UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

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

THÈSE

présentée et soutenue publiquement le TODO : Définir une date

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Matthieu Nicolas

Composition du jury

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

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

Mis en page avec la classe thesul.

Remerciements

WIP

WIP

Sommaire

Chapitre 1
Introduction **1**

1.1 Contexte 1
1.2 Questions de recherche et contributions 5
1.2.1 Ré-identification sans coordination synchrone pour les Conflict-free Replicated Data Types (CRDTs) pour le type Séquence 5
1.2.2 Éditeur de texte collaboratif Pair-à-Pair (P2P) temps réel chiffré de bout en bout 6
1.3 Plan du manuscrit 8
1.4 Publications 8

Chapitre 2
État de l'art **11**

2.1 Modèle du système 12
2.2 Types de données répliquées sans conflits 13
2.2.1 Sémantiques en cas de conflits 17
2.2.2 Modèles de synchronisation 21
2.3 Séquences répliquées sans conflits 30
2.3.1 Approche à pierres tombales 33
2.3.2 Approche à identifiants densément ordonnés 41
2.3.3 Synthèse 49
2.4 LogootSplit 52
2.4.1 Identifiants 52
2.4.2 Aggrégation dynamique d'élements en blocs 53
2.4.3 Modèle de données 55
2.4.4 Modèle de livraison 57
2.4.5 Limites 60

2.5 Mitigation du surcoût des séquences répliquées sans conflits 62
2.5.1 Mécanisme de Garbage Collection (GC) des pierres tombales 62
2.5.2 Ré-équilibrage de l'arbre des identifiants de position 63
2.5.3 Réduction de la croissance des identifiants de position 63
2.5.4 Synthèse 64
2.6 Synthèse 65
2.7 Proposition 65

Chapitre 3
MUTE, un éditeur de texte web collaboratif P2P temps réel chiffré de bout en bout 67

3.1 Présentation 70
3.1.1 Objectifs 70
3.1.2 Fonctionnalités 70
3.1.3 Architecture système 72
3.1.4 Architecture logicielle 73
3.2 Couche interface utilisateur 75
3.3 Couche réplication 76
3.3.1 Modèle de données du document texte 76
3.3.2 Collaborateur-rices 77
3.3.3 Curseurs 81
3.4 Couche livraison 81
3.4.1 Livraison des opérations en exactement un exemplaire 82
3.4.2 Livraison de l'opération *remove* après l'opération *insert* 84
3.4.3 Livraison des opérations après l'opération *rename* introduisant leur époque 86
3.4.4 Livraison des opérations à terme 88
3.5 Couche réseau 89
3.5.1 Établissement d'un réseau P2P entre navigateurs 89
3.5.2 Topologie réseau 91
3.6 Couche sécurité 91
3.7 Conclusion 93

Chapitre 4
Conclusions et perspectives **95**

4.1 Résumés des contributions 95
4.1.1 Réflexions sur l'état de l'art des CRDTs 95
4.1.2 Ré-identification sans coordination pour les CRDTs pour Séquence 97
4.1.3 Éditeur de texte collaboratif P2P chiffré de bout en bout 99
4.2 Perspectives 101
4.2.1 Définition de relations de priorité pour minimiser les traitements . . 101
4.2.2 Détection et fusion manuelle de versions distantes 102
4.2.3 Étude comparative des différents modèles de synchronisation pour CRDTs 106
4.2.4 Approfondissement du patron de conception de Pure Operation-Based CRDTs 108

Annexe A
Entrelacement d'insertions concurrentes dans Treedoc

Annexe B
Algorithmes renameId

Annexe C
Algorithmes revertRenameId

Index **117**

Bibliographie

Table des figures

1.1 Caption for lfs-properties 3

2.1 Spécification algébrique du type abstrait usuel Ensemble 14
2.2 Modifications concurrentes d'un Ensemble répliqué provoquant un conflit suite à l'ajout et la suppression d'un même élément 15
2.3 Résolution du conflit en utilisant la sémantique *Last-Writer-Wins* (LWW) 17
2.4 Résolution du conflit en utilisant la sémantique *Multi-Value* (MV) 18
2.5 Résolution du conflit en utilisant soit la sémantique *Add-Wins* (AW), soit la sémantique *Remove-Wins* (RW) 19
2.6 Résolution du conflit en utilisant la sémantique *Causal-Length* (CL) 20
2.7 Modifications en concurrence d'un Ensemble répliqué par les noeuds A et B 21
2.8 Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par états 23
2.9 Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par opérations 25
2.10 Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par différences d'états 27
2.11 Représentation de la séquence "HELLO" 30
2.12 Spécification algébrique du type abstrait usuel Séquence 30
2.13 Modifications concurrentes d'une séquence 31
2.14 Modifications concurrentes d'une séquence répliquée WOOT 35
2.15 Modifications concurrentes d'une séquence répliquée Replicated Growable Array (RGA) 38
2.16 Entrelacement d'éléments insérés de manière concurrente 40
2.17 Identifiants de positions 42
2.18 Identifiants de position avec désambiguateurs 43
2.19 Modifications concurrentes d'une séquence répliquée Treedoc 43
2.20 Modifications concurrentes d'une séquence répliquée Logoot 47
2.21 Représentation d'une séquence LogootSplit contenant les éléments "HLO" 54
2.22 Spécification algébrique du type abstrait LogootSplit 55
2.23 Modifications concurrentes d'une séquence répliquée LogootSplit 56
2.24 Résurgence d'un élément supprimé suite à la relivraison de son opération *ins* 58
2.25 Non-effet de l'opération *rmv* car reçue avant l'opération *ins* correspondante 59
2.26 Insertion menant à une augmentation de la taille des identifiants 60
2.27 Insertion menant à une augmentation de la taille des identifiants 61

