- Facilement déployable par des tiers
- S'inscrit dans la mouvance Local-First Software [28, 24]

3.1.2 Architecture

- Pour répondre à ces besoins, a effectué les choix suivants
- Application web
- Utilise CRDT pour représenter le document partagé
- Nous permet de supporter les différents modes de collaboration
- Nous permet aussi d'adopter une architecture P2P garantissant la privacy et le passage à l'échelle
- Mais présence de plusieurs serveurs, aux responsabilités limitées, pour simplifier la collaboration (signaling server, pulsar, bots(?))
- *Matthieu: TODO : Insérer schéma de l'architecture d'une collaboration (noeuds, types de noeuds et lien)*
- L'architecture d'un pair se décompose en plusieurs couches
- *Matthieu: TODO : Insérer schéma de l'architecture logicielle d'un pair*

3.2 Couche interface

- Éditeur Markdown
- Permet d'incorporer le style des éléments directement dans la séquence représentant le document texte
- Mécanisme de conscience de groupe
  - Liste des collaborateurs
  - Curseurs et sélections des autres collaborateurs
  - Indicateur de connexion
- Stocke au sein du navigateur les données du document (état du document, log des opérations...)
- Glue le reste des couches ensemble

3.3 Couche réplication

3.3.1 Modèle de données du document texte

MUTE propose plusieurs alternatives pour représenter le document texte. MUTE permet de soit utiliser une implémentation de LogootSplit [1], soit de RenamableLogootSplit[1] ou soit de Dotted LogootSplit [2]. Ce choix est effectué via une valeur de configuration de l'application choisie au moment de son déploiement.

Le modèle de données utilisé interagit avec l'éditeur de texte par l'intermédiaire de d'opérations textes. Lorsque l'utilisateur effectue des modifications locales, celles-ci sont détectées et mises sous la forme d'opérations textes. Elles sont transmises au modèle de données, qui les intègre alors à la structure de données répliquées. Le CRDT retourne en résultat l'opération distante à propager aux autres noeuds.

De manière complémentaire, lorsqu'une opération distante est délivrée au modèle de données, elle est intégrée par le CRDT pour actualiser son état. Le CRDT génère les opérations textes correspondantes et les transmet à l'éditeur de texte pour mettre à jour la vue.

3.3.2 Module de livraison des opérations

Dans le cadre de LogootSplit et de RenamableLogootSplit, le modèle de données utilisé pour représenter le document texte est couplé au composant `Sync`. Le rôle de ce composant est d'assurer le respect du modèle de livraison des opérations au CRDT. Pour cela, le module `Sync` doit implémenter les contraintes présentées dans la sous-section 1.4.4 et la sous-section 2.2.3.

Livraison des opérations en exactement un exemplaire

- Afin de respecter l'exactly-once delivery, doit identifier de manière unique chaque opération
- Pour cela, ajoute un dot à chaque opération
- dot est formé de l'identifiant du pair et d'un numéro séquentiel
  - numéro séquentiel différent de celui-ci utilisé par le CRDT, puisque celui doit augmenter avec chaque opération
- doit alors maintenir une structure de données représentant l'ensemble des opérations reçues par le pair
- à la réception d'une opération, vérifie si son dot est présent dans cette structure
- si absent, peut délivrer l'opération
- sinon, opération déjà délivrée précédemment, peut l'ignorer sans risque

1. Les deux implémentations proviennent de la librairie `mute-structs` : `https://github.com/coast-team/mute-structs`
2. Implémentation fournie par la librairie suivante : `https://github.com/coast-team/dotted-logootsplit`

- pour maintenir l'ensemble des opérations reçues, plusieurs structures de données adaptées
- dans le cadre de MUTE, avons fait le choix d'utiliser un version vector
- nous permet de réduire à un dot par pair le surcoût en métadonnées du mécanisme
  - maintient seulement le dot le plus récent
- à la réception d'une opération, vérifie qu'il s'agit de la prochaine opération attendue, c.-à-d. que son le numéro séquentiel de son dot est le successeur du dernier dot enregistré pour ce pair
- si c'est le cas, délivre l'opération et met à jour l'entrée du pair dans le version vector avec ce nouveau numéro séquentiel
- si le dot est un dot futur, met en attente l'opération en attendant d'avoir reçu et délivré les opérations manquantes de ce pair
- sinon, si dot est déjà présent dans le version vector, ignore l'opération
- ce fonctionnement se traduit dans les faits par une livraison FIFO des opérations par noeud
- ajoute une contrainte non-nécessaire qui peut introduire des délais dans livraison des opérations si perd juste une opération d'un noeud
- nous paraît un compromis acceptable entre le surcoût du mécanisme de livraison et son impact sur l'expérience utilisateur
- une extension possible serait de remplacer cette structure par un *Interval Version Vector* [30], qui rend possible la livraison dans le désordre
- permettrait de supprimer la contrainte de livraison FIFO tout en permettant de compacter efficacement la représentation des opérations reçues par le noeud à terme