2.28 Taille du contenu comparé à la taille de la séquence LogootSplit 62

3.1 Capture d'écran d'une session d'édition collaborative avec MUTE 71
3.2 Capture d'écran de la liste des documents. 72
3.3 Architecture système de l'application MUTE 72
3.4 Architecture logicielle de l'application MUTE 74
3.5 Exécution du mécanisme de détection des défaillances par le noeud C pour tester le noeud B 78
3.6 Gestion de la livraison en exactement un exemplaire des opérations 83
3.7 Gestion de la livraison des opérations *remove* après les opérations *insert* correspondantes 85
3.8 Gestion de la livraison des opérations après l'opération *rename* qui introduit leur époque 87
3.9 Utilisation du mécanisme d'anti-entropie par le noeud C pour se synchroniser avec le noeud B 88
3.10 Architecture système pour la couche réseau de MUTE 90
3.11 Architecture système pour la couche sécurité de MUTE 92

A.1 Modifications concurrentes d'une séquence Treedoc résultant en un entrelacement 111

Chapitre 1

Introduction

Sommaire

1.1 Contexte 1
1.2 Questions de recherche et contributions 5
1.2.1 Ré-identification sans coordination synchrone pour les CRDTs pour le type Séquence 5
1.2.2 Éditeur de texte collaboratif P2P temps réel chiffré de bout en bout 6
1.3 Plan du manuscrit 8
1.4 Publications 8

1.1 Contexte

L'évolution des technologies du web a conduit à l'avènement de ce qui est communément appelé le Web 2.0. La principale caractéristique de ce média est la possibilité aux utilisateur-rices non plus seulement de le consulter, mais aussi d'y contribuer.

Ces nouvelles fonctionnalités ont permis l'apparition d'applications incitant les utilisateur-rices à créer et partager leur propre contenu, ainsi que d'échanger avec d'autres utilisateur-rices à ce sujet. [1] définit ce type d'applications, c.-à-d. les *réseaux sociaux*, de la manière suivante :

Définition 1. Un réseau social est une application respectant les critères suivants :

(i) Elle prend la forme d'une application interactive.

(ii) Elle permet à ses utilisateur-rices de produire et de partager leur propre contenu. Ce contenu généré par les utilisateur-rices constitue le contenu principal de l'application.

(iii) Elle permet à ses utilisateur-rices de possèder et de maintenir leur propre profil sur la plateforme.

(iv) Elle encourage ses utilisateur-rices à étendre leur réseau social en se connectant à d'autres utilisateur-rices, voire en créant ou en s'intégrant à des communautés.

De nos jours, les réseaux sociaux représentent les applications les plus populaires du paysage internet, e.g. Facebook compte 2,9 millards d'utilisateur-rices par mois [2], Youtube 2,5 milliards [2], Wikipedia 788 millions [3] ou encore Quora 300 millions [2].

La démocratisation de ces applications présente plusieurs bienfaits. Notamment, nous notons que les réseaux sociaux permettent, en diminuant le coût individuel pour tout à chacun-e de contribuer, d'améliorer la diffusion de l'information [4, 5]. Aussi, ils contribuent à la création de communautés [6]. Finalement, en permettant à chacun-e de partager son savoir, ils permettent la création de bases de connaissances complètes [7, 8].

Cependant, la conception de réseaux sociaux fait face à de nombreux défis. Notamment, ces applications doivent assurer leur haute disponibilité, tolérance aux pannes et capacité de passage à l'échelle en raison de leur popularité et importance dans notre quotidien.

Définition 2 (Disponibilité)**.** La disponibilité d'un système indique sa capacité à répondre à tout moment à une requête d'un-e utilisateur-rice.

Définition 3 (Tolérance aux pannes)**.** La tolérance aux pannes d'un système indique sa capacité à continuer à répondre aux requêtes malgré l'absence de réponse d'un ou plusieurs de ses composants.

Définition 4 (Capacité de passage à l'échelle)**.** La capacité de passage à l'échelle d'un système indique sa capacité à traiter un volume toujours plus conséquent de requêtes.

Pour cela, ces systèmes adoptent une architecture décentralisée [1], c.-à-d. une architecture reposant sur un ensemble de serveurs qui se répartissent la charge de travail et les tâches. Malgré ce que le nom de cette architecture peut suggérer, il convient de noter que de manière globale les serveurs jouent toujours un rôle central dans les systèmes décentralisés. Par exemple, les serveurs servent à authentifier les utilisateur-rices, à stocker leurs données ou encore à assurer la communication entre utilisateur-rices.

Additionnellement, il convient de préciser que ces serveurs ne sont pas une ressource libre. En effet, ils sont mis à disposition et maintenus par la ou les organisations qui proposent le réseau social. Ces organisations font alors office d'*autorités centrales* du système, e.g. en se portant garantes de l'identité des utilisateur-rices ou encore de l'authenticité d'un contenu.