Livraison de l'opération *remove* après l'opération *insert*

- Pour livrer une opération *remove* après opérations *insert* correspondantes
  - besoin d'identifier les opérations *insert* concernées
  - de les ajouter en dépendances de l'opération *remove*
  - pour cela, ajoute leur dot à l'opération
  - à la livraison de l'opération, vérifie que les dots des opérations *insert* concernées sont présents dans version vector en plus de vérifier le dot de l'opération *remove* elle-même
  - si un dot est manquant, met l'opération en attente
- Pour identifier opérations *insert* correspondantes, plusieurs façons de procéder
- manière la plus précise est de parcourir le log des opérations à la recherche des opérations *insert* ajoutant les identifiants supprimés
- mais s'avère coûteux
- et incompatible avec un mécanisme tronquant le log des opérations en utilisant la stabilité

— dans MUTE, nous préférons ajouter le dot le plus récent du noeud auteur de l'identifiant supprimé comme dépendance de l'opération

— nous permet de réduire le surcoût computationnel du calcul des dépendances d'une opération *remove*

— en contrepartie, il s'agit de dépendances approximatives

— retarde probablement la livraison de l'opération par rapport aux dépendances véritables

— mais là aussi, nous paraît un compromis acceptable entre surcoût du mécanisme de livraison et expérience utilisateur

Livraison des opérations après l'opération *rename* introduisant leur époque

— Afin de livrer les opérations après l'opération *rename* qui introduit leur époque

  — besoin d'ajouter son dot en dépendance des opérations

  — pour cela, nécessaire de garder le dot l'opération *rename* qui a introduit l'époque courante et de l'ajouter à chaque nouvelle opé

  — mais historiquement, `DocService` et `Sync` complètement séparés

  — `Sync` n'a pas moyen d'interroger `DocService` pour récupérer l'époque courante

  — a dû donc trouver une alternative

  — a implémenté la solution suivante

  — maintient un vecteur de dots des dernières opérations *rename*

  — ajoute le contexte causale d'une opération *rename* en tant que dépendances

  — lors de la livraison d'une opération *rename*, utilise ce contexte causale pour retirer du vecteur de dots des opés *rename* les dots qui sont couverts par l'opération

  — puis ajoute le dot de la nouvelle opé *rename*

  — lors de l'émission d'une opération *insert* ou *remove*, ajoute en dépendances les dots du vecteur d'opérations *rename*

— Garantit ainsi livraison *epoch-based* des opérations, sans reposer sur $<_\varepsilon$

— mais nécessite livraison causale des opérations *rename*

— ces opérations étant rares, le surcoût en métadonnées et le potentiel délai introduit paraissent acceptables

Matthieu: TODO : Voir pour une autre implémentation. Implémentation actuelle dûe à la séparation entre `DocService` *et* `Sync` *: `Sync` n'a pas de moyen de connaître l'époque courante. Donc peut pas cibler et ajouter uniquement le dot de l'opération* rename *correspondante.*

Mécanisme d'anti-entropie

- Réseau peut perdre des messages. Doit rediffuser les messages perdus pour assurer la livraison à terme. Pour cela, implémente le mécanisme d'anti-entropie basé sur [41].
- Consiste à synchroniser deux à deux les pairs. De manière périodique, noeud choisit un autre pair de manière aléatoire. Lui envoie une représentation de son état courant, c.-à-d. son version vector.
- À la réception de ce message, le second pair compare le version vector reçu par rapport au sien. Identifie de cette manière les dots des opérations qu'il a reçues mais qui sont inconnues pour l'initiateur de la synchronisation et récupère les opérations correspondantes depuis le log des opérations. Identifie aussi les dots des opérations connues par l'initiateur de la synchronisation mais pas par lui pour les demander en retour. Renvoie alors une réponse composée des opérations récupérées et des dots des opérations demandées.
- À la réception de la réponse, le noeud initiateur intègre les opérations reçues. Parcourt son log des opérations à la recherche des opérations demandées par le second pair et les rediffuse sur le réseau.
- Mécanisme d'anti-entropie permet ainsi de garantir la livraison à terme de toutes les opérations et de compenser opérations perdues. Sert aussi comme mécanisme de synchronisation : à la connexion d'un pair, celui-ci utilise ce mécanisme pour récupérer le travail effectué depuis sa dernière connexion.
- Avantages sont sa simplicité et que réexploite le causal context.
- Principale limite est que nécessite par contre de maintenir et de parcourir périodiquement le log des opérations pour répondre aux requêtes de synchronisation. Complexité en espace dépend du nombre de noeuds et du nombre d'opérations. Peut pas tronquer les opérations du log car nécessaire pour mettre à niveau les nouveaux pairs (à moins d'un mécanisme de snapshot comme évoqué dans sous-section 2.5.5).
- Serait intéressant de comparer à d'autres méthodes de synchronisation : mécanisme d'anti-entropie basé sur un Merkle Tree *Matthieu: TODO : retrouver ref*, synchronisation par états (state/delta-based CRDTs).