De part le rôle que jouent les serveurs dans les systèmes décentralisés, ces derniers échouent à assurer un second ensemble de propriétés, que nous jugeons néanmoins fondamentales :

Définition 5 (Confidentialité des données)**.** La confidentialité des données d'un système indique sa capacité à garantir à ses utilisateur-rices que leurs données ne seront pas accessibles par des tiers non autorisés ou par le système lui-même.

Définition 6 (Souveraineté des données)**.** La souveraineté des données d'un système indique sa capacité à garantir à ses utilisateur-rices leur maîtrise de leurs données, c.-à-d. leur capacité à les consulter, modifier, partager, exporter ; supprimer ou encore à décider de l'usage qui en est fait.

1. Nous utilisons la classification présentée dans [9] pour distinguer les architectures systèmes centralisées, décentralisées et distribuées.

Définition 7 (Pérennité)**.** La pérennité d'un système indique sa capacité à garantir à ses utilisateur-rices son fonctionnement continu dans le temps.

Définition 8 (Résistance à la censure)**.** La résistance à la censure d'un système indique sa capacité à garantir à ses utilisateur-rices son fonctionnement malgré des actions de contrôle de l'information par des autorités.

Ainsi, les utilisateur-rices des réseaux sociaux prennent, de manière consciente ou non, le risque que ces propriétés soient transgressées par les autorités auxquelles appartiennent ces applications ou par des tiers avec lesquelles ces autorités interagissent, e.g. des gouvernements.

Qui plus est, il convient de noter que les autorités auxquelles appartiennent ces applications, e.g. des entreprises, possèdent leurs propres intérêts, e.g. leur propre profit. De nombreux faits d'actualité ont malheureusement montré que ces autorités ont tendance à privilégier leurs intérêts, quitte à prendre des décisions s'avérant nocives pour une partie de leurs utilisateur-rices. Nous pouvons par exemple noter la mise en avant de contenu à caractère raciste [10], la non-modération de contenu misogyne [11], l'encouragement à la production de contenu toxique [12] ou l'inaction et même entrave à la correction des effets délétères de son réseau social sur la santé mentale de ses utilisatrices [13].

Ainsi, la présence d'autorités centrales dans les réseaux sociaux représente un danger pour les utilisateur-rices. Il nous paraît alors fondamental de proposer des moyens technologiques pour concevoir et déployer des réseaux sociaux alternatifs qui minimisent le rôle des autorités centrales, voire l'éliminent.

Dans cette optique, une piste de recherche que nous jugeons intéressante à étudier est celle présentée dans [14]. Dans ce travail, les auteurs proposent un nouveau paradigme de conception d'applications, nommées Local-First Softwares (LFS). Ce type d'applications se démarque de ceux existants, e.g. les applications basées sur le cloud, par la place centrale donnée aux utilisateur-rices et leurs propres appareils, les éventuels serveurs étant relegués qu'à de simples rôles de support.

[14] identifie 7 propriétés que doivent satisfaire les applications LFS, illustrées dans la Figure 1.1. Ces propriétés se recoupent en grande partie avec celles que nous avons

	1. Fast	2. Multi-device	3. Offline	4. Collaboration	5. Longevity	6. Privacy	7. User control
???	✔	✔	✔	✔	✔	✔	✔

FIGURE 1.1 – Liste des propriétés visées par les LFS [2]

identifiées précédemment :

(i) Le fonctionnement en mode hors-ligne et le fonctionnement avec une latence minimale sont tous deux assurés en privilégiant la disponibilité [15] (Définition 2).

(ii) Le respect de la vie privée des utilisateur-rices correspond à la propriété de confidentialité (Définition 5).

2. Source : `https://www.inkandswitch.com/local-first/#towards-a-better-future`

(iii) Le contrôle des utilisateur-rices sur leurs données correspond à la propriété de souveraineté (Définition 6).

(iv) La longétivité de l'application correspond aux propriétés de pérennité (Définition 7) et de résistance à la censure (Définition 8).

Notons qu'il découle de ces propriétés que les applications LFS sont fondamentalement P2P.

De manière similaire, observons que ce paradigme met en lumière une dernière propriété des applications LFS :

Définition 9 (Collaborativité)**.** La collaborativité d'un système indique sa capacité à supporter ses utilisateur-rices dans leurs processus de collaboration pour la réalisation de tâches.

Matthieu: TODO : Introduire notion d'agents artificiels/logiciels Nous précisons que nous considérons dans ce manuscrit qu'une collaboration peut prendre bien des formes. Une collaboration peut ainsi prendre la forme d'un échange entre utilisateur-rices, e.g. un fil de discussion sur une plateforme de questions et réponses, ou d'une édition collaborative d'un contenu, e.g. la rédaction d'une page de wiki.

Matthieu: TODO : Voir si angle écologique/réduction consommation d'énergie peut être pertinent.

Ainsi, [14] établit un paradigme de conception d'applications correspondant à notre vision. Cependant, de nombreuses problématiques de recherche identifiées dans ce travail sont encore non résolues et entravent la démocratisation des applications LFS.

Notamment, les applications LFS se doivent de répliquer les données entre appareils pour permettre :

(i) Le fonctionnement en mode hors-ligne et le fonctionnement avec une faible latence.

(ii) Le partage de contenu entre appareils d'un-e même utilisateur-rice.

(iii) Le partage de contenu entre utilisateur-rices pour la collaboration.

Cependant, compte tenu des propriétés visées par les applications LFS, plusieurs contraintes restreignent le choix des méthodes de réplication possibles. Ainsi, pour permettre le fonctionnement en mode hors-ligne de l'application, c.-à-d. la consultation et la modification de contenu, les applications LFS doivent relaxer la propriété de cohérence des données.

Définition 10 (Cohérence)**.** La cohérence d'un système indique sa capacité à présenter une vue uniforme de son état à chacun de ses utilisateur-rices à un moment donné.

Les applications LFS ne peuvent donc pas reposer sur des méthodes de réplication dites pessimistes, c.-à-d. qui empêchent toutes modifications concurrentes d'une même donnée.

Les applications LFS doivent donc adopter des méthodes de réplication dites optimistes [16]. Ces méthodes autorisent chaque noeud possédant une copie de la donnée de la consulter et de la modifier sans coordination au préalable avec les autres noeuds. L'état

des copies des noeuds peut donc diverger temporairement. Un mécanisme de synchronisation permet ensuite aux noeuds de partager les modifications effectuées et de les intégrer de façon à converger à terme [17], c.-à-d. obtenir à terme de nouveau des états équivalents.

Cependant, il convient de noter que les méthodes de réplication optimistes autorisent la génération en concurrence de modifications provoquant un conflit, e.g. la modification et la suppression d'une même page dans un wiki. Un mécanisme de résolution de conflits est alors nécessaire pour assurer la convergence à terme des noeuds.

De nouveau, le modèle du système des applications LFS limitent les choix possibles concernant les mécanismes de résolution de conflits. Notamment, les applications LFS ne disposent d'aucun contrôle sur le nombre de noeuds qui compose le système, c.-à-d. le nombre d'appareils utilisés par l'ensemble de leurs utilisateur-rices. Le nombre de noeuds peut donc croître de manière non-bornée. La complexité algorithmique des mécanismes de résolution de conflits doit donc être indépendante de ce paramètre, ou alors en être fonction uniquement de manière logarithmique.

De plus, ces noeuds n'offrent aucune garantie sur leur stabilité. Des noeuds peuvent donc rejoindre et participer au système, mais uniquement de manière éphèmère. Ce phénonème est connu sous le nom de *churn* [18]. Ainsi, de part l'absence de garantie sur le nombre de noeuds connectés de manière stable, les applications LFS ne peuvent pas utiliser des mécanismes de résolution de conflits reposant sur une coordination synchrone d'une proportion des noeuds du système, c.-à-d. sur des algorithmes de consensus [19, 20].

Ainsi, pour permettre la conception d'applications LFS, il convient de disposer de mécanismes de résolution de conflits pour l'ensemble des types de données avec une complexité algorithmique efficace par rapport au nombre de noeuds et ne nécessitant pas de coordination synchrone entre une proportion des noeuds du système.

1.2 Questions de recherche et contributions

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

Les Conflict-free Replicated Data Types (CRDTs) [21, 22] sont des types de données répliqués. Ils sont conçus pour permettre à un ensemble de noeuds d'un système de répliquer une donnée et pour leur permettre de la consulter et dee la modifier sans aucune coordination préalable. Dans ce but, les CRDTs incorporent des mécanismes de résolution de conflits automatiques directement au sein leur spécification.

Cependant, ces mécanimes induisent un surcoût, aussi bien en termes de métadonnées et de calculs que de bande-passante. Ces surcoûts sont néanmoins jugés acceptables par la communauté pour une variété de types de données, e.g. le Registre ou l'Ensemble. Cependant, le surcoût des CRDTs pour le type Séquence constitue toujours une problématique de recherche.

En effet, la particuliarité des CRDTs pour le type Séquence est que leur surcoût croît de manière monotone au cours de la durée de vie de la donnée, c.-à-d. au fur et à mesure des modifications effectuées. Le surcoût introduit par les CRDTs pour ce type de données se révèle donc handicapant dans le contexte de collaborations sur de longues durées ou à

large échelle.

De manière plus précise, le surcoût des CRDTs pour le type Séquence provient de la croissance des métadonnées utilisées par leur mécanisme de résolution de conflits automatique. Ces métadonnées correspondent à des identifiants qui sont associés aux éléments de la Séquence. Ces identifiants permettent de résoudre les conflits, e.g. en précisant quel est l'élement à supprimer ou en spécifiant la position d'un nouvel élément à insérer par rapport aux autres.

Plusieurs approches ont été proposées pour réduire le coût induit par ces identifiants. Notamment, [23, 24] proposent un mécanisme de ré-assignation des identifiants pour réduire leur coût a posteriori. Ce mécanisme génère toutefois des conflits en cas de modifications concurrentes de la séquence, c.-à-d. l'insertion ou la suppression d'un élément. Les auteurs résolvent ce problème en proposant un mécanisme de transformation des modifications concurrentes par rapport à l'effet du mécanisme de ré-assignation des identifiants.

Cependant, l'exécution en concurrence du mécanisme de ré-assignation des identifiants par plusieurs noeuds provoque elle-même un conflit. Pour éviter ce dernier type de conflit, les auteurs choisissent de subordonner à un algorithme de consensus l'exécution du mécanisme de ré-assignation des identifiants. Ainsi, le mécanisme de ré-assignation des identifiants ne peut être déclenché en concurrence par plusieurs noeuds du systèmes.