3.3.3 Métadonnées

- Titre (Simple LWW-Register)
- Mode de chiffrement (fixe)

3.3.4 Collaborateurs

- Implémente Swim [19]
- Découple protocole de détection des failures du protocole de diffusion de l'évolution du groupe
- Protocole de détection des failures

  - Basé sur un système de rounds, basés sur un interval de temps
  - À chaque round, chaque pair probe de manière aléatoire un autre pair
  - Si pas de réponse, demande à un autre pair de le contacter
  - Si pas de réponse par leur intermédiaire, pair devient suspect
  - Si toujours pas de nouvelles du pair après un certain temps, le considère déconnecté
- Protocole de diffusion de l'évolution du groupe
  - Plutôt que de diffuser chaque évolution du groupe, adopte un modèle de diffusion épidémique
  - Piggyback les évolutions aux messages du protocole de détection des failures
- Modifie le fonctionnement du protocole pour en faire un CRDT
- Afin de permettre un nouveau pair de récupérer instantanément état courant du groupe
- Autorise aussi un pair à se déconnecter puis reconnecter en modifiant l'ordre de priorité entre les différents messages
  - Dans protocole original, un pair déconnecté doit revenir sous une nouvelle identité
  - Afin de maintenir l'identifiant du pair, notamment pour ses opérations sur le document

3.3.5 Curseurs

- Repose sur des identifiants pour indiquer la position des curseurs
- Vecteur de LWW-Registers, chaque LWW-Register étant associé à un pair actuellement connecté *Matthieu: NOTE : C'est vrai ça ? J'ai un doute sur le fait qu'on avait mis en place un CRDT pour cette structure. À vérifier.*

3.4 Couche sécurité

3.4.1 Objectifs

- Souhaite garantir confidentialité, authenticité et intégrité des messages. Pour cela, vise chiffrement de bout en bout.
- Souhaite aussi garantir backward et forward secrecies. Backward : nouveau pair ne peut pas déchiffrer les messages précédemments envoyés. Forward : ancien pair ne peut pas déchiffrer les messages envoyés après son départ.
- Besoin de mettre à jour clé au fur et à mesure des évolutions du groupe. Comment procéder à l'évolution de la clé en distribué ?

3.4.2 Approche choisie

— Intègre un mécanisme de chiffrement à base de clé de groupe. Implémente protocole Burmester - Desmedt pour établir la clé de chiffrement de groupe. Nécessite clés publiques des différents participants. Besoin d'authentifier les clés publiques pour éviter man-in-the-middle attack.

Trusternity

— Mécanisme d'audit des PKI. [33, 34]

Protocole Burmester-Desmedt

— Permet d'établir la clé de chiffrement de groupe. [15].

3.4.3 Limites

— Plusieurs limites à ce protocole.

— Computationnellement lourd. *Matthieu: TODO : Limite provenant d'une slide de JP, voir ce qui justifie cette remarque*

— Requiert participation de l'ensemble des noeuds lors évolution du groupe. Peut être effectué de manière asynchrone. Mais que faire en attendant que l'ensemble des noeuds ? Peu adapté dans un système hautement dynamique. Va passer son temps à regénérer une nouvelle clé.

— Pour éviter ce problème, considère que le groupe est l'ensemble des noeuds connectés. Mais implique de générer une nouvelle clé à chaque connexion et déconnexion. Peu efficace pour les collaborations où le groupe est stable, mais avec des connexions/déconnexions.

3.4.4 Perspectives

Contrôle d'accès

— Pour le moment, n'importe quel utilisateur ayant l'URL du document peut y accéder dans MUTE

— Pour des raisons de confidentialité, peut vouloir contrôler quels utilisateurs ont accès à un document

— Nécessite l'implémentation de liste de contrôle d'accès

— Mais s'agit d'une tâche complexe dans le cadre d'un système distribué

— Peut s'inspirer des travaux réalisés au sein de la communautée CRDTs [56] pour cela

Détection et éviction de pairs malhonnêtes

- À l'heure actuelle, MUTE suppose qu'ensemble des collaborateurs honnêtes
- Vulnérable à plusieurs types d'attaques par des adversaires byzantins, tel que l'équivoque
- Ce type d'attaque peut provoquer des divergences durables et faire échouer des collaborations
- Dans [22, 21], Elvinger propose un mécanisme permettant de maintenir des logs authentifiés dans un système distribué
- Les logs authentifiés permettent de mettre en lumière les comportements malveillants des adversaires et de limiter le nombre d'actions malveillantes qu'ils peuvent effectuer avant d'être évincer
- Implémenter ce mécanisme permettrait de rendre compatible MUTE avec des environnements avec adversaires byzantins
- Nécessiterait tout de même de faire évoluer le CRDT pour résoudre les équivoques détectés

3.5 Couche réseau

3.5.1 Netflux

- Réseau P2P
- Interface uniforme permettant d'interagir à la fois avec des navigateurs et des bots
- Connectent les noeuds en utilisant la technologie WebRTC
- Connectent les bots en utilisant la technologie WebSocket
- Repose sur l'utilisation d'un signaling server pour permettre aux pairs de rejoindre la collaboration
- Topologie maillée