Comme nous l'avons évoqué précédemment, reposer sur un algorithme de consensus qui requiert une coordination synchrone entre une proportion de noeuds du système est une contrainte incompatible avec les systèmes P2P à large échelle sujets au churn. Notre problématique de recherche est donc la suivante : *pouvons-nous proposer un mécanisme sans coordination synchrone de réduction du surcoût des CRDTs pour Séquence, c.-à-d. adapté aux applications LFS ?*

Pour répondre à cette problématiquee, nous proposons RenamableLogootSplit, un nouveau CRDT pour le type Séquence. Ce CRDT intègre un mécanisme de ré-assignation des identifiants, dit de renommage, directement au sein de sa spécification. Nous associons au mécanisme de renommage un mécanisme de résolution de conflits automatique additionnel pour gérer ses exécutions concurrentes. Ainsi, nous proposons un CRDT pour le type Séquence dont le surcoût est périodiquement réduit par le biais d'un mécanisme n'introduisant aucune contrainte de coordination synchrone entre les noeuds du système.

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

Les systèmes collaboratifs permettent à plusieurs utilisateur-rices de collaborer pour la réalisation d'une tâche. Les systèmes collaboratifs actuels adoptent principalement une architecture décentralisée, c.-à-d. un ensemble de serveurs avec lesquels les utilisateur-rices interagissent pour réaliser leur tâche, e.g. Google Docs [25]. Par rapport à une architecture centralisée, cette architecture leur permet d'améliorer leur disponibilité et tolérance aux pannes, notamment grâce aux méthodes de réplication de données. Cette architecture à base de serveurs facilite aussi la collaboration, les serveurs permettant d'intégrer les modifications effectuées par les utilisateur-rices, de stocker les données, d'assurer la communication entre les utilisateur-rices ou encore de les authentifier.

De part le rôle qui leur incombe, ces serveurs occupent une place primordiale dans ces systèmes. Il en découle plusieurs problématiques :

(i) Ces serveurs manipulent et hébergent les données faisant l'objet de collaborations. Ces systèmes ont donc connaissance des données manipulées et de l'identité des auteur-rices des modifications. Les systèmes collaboratifs décentralisés demandent donc à leurs utilisateur-rices d'abandonner la souveraineté et la confidentialité de leur travail.

(ii) Ces serveurs sont gérés par des autorités centrales, e.g. Google. Les systèmes collaboratifs devenant non-fonctionnels en cas d'arrêt de leurs serveurs, les utilisateur-rices de ces systèmes dépendent de ces autorités centrales. Ainsi, de part leur pouvoir de vie et de mort sur les services qu'elles proposent, les autorités centrales représentent une menace pour la pérennité de ces systèmes, e.g. [26].

Pour répondre à ces problématiques, c.-à-d. confidentialité et souveraineté des données, dépendance envers des tiers, pérennité des systèmes, un nouveau paradigme de conception d'applications propose de concevoir des applications LFS, c.-à-d. des applications mettant les utilisateur-rices et leurs appareils au coeur du système. Dans ce cadre d'applications P2P, les serveurs sont relégués seulement à un rôle de support à la collaboration.

Dans le cadre de ses travaux, notre équipe de recherche étudie notamment la conception d'applications respectant ce paradigme. Ce changement de modèle, d'une architecture décentralisée appartenant à des autorités centrales à une architecture P2P sans autorités centrales, introduit un ensemble de problématiques de domaines variés, e.g.

(i) Comment permettre aux utilisateur-rices de collaborer en l'absence d'autorités centrales pour résoudre les conflits de modifications ?

(ii) Comment authentifier les utilisateur-rices en l'absence d'autorités centrales ?

(iii) Comment structurer le réseau de manière efficace, c.-à-d. en limitant le nombre de connexions par pair ?

Cet ensemble de questions peut être résumé en la problématique suivante : *pouvons-nous concevoir une application collaborative P2P à large échelle, sûre et sans autorités centrales ?*

Pour étudier cette problématique, l'équipe Coast développe l'application Multi User Text Editor (MUTE) [3] [27]. Il s'agit d'un éditeur de texte web collaboratif P2P temps réel chiffré de bout en bout. Ce projet nous permet de présenter les travaux de recherche de l'équipe portant sur les mécanismes de résolutions de conflits automatiques pour le type Séquence [28, 29, 30] et les mécanismes d'authentification des pairs dans les systèmes sans autorités centrales [31, 32]. Puis, ce projet nous donne l'opportunité d'étudier la littérature des nombreux domaines de recherche nécessaires à la conception d'un tel système, c.-à-d. le domaine des protocoles d'appartenance aux groupes [33, 34], des topologies réseaux P2P [35] ou encore des protocoles d'établissement de clés de chiffrement de groupe [36]. Ce projet nous permet ainsi de valoriser nos travaux et d'identifier de nouvelles perspectives de recherche. Finalement, il résulte de ce projet le Proof of Concept (PoC) le plus complet d'applications LFS, à notre connaissance. *Matthieu: TODO : Vérifier du côté des applis de IPFS*

3. Disponible à l'adresse : `https://mutehost.loria.fr`

1.3 Plan du manuscrit

Ce manuscrit de thèse est organisé de la manière suivante :

Dans le chapitre 2, nous introduisons le modèle du système que nous considérons, c.-à-d. les systèmes P2P à large échelle sujets au churn et sans autorités centrales. Puis nous présentons dans ce chapitre l'état de l'art des mécanismes de résolution de conflits automatiques utilisés dans les systèmes adoptant le paradigme de la réplication optimiste. À partir de cet état de l'art, nous identifions et motivons notre problématique de recherche, c.-à-d. l'absence de mécanisme adapté aux systèmes P2P à large échelle sujets au churn permettant de réduire le surcoût induit par les mécanismes de résolution de conflits automatiques pour le type Séquence.