3.5.2 Pulsar

- Log-based message broker
- Propose plusieurs modes de fonctionnement
- En mode log-based message broker, maintient l'ensemble des messages reçus dans un log
- Permet, lorsque utilisé pour diffuser les opérations, de conserver le log complet des opérations
- Permet alors à un nouveau noeud de récupérer l'ensemble des opérations connues et de reconstruire l'état courant du document, même si actuellement aucun autre pair n'est connecté

— En mode message broker, diffuse seulement les messages aux noeuds actuellement connectés au topic
— Permet de communiquer les messages transients (protocole d'établissement de la clé de groupe, heartbeat de Swim, mécanisme d'anti-entropie du document) sans polluer le log
— Pose néanmoins des questions de sécurité et d'utilisabilité
— Besoin de chiffrer E2E les opérations
— Dans ce cas
  — comment un nouveau pair peut obtenir la clé de chiffrement si les autres pairs ne sont pas connectés ? (à moins de revenir à un chiffrement à base de passphrase, avec les problèmes qui en découlent)
  — comment un nouveau pair peut relire les opérations chiffrées avec l'ancienne clé ?

3.6 Pistes d'amélioration et de recherche

3.6.1 Fusion de versions distantes d'un document collaboratif

3.6.2 Rôles et places des bots dans systèmes collaboratifs

— Stockage du document pour améliorer sa disponibilité
— Overleaf en P2P ?
— Comment réinsérer des bots dans la collaboration sans en faire des éléments centraux, sans créer des failles de confidentialité, et tout en rendant ces fonctionnalités accessibles ?

3.7 Conclusion

Chapitre 4

Conclusions et perspectives

Sommaire

4.1 Résumé des contributions 67
4.2 Perspectives 67
4.2.1 Définition de relations de priorité pour minimiser les traitements 67
4.2.2 Redéfinition de la sémantique du renommage en déplacement d'éléments 67
4.2.3 Définition de types de données répliquées sans conflits plus complexes 67

4.1 Résumé des contributions

4.2 Perspectives

4.2.1 Définition de relations de priorité pour minimiser les traitements

4.2.2 Redéfinition de la sémantique du renommage en déplacement d'éléments

4.2.3 Définition de types de données répliquées sans conflits plus complexes

Annexe A

Algorithmes RENAMEID

```
function RENIDLESSTHANFIRSTID(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
```

FIGURE A.1 – Remaining functions to rename an identifier

Annexe B

Algorithmes revertRenameId

```
function REVRENIDLESSTHANNEWFIRSTID(id, firstId, newFirstId)
    predNewFirstId ← createIdFromBase(newFirstId, -1)
    if isPrefix(predNewFirstId, id) then
        tail ← getTail(id, 1)
        if tail < firstId then
            return tail
        else
                                                ▷ id has been inserted causally after the rename op
            offset ← getLastOffset(firstId)
            predFirstId ← 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 ← 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
```

FIGURE B.1 – Remaining functions to revert an identifier renaming

Index

Voici un index

FiXme :

Notes :

10 : Matthieu : TODO : Montrer que cet ensemble d'identifiants est un ensemble dense, 8

11 : Matthieu : TODO : indiquer que le couple hAnodeId, nodeSeq$B$ permet d'identifier de manière unique la base d'un bloc ou d'un identifiant, 9

12 : Matthieu : NOTE : Pourrait définir dans cette sous-section la notion de séquence bien-formée, 11

13 : Matthieu : QUESTION : Ajouter quelques lignes ici sur comment faire ça en pratique (Ajout d'un dot aux opérations, maintien d'un dot store au niveau de la couche livraison, vérification que dot pas encore présent dans dot store avant de passer opération à la structure de données) ? Ou je garde ça pour le chapitre sur MUTE ?, 12

14 : Matthieu : QUESTION : Même que pour la exactly-once delivery, est-ce que j'explique ici comment assurer cette contrainte plus en détails (Ajout des dots des opérations *insert* en dépendances de l'opération *remove*, vérification que dots présents dans dot store avant de passer l'opération *remove* à la structure de données) ou je garde ça pour le chapitre sur MUTE ?, 13

15 : Matthieu : TODO : Ajouter une phrase pour expliquer que la croissance des identifiants impacte aussi le temps d'intégration des modifications, 14

16 : Matthieu : TODO : Trouver référence sur la stabilité causale dans systèmes dynamiques, 15

17 : Matthieu : TODO : Modifier exemple pour illustrer le cas de figure où on a besoin de MIN/MAX_TUPLE, 30

18 : Matthieu : TODO : Remplacer Figure 2.13 par un exemple avec plus d'opérations *rename* pour mieux faire apparaître les calculs et manipulations effectués sur les chemins dans l'*arbre des époques*, 32

19 : Matthieu : TODO : Trouver un autre terme que pointillé pour dotted, 36

1 : Matthieu : TODO : Mentionner TP1 et TP2, 4

20 : Matthieu : TODO : Trouver structure de données adaptée à l'*arbre des époques*. Besoin de pouvoir établir rapidement le chemin entre la racine et une époque. Pour cela, le mieux serait d'avoir accès directement à l'époque et qu'elle référencie l'époque parente. , 38