Dans le **??**, nous présentons notre approche pour présenter un tel mécanisme, c.-à-d. un mécanisme de résolution de conflits automatiques pour le type Séquence auquel nous associons un mécanisme de GC de son surcoût ne nécessitant pas de coordination synchrone entre les noeuds du système. Nous détaillons le fonctionnement de notre approche, sa validation par le biais d'une évaluation empirique puis comparons notre approche par rapport aux approches existantes Finalement, nous concluons la présentation de notre approche en identifiant et en détaillant plusieurs de ses limites.

Dans le chapitre 3, nous présentons MUTE, l'éditeur de texte collaboratif temps réel P2P chiffré de bout en bout que notre équipe de recherche développe dans le cadre de ses travaux de recherche. Nous présentons les différentes couches logicielles formant un pair et les services tiers avec lesquels les pairs interagissent, et détaillons nos travaux dans le cadre de ce projet, c.-à-d. l'intégration de notre mécanisme de résolution de conflits automatiques pour le type Séquence et le développement de la couche de livraison des messages associée. Pour chaque couche logicielle, nous identifions ses limites et présentons de potentielles pistes d'améliorations.

Finalement, nous récapitulons dans le chapitre 4 les contributions réalisées dans le cadre de cette thèse. Puis nous clotûrons ce manuscrit en introduisant plusieurs des pistes de recherches que nous souhaiterons explorer dans le cadre de nos travaux futurs.

1.4 Publications

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

(i) Dans [37], nous motivons le problème identifié et présentons l'idée de notre approche pour y répondre.

(ii) Dans [38], nous détaillons une première partie de notre approche et présentons notre protocole d'évaluation expérimentale ainsi que ses premiers résultats.

(iii) Dans [30], nous détaillons notre proposition dans son entièreté. Nous accompagnons cette proposition d'une évaluation expérimentale poussée. Finalement, nous complétons notre travail d'une discussion identifiant plusieurs de ses limites et présentant des pistes de travail possibles pour y répondre.

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

Efficient renaming in CRDTs [37]

Auteur Matthieu Nicolas

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

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

Efficient Renaming in Sequence CRDTs [38]

Auteurs Matthieu Nicolas, Gérald Oster, Olivier Perrin

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

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

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

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

Efficient Renaming in Sequence CRDTs [30]

Auteurs Matthieu Nicolas, Gérald Oster, Olivier Perrin

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

Abstract To achieve high availability, large-scale distributed systems have to replicate data and to minimise coordination between nodes. For these purposes, literature and industry increasingly adopt Conflict-free Replicated Data Types (CRDTs) to design such systems. CRDTs are new specifications of existing data types, e.g. Set or Sequence. While CRDTs have the same behaviour as previous specifications in sequential executions, they actually shine in distributed settings as they natively support concurrent updates. To this end, CRDTs embed in their specification conflict resolution mechanisms. These mechanisms usually rely on identifiers attached to elements of the data structure to resolve conflicts in a deterministic and coordination-free manner. Identifiers have to comply with several constraints, such as being unique or belonging to a dense total order. These constraints may hinder the identifier size from being bounded. Identifiers hence tend to grow as the system progresses, which increases the overhead of CRDTs over time and leads to performance issues. To address this issue, we propose a novel Sequence CRDT which embeds a renaming mechanism. It enables nodes to reassign shorter identifiers to elements in an uncoordinated manner. Experimental results demonstrate that this mechanism decreases the overhead of the replicated data structure and eventually minimises it.

Chapitre 2

État de l'art

Sommaire

2.1 Modèle du système 12
2.2 Types de données répliquées sans conflits 13
2.2.1 Sémantiques en cas de conflits 17
2.2.2 Modèles de synchronisation 21
2.3 Séquences répliquées sans conflits 30
2.3.1 Approche à pierres tombales 33
2.3.2 Approche à identifiants densément ordonnés 41
2.3.3 Synthèse 49
2.4 LogootSplit 52
2.4.1 Identifiants 52
2.4.2 Aggrégation dynamique d'élements en blocs 53
2.4.3 Modèle de données 55
2.4.4 Modèle de livraison 57
2.4.5 Limites 60
2.5 Mitigation du surcoût des séquences répliquées sans conflits 62
2.5.1 Mécanisme de GC des pierres tombales 62
2.5.2 Ré-équilibrage de l'arbre des identifiants de position 63
2.5.3 Réduction de la croissance des identifiants de position 63
2.5.4 Synthèse 64
2.6 Synthèse 65
2.7 Proposition 65

Dans ce chapitre, nous définissons le modèle du système que nous considérons (section 2.1). Puis nous présentons le fonctionnement de LogootSplit, le Conflict-free Replicated Data Type (CRDT) pour le type Séquence qui sert de base pour nos travaux (section 2.4). Ensuite, nous présentons les approches proposées pour réduire le surcoût des CRDTs pour le type Séquence et identifions leurs limites (sous-section 2.5.2 et sous-section 2.5.3). Finalement, nous introduisons l'approche que nous proposons (section 2.7) pour répondre à notre première problématique de recherche (cf. sous-section 1.2.1, page 5), que nous présentons en détails par la suite dans le **??**.