21 : Matthieu : NOTE : Une table de hachage correspond bien (ce que j'utilise dans l'implem). Mais pas le plus adapté pour la garbage collection des époques obsolètes (besoin de parcourir l'ensemble des clés et

de supprimer celles n'appartenant plus aux *époques requises*). , 38
22 : Matthieu : NOTE : Dans sous-section 2.3.5, je présente le principe du mécanisme de GC. Dans cette partie, je décris l'algo correspondant et l'évalue. Rédiger l'algo ? Dans quelle partie l'insérer dans ce cas ? , 40
23 : Matthieu : TODO : Développer ce paragraphe : noeuds doivent retrouver les blocs renommés à partir des données reçues. Pour cela, parcourent leur état. Suffit de retrouver un identifiant avec le même couple hAnodeId, nodeSeq$B$ pour reformer un bloc. Certains couples hAnodeId, nodeSeq$B$ peuvent avoir été supprimés en concurrence et ne plus être présent dans la séquence. Donc besoin d'aussi parcourir le log des opérations *remove* concurrentes., 50
24 : Matthieu : TODO : ajouter figure d'un epoch tree où une longue branche se fait remplacer par une époque isolée, 50
25 : Matthieu : TODO : Ajouter figure où noeud reçoit successivement plusieurs opérations *rename* concurrentes et procéde au renommage de son état à chaque fois, 51
26 : Matthieu : TODO : Étudier si y a un intérêt à privilégier la synchronisation basée sur l'intégration successive de toutes les opérations quand on a cette méthode de synchronisation par snapshot/checkpoint de possible, 53
27 : Matthieu : TODO : Ajouter conclusion à cette sous-section, 55
28 : Matthieu : Serait intéressant d'avoir une implémentation combinant LogootSplit et LSEQ pour vérifier si les contraintes sur la création de blocs dans LogootSplit ne sabotent pas la croissance polylogarithmique des identifiants de LSEQ, 55
29 : Matthieu : Peut aussi aborder les travaux de Jim Bauwens et Elisa Gonzalez Boix [12, 10, 11] sur l'accélération de la stabilité causale : ne concerne pas seulement les séquences, mais les operation-based CRDTs. Permet de tronquer le log des opérations mais aussi d'accélerer le mécanisme de GC de RGA (et le mien aussi), 55
2 : Matthieu : TODO : Spécification faible et forte des séquences répliquées, 4
30 : Matthieu : Peut aborder les travaux de Weidner, Miller et Meiklejohn [51, 50] qui combinent aussi CRDT et OT dans une certaine mesure. Pas vraiment dans le but de réduire les métadonnées du CRDT. Mais reste intéressant à présenter pour se différencier (eux proposent d'utiliser OT pour fusionner 2 CRDTs, moi pour ajouter une action qui est incompatible nativement avec les autres actions du CRDT), 55
31 : Matthieu : TODO : Insérer schéma de l'architecture d'une collaboration (noeuds, types de noeuds et lien), 58
32 : Matthieu : TODO : Insérer schéma de l'architecture logicielle d'un pair, 58
33 : Matthieu : TODO : Voir pour une autre implémentation. Implémentation actuelle dûe à la séparation entre `DocService` et `Sync` : `Sync` n'a pas de moyen de connaître l'époque courante. Donc peut pas cibler et ajouter uniquement le dot de l'opération *rename* correspondante. , 61
34 : Matthieu : TODO : retrouver ref, 62
35 : Matthieu : NOTE : C'est vrai ça ? J'ai un doute sur le fait qu'on avait mis en place un CRDT pour cette

structure. À vérifier., 63
36 : Matthieu : TODO : Limite provenant d'une slide de JP, voir ce qui justifie cette remarque, 64
3 : Matthieu : Faire le lien avec les travaux de Burckhardt [14] et les MRDTs [26], 5
4 : Matthieu : TODO : Ajouter forces, faiblesses et cas d'utilisation de cette approche, 6
5 : Matthieu : TODO : Ajouter référence mécanisme d'anti-entropie basé sur Merkle Tree, 6
6 : Matthieu : NOTE : Ajouter LogootSplit de manière sommaire aussi à cet endroit ?, 7
7 : Matthieu : TODO : Autres Sequence CRDTs à considérer : String-wise CRDT [58], Chronofold [23], 7
8 : Matthieu : TODO : Ajouter une relation d'ordre sur les tuples, 7
9 : Matthieu : TODO : Définir la notion de base (et autres fonctions utiles sur les identifiants ? genre isPrefix, concat, getTail...), 8
FiXme (Matthieu) :
Notes :
10 : TODO : Montrer que cet ensemble d'identifiants est un ensemble dense, 8
11 : TODO : indiquer que le couple hAnodeId, nodeSeq$B$ permet d'identifier de manière unique la base d'un bloc ou d'un identifiant, 9
12 : NOTE : Pourrait définir dans cette sous-section la notion de séquence bien-formée, 11
13 : QUESTION : Ajouter quelques lignes ici sur comment faire ça en pratique (Ajout d'un dot aux opérations, maintien d'un dot store au niveau de la couche livraison, vérification que dot pas encore présent dans dot store avant de passer opération à la structure de données) ? Ou je garde ça pour le chapitre sur MUTE ?, 12
14 : QUESTION : Même que pour la exactly-once delivery, est-ce que j'explique ici comment assurer cette contrainte plus en détails (Ajout des dots des opérations *insert* en dépendances de l'opération *remove*, vérification que dots présents dans dot store avant de passer l'opération *remove* à la structure de données) ou je garde ça pour le chapitre sur MUTE ?, 13
15 : TODO : Ajouter une phrase pour expliquer que la croissance des identifiants impacte aussi le temps d'intégration des modifications, 14
16 : TODO : Trouver référence sur la stabilité causale dans systèmes dynamiques, 15
17 : TODO : Modifier exemple pour illustrer le cas de figure où on a besoin de MIN/MAX_TUPLE, 30
18 : TODO : Remplacer Figure 2.13 par un exemple avec plus d'opérations *rename* pour mieux faire apparaître les calculs et manipulations effectués sur les chemins dans l'*arbre des époques*, 32
19 : TODO : Trouver un autre terme que pointillé pour dotted, 36
1 : TODO : Mentionner TP1 et TP2, 4
20 : TODO : Trouver structure de données adaptée à l'*arbre des époques*. Besoin de pouvoir établir rapidement le chemin entre la racine et une époque. Pour cela, le mieux serait d'avoir accès directement à l'époque et qu'elle référencie l'époque parente. , 38
21 : NOTE : Une table de hachage correspond bien (ce que j'utilise dans l'implem). Mais pas le plus adapté pour la garbage collection des époques obsolètes (besoin de parcourir l'ensemble des clés et de supprimer celles

n'appartenant plus aux *époques requises*). , 38
22 : NOTE : Dans sous-section 2.3.5, je présente le principe du mécanisme de GC. Dans cette partie, je décris l'algo correspondant et l'évalue. Rédiger l'algo ? Dans quelle partie l'insérer dans ce cas ? , 40
23 : TODO : Développer ce paragraphe : noeuds doivent retrouver les blocs renommés à partir des données reçues. Pour cela, parcourent leur état. Suffit de retrouver un identifiant avec le même couple hAnodeId, nodeSeq$B$ pour reformer un bloc. Certains couples hAnodeId, nodeSeq$B$ peuvent avoir été supprimés en concurrence et ne plus être présent dans la séquence. Donc besoin d'aussi parcourir le log des opérations *remove* concurrentes., 50
24 : TODO : ajouter figure d'un epoch tree où une longue branche se fait remplacer par une époque isolée, 50
25 : TODO : Ajouter figure où noeud reçoit successivement plusieurs opérations *rename* concurrentes et procéde au renommage de son état à chaque fois, 51
26 : TODO : Étudier si y a un intérêt à privilégier la synchronisation basée sur l'intégration successive de toutes les opérations quand on a cette méthode de synchronisation par snapshot/checkpoint de possible, 53
27 : TODO : Ajouter conclusion à cette sous-section, 55
28 : Serait intéressant d'avoir une implémentation combinant LogootSplit et LSEQ pour vérifier si les contraintes sur la création de blocs dans LogootSplit ne sabotent pas la croissance polylogarithmique des identifiants de LSEQ, 55
29 : Peut aussi aborder les travaux de Jim Bauwens et Elisa Gonzalez Boix [12, 10, 11] sur l'accélération de la stabilité causale : ne concerne pas seulement les séquences, mais les operation-based CRDTs. Permet de tronquer le log des opérations mais aussi d'accélerer le mécanisme de GC de RGA (et le mien aussi), 55
2 : TODO : Spécification faible et forte des séquences répliquées, 4
30 : Peut aborder les travaux de Weidner, Miller et Meiklejohn [51, 50] qui combinent aussi CRDT et OT dans une certaine mesure. Pas vraiment dans le but de réduire les métadonnées du CRDT. Mais reste intéressant à présenter pour se différencier (eux proposent d'utiliser OT pour fusionner 2 CRDTs, moi pour ajouter une action qui est incompatible nativement avec les autres actions du CRDT), 55
31 : TODO : Insérer schéma de l'architecture d'une collaboration (noeuds, types de noeuds et lien), 58
32 : TODO : Insérer schéma de l'architecture logicielle d'un pair, 58
33 : TODO : Voir pour une autre implémentation. Implémentation actuelle dûe à la séparation entre `DocService` et `Sync` : `Sync` n'a pas de moyen de connaître l'époque courante. Donc peut pas cibler et ajouter uniquement le dot de l'opération *rename* correspondante. , 61
34 : TODO : retrouver ref, 62
35 : NOTE : C'est vrai ça ? J'ai un doute sur le fait qu'on avait mis en place un CRDT pour cette structure. À vérifier., 63
36 : TODO : Limite provenant d'une slide de JP, voir ce qui justifie cette remarque, 64
3 : Faire le lien avec les travaux de Burckhardt [14] et les MRDTs [26],

5
4 : TODO : Ajouter forces, faiblesses et cas d'utilisation de cette approche, 6
5 : TODO : Ajouter référence mécanisme d'anti-entropie basé sur Merkle Tree, 6
6 : NOTE : Ajouter LogootSplit de manière sommaire aussi à cet endroit ?, 7
7 : TODO : Autres Sequence CRDTs à considérer : String-wise CRDT [58], Chronofold [23], 7
8 : TODO : Ajouter une relation d'ordre sur les tuples, 7
9 : TODO : Définir la notion de base (et autres fonctions utiles sur les identifiants ? genre isPrefix, concat, getTail...), 8

Bibliographie

[1] D. 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.