Néanmoins, afin d'offrir une vision plus globale de notre domaine de recherche, nous complétons notre état de l'art de plusieurs points. Dans la section 2.2, nous rappelons la notion de CRDTs, c.-à-d. de types de données répliquées sans conflits. Ce rappel se compose d'une section présentant la notion de sémantique pour un mécanisme de résolution de conflits automatiques (sous-section 2.2.1) et d'une section présentant les différents modèles de synchronisation pour CRDTs définis dans la littérature, c.-à-d. la synchronisation par états, la synchronisation par opérations et la synchronisation par différences d'états (sous-section 2.2.2). À notre connaissance, nous présentons une des études les plus complètes comparant ces modèles de synchronisation en guise de synthèse de cette même section.

De manière similaire, nous rappelons les différents CRDTs pour le type Séquence définis dans la littérature dans la section 2.3. Ce rappel prend la forme d'un historique des CRDTs pour le type Séquence, catégorisés en fonction de l'approche sur laquelle se base leur mécanisme de résolution de conflits, c.-à-d. l'approche à pierres tombales ou l'approche à identifiants densément ordonnés. De nouveau, ce rappel aboutit à notre connaissance à l'une des études les plus précises comparant ces deux approches (sous-section 2.3.3).

2.1 Modèle du système

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

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

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

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

Définition 11 (Convergence à terme)**.** La convergence à terme est une propriété de sûreté indiquant que l'ensemble des noeuds du système ayant intégrés le même ensemble

de modifications obtiendront des états équivalents [4].

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

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

Définition 12 (Livraison à terme)**.** La livraison à terme est un modèle de livraison garantissant que l'ensemble des messages du système seront livrés à l'ensemble des noeuds du système à terme.

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

2.2 Types de données répliquées sans conflits

Afin d'offrir une haute disponibilité à leurs clients et afin d'accroître leur tolérance aux pannes [39], les systèmes distribués peuvent adopter le paradigme de la réplication optimiste [16]. Ce paradigme consiste à ce que chaque noeud composant le système possède une copie de la donnée répliquée. Chaque noeud possède le droit de la consulter et de la modifier, sans coordination préalable avec les autres noeuds. Les noeuds peuvent alors temporairement diverger, c.-à-d. posséder des états différents. Un mécanisme de synchronisation leur permet ensuite de partager leurs modifications respectives et d'obtenir de nouveau des états équivalent, c.-à-d. de converger à terme [17].

Pour permettre aux noeuds de converger, les protocoles de réplication optimiste ordonnent généralement les évènements se produisant dans le système distribué. Pour les ordonner, la littérature repose généralement sur la relation de causalité entre les évènements, qui est définie par la relation *happens-before* [40]. Nous l'adaptons ci-dessous à notre contexte, en ne considérant que les modifications [5] effectuées et celles intégrées :

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

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

Définition 13 (Relation *happens-before*)**.** La relation *happens-before* indique qu'une modification m_1 a eu lieu avant une modification m_2, notée $m_1 \rightarrow m_2$, si et seulement si une des conditions suivantes est satisfaite :

(i) m_1 a été effectuée avant m_2 sur le même noeud.

(ii) m_1 a été intégrée par le noeud auteur de m_2 avant qu'il n'effectue m_2.

(iii) Il existe une modification m telle que $m_1 \rightarrow m \wedge m \rightarrow m_2$.

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

Définition 14 (Concurrence)**.** Deux modifications m_1 et m_2 sont concurrentes, noté $m_1 \parallel m_2$, si et seulement si $m_1 \nrightarrow m_2 \wedge m_1 \nrightarrow m_2$.

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

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

payload
$S \in Set\langle E \rangle$

constructor

emp	:		$\longrightarrow S$

mutators

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

queries

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

FIGURE 2.1 – Spécification algébrique du type abstrait usuel Ensemble

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

La spécification définit deux modifications sur l'ensemble :

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

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

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

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

Elle définit aussi deux observateurs :

(i) $len(s)$, qui permet de récupérer le nombre d'éléments présents dans un ensemble s.

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

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

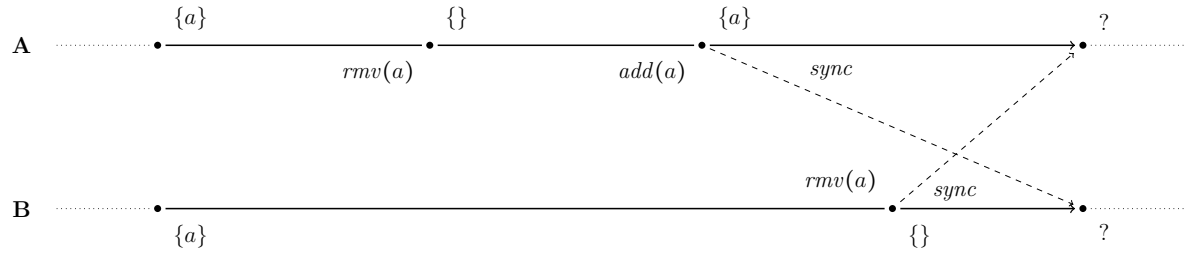


FIGURE 2.2 – Modifications concurrentes d'un Ensemble répliqué provoquant un conflit suite à l'ajout et la suppression d'un même élément

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

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

Les Conflict-free Replicated Data Types (CRDTs) [22, 41, 42] répondent à ce besoin.