[2] Mehdi Ahmed-Nacer et al. « Evaluating CRDTs for Real-time Document Editing ». In : *11th ACM Symposium on Document Engineering*. Sous la dir. d'ACM. Mountain View, California, United States, sept. 2011, p. 103–112. doi : 10.1145/2034691.2034717. url : https://hal.inria.fr/inria-00629503.

[3] Paulo Sérgio Almeida, Ali Shoker et Carlos Baquero. « Delta state replicated data types ». In : *Journal of Parallel and Distributed Computing* 111 (jan. 2018), p. 162–173. issn : 0743-7315. doi : 10.1016/j.jpdc.2017.08.003. url : http://dx.doi.org/10.1016/j.jpdc.2017.08.003.

[4] 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.

[5] Luc André et al. « 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.

[6] Automerge. *Automerge : data structures for building collaborative applications in Javascript*. url : https://github.com/automerge/automerge.

[7] Carlos Baquero, Paulo Sergio Almeida et Ali Shoker. *Pure Operation-Based Replicated Data Types*. 2017. arXiv : 1710.04469 [cs.DC].

[8] 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.

[9] 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.

[10] 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.

[11] Jim Bauwens et Elisa Gonzalez Boix. « From Causality to Stability : Understanding and Reducing Meta-Data in CRDTs ». In : *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*. New York, NY, USA : Association for Computing Machinery, 2020, p. 3–14. isbn : 9781450388535. url : https://doi.org/10.1145/3426182.3426183.

[12] Jim Bauwens et Elisa Gonzalez Boix. « Memory Efficient CRDTs in Dynamic Environments ». In : *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2019. Athens, Greece : Association for Computing Machinery, 2019, p. 48–57. isbn : 9781450369879. doi : 10.1145/3358504.3361231. url : https://doi.org/10.1145/3358504.3361231.

[13] Loïck Briot, Pascal Urso et Marc Shapiro. « High Responsiveness for Group Editing CRDTs ». In : *ACM International Conference on Supporting Group Work*. Sanibel Island, FL, United States, nov. 2016. doi : 10.1145/2957276.2957300. url : https://hal.inria.fr/hal-01343941.

[14] Sebastian Burckhardt et al. « 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.

[15] 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.

[16] Concordant. *Concordant*. url : http://www.concordant.io/.

[17] The SyncFree Consortium. *AntidoteDB : A planet scale, highly available, transactional database*. url : http://antidoteDB.eu/.

[18] 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.

[19] A. Das, I. Gupta et A. 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.

[20] Kevin De Porre et al. « CScript : A distributed programming language for building mixed-consistency applications ». In : *Journal of Parallel and Distributed Computing volume 144* (oct. 2020), p. 109–123. ISSN : 0743-7315. DOI : 10.1016/j.jpdc.2020.05.010.

[21] VIctorien Elvinger. « Réplication sécurisée dans les infrastructures pair-à-pair de collaboration ». Theses. Université de Lorraine, juin 2021. URL : https://hal.univ-lorraine.fr/tel-03284806.

[22] Victorien Elvinger, Gérald Oster et François Charoy. « Prunable Authenticated Log and Authenticable Snapshot in Distributed Collaborative Systems ». In : *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2018, p. 156–165.

[23] Victor Grishchenko et Mikhail Patrakeev. « Chronofold : A Data Structure for Versioned Text ». 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 : 10.1145/3380787.3393680. URL : https://doi.org/10.1145/3380787.3393680.

[24] Peter van Hardenberg et Martin Kleppmann. « PushPin : Towards Production-Quality Peer-to-Peer Collaboration ». In : *7th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC 2020. ACM, avr. 2020. DOI : 10.1145/3380787.3393683.

[25] Claudia-Lavinia Ignat. « Maintaining consistency in collaboration over hierarchical documents ». Thèse de doct. ETH Zurich, 2006.

[26] Gowtham Kaki et al. « Mergeable Replicated Data Types ». In : *Proc. ACM Program. Lang.* 3.OOPSLA (oct. 2019). DOI : 10.1145/3360580. URL : https://doi.org/10.1145/3360580.

[27] Martin Kleppmann et Alastair R. Beresford. « A Conflict-Free Replicated JSON Datatype ». In : *IEEE Transactions on Parallel and Distributed Systems* 28.10 (oct. 2017), p. 2733–2746. ISSN : 1045-9219. DOI : 10.1109/tpds.2017.2697382. URL : http://dx.doi.org/10.1109/TPDS.2017.2697382.

[28] Martin Kleppmann et al. « 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.

[29] Christopher Meiklejohn et Peter Van Roy. « Lasp : A Language for Distributed, Coordination-free Programming ». In : *17th International Symposium on Principles and Practice of Declarative Programming*. PPDP 2015. ACM, juil. 2015, p. 184–195. DOI : 10.1145/2790449.2790525.

[30] Madhavan Mukund, Gautham Shenoy et SP Suresh. « Optimized or-sets without ordering constraints ». In : *International Conference on Distributed Computing and Networking*. Springer. 2014, p. 227–241.

[31] Brice NÉDELEC, Pascal MOLLI et Achour MOSTEFAOUI. « CRATE : Writing Stories Together with our Browsers ». In : *25th International World Wide Web Conference*. WWW 2016. ACM, avr. 2016, p. 231–234. DOI : 10.1145/2872518.2890539.

[32] 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.

[33] 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.

[34] Hoang-Long NGUYEN et al. « 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.

[35] Petru NICOLAESCU et al. « Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types ». In : *19th International Conference on Supporting Group Work*. GROUP 2016. ACM, nov. 2016, p. 39–49. DOI : 10.1145/2957276.2957310.

[36] Petru NICOLAESCU et al. « Yjs : A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types ». In : *15th International Conference on Web Engineering*. ICWE 2015. Springer LNCS volume 9114, juin 2015, p. 675–678. DOI : 10.1007/978-3-319-19890-3_55. URL : http://dbis.rwth-aachen.de/~derntl/papers/preprints/icwe2015-preprint.pdf.

[37] 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.

[38] 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.

[39] Matthieu NICOLAS et al. « 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, août 2017, p. 1–4. DOI : 10.18420/ecscw2017_p5. URL : https://hal.inria.fr/hal-01655438.

[40] Gérald Oster et al. « 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.

[41] D. S. Parker et al. « 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.

[42] Nuno Preguica et al. « 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.

[43] Riak. *Riak KV*. URL : http://riak.com/.

[44] Hyun-Gul Roh et al. « 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.

[45] 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.

[46] Marc Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, jan. 2011, p. 50. URL : https://hal.inria.fr/inria-00555588.

[47] Marc Shapiro et al. « 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.

[48] 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.

[49] Chengzheng Sun et al. « Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems ». In : *ACM Trans. Comput.-Hum. Interact.* 5.1 (mar. 1998), p. 63–108. ISSN : 1073-0516. DOI : 10.1145/274444.274447. URL : https://doi.org/10.1145/274444.274447.

[50] Matthew Weidner, Heather Miller et Christopher Meiklejohn. « Composing and Decomposing Op-Based CRDTs with Semidirect Products ». In : *Proc. ACM Program. Lang.* 4.ICFP (août 2020). DOI : 10.1145/3408976. URL : https://doi.org/10.1145/3408976.

[51] Matthew Weidner, Heather Miller et Christopher Meiklejohn. « Composing and Decomposing Op-Based CRDTs with Semidirect Products : (Summary) ». 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 : 10.1145/3380787.3393687. url : https://doi.org/10.1145/3380787.3393687.

[52] 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.ieeecomputersociety.org/10.1109/ICDCS.2009.75.

[53] Stéphane Weiss, Pascal Urso et Pascal Molli. « Logoot-Undo : Distributed Collaborative Editing System on P2P Networks ». In : *IEEE Transactions on Parallel and Distributed Systems* 21.8 (août 2010), p. 1162–1174. doi : 10.1109/TPDS.2009.173. url : https://hal.archives-ouvertes.fr/hal-00450416.

[54] Stéphane Weiss, Pascal Urso et Pascal Molli. « Wooki : a P2P Wiki-based Collaborative Writing Tool ». In : t. 4831. Déc. 2007. isbn : 978-3-540-76992-7. doi : 10.1007/978-3-540-76993-4_42.

[55] C. Wu et al. « Anna : A KVS for Any Scale ». In : *IEEE Transactions on Knowledge and Data Engineering* 33.2 (2021), p. 344–358. doi : 10.1109/TKDE.2019.2898401.

[56] Elena Yanakieva et al. « Access Control Conflict Resolution in Distributed File Systems Using 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 : 10.1145/3447865.3457970. url : https://doi.org/10.1145/3447865.3457970.

[57] Yjs. *Yjs : A CRDT framework with a powerful abstraction of shared data*. url : https://github.com/yjs/yjs.

[58] Weihai Yu. « A String-Wise CRDT for Group Editing ». In : *Proceedings of the 17th ACM International Conference on Supporting Group Work*. GROUP '12. Sanibel Island, Florida, USA : Association for Computing Machinery, 2012, p. 141–144. isbn : 9781450314862. doi : 10.1145/2389176.2389198. url : https://doi.org/10.1145/2389176.2389198.

[59] Weihai Yu et Claudia-Lavinia Ignat. « Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge ». In : *IEEE International Conference on Smart Data Services, 2020 IEEE World Congress on Services*. Beijing, China, oct. 2020. url : https://hal.inria.fr/hal-02983557.

[60] 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.

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