Définition 15 (Conflict-free Replicated Data Type)**.** Les CRDTs sont de nouvelles spécifications des types de données existants, e.g. l'Ensemble ou la Séquence. Ces nouvelles spécifications sont conçues pour être utilisées dans des systèmes distribués adoptant la réplication optimiste. Ainsi, elles offrent les deux propriétés suivantes :

(i) Les CRDTs peuvent être modifiés sans coordination avec les autres noeuds.

(ii) Les CRDTs garantissent la *convergence forte* [22].

Définition 16 (Convergence forte)**.** La convergence forte est une propriété de sûreté indiquant que l'ensemble des noeuds d'un système ayant intégrés le même ensemble de modifications obtiendront des états équivalents, sans échange de message supplémentaire.

Pour offrir la propriété de *convergence forte*, la spécification des CRDTs reposent sur la théorie des treillis [43] :

Définition 17 (Spécification des CRDTs)**.** Les CRDTs sont spécifiés de la manière suivante :

(i) Les différents états possibles d'un CRDT forment un sup-demi-treillis, possédant une relation d'ordre partiel $\leq$.

(ii) Les modifications génèrent par inflation un nouvel état supérieur ou égal à l'état original d'après $\leq$.

(iii) Il existe une fonction de fusion qui, pour toute paire d'états, génère l'état minimal supérieur d'après $\leq$ aux deux états fusionnés. Nous parlons alors de borne supérieure ou de Least Upper Bound (LUB) pour catégoriser l'état résultant de cette fusion.

Malgré leur spécification différente, les CRDTs partagent la même sémantique, c.-à-d. le même comportement, et la même interface que les types séquentiels [6] correspondants du point de vue des utilisateur-rices. Ainsi, les CRDTs partagent le comportement des types séquentiels dans le cadre d'exécutions séquentielles. Cependant, ils définissent aussi une sémantique additionnelle pour chaque type de conflit ne pouvant se produire que dans le cadre d'une exécution distribuée.

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

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

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

Matthieu: TODO : Faire le lien avec les travaux de Burckhardt [44] et les MRDTs [45]

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

2.2.1 Sémantiques en cas de conflits

Plusieurs sémantiques peuvent être proposées pour résoudre les conflits. Certaines de ces sémantiques ont comme avantage d'être générique, c.-à-d. applicable à l'ensemble des types de données. En contrepartie, elles souffrent de cette même généricité, en ne permettant que des comportements simples en cas de conflits.

À l'inverse, la majorité des sémantiques proposées dans la littérature sont spécifiques à un type de données. Elles visent ainsi à prendre plus finement en compte l'intention des modifications pour proposer des comportements plus précis.

Dans la suite de cette section, nous présentons ces sémantiques génériques ainsi que celles spécifiques à l'Ensemble et, à titre d'exemple, les illustrons à l'aide du scénario présenté dans la Figure 2.2.

Sémantique *Last-Writer-Wins*

Une manière simple pour résoudre un conflit consiste à trancher de manière arbitraire et de sélectionner une modification parmi l'ensemble des modifications en conflit. Pour faire cela de manière déterministe, une approche est de reproduire et d'utiliser l'ordre total sur les modifications qui serait instauré par une horloge globale pour choisir la modification à prioritiser.

Cette approche, présentée dans [46], correspond à la sémantique nommée *Last-Writer-Wins* (LWW). De par son fonctionnement, cette sémantique est générique et est donc utilisée par une variété de CRDTs pour des types différents. La Figure 2.3 illustre son application à l'Ensemble pour résoudre le conflit de la Figure 2.2.

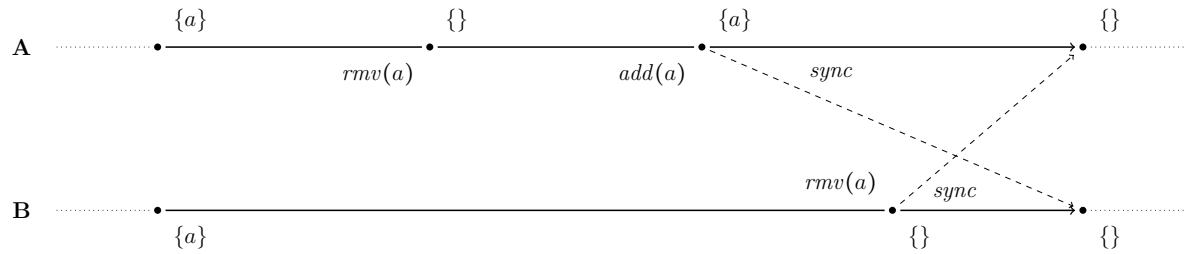


FIGURE 2.3 – Résolution du conflit en utilisant la sémantique LWW

Comme indiqué précédemment, le scénario illustré dans la Figure 2.3 présente un conflit entre les modifications concurrentes $add(a)$ et $rmv(a)$ générées de manière concurrente respectivement par les noeuds A et B. Pour le résoudre, la sémantique LWW associe à chaque modification une estampille. L'ordre créé entre les modifications par ces dernières permet de déterminer quelle modification désigner comme prioritaire. Ici, nous considérons que $add(a)$ a eu lieu plus tôt que $rmv(a)$. La sémantique LWW désigne donc $rmv(a)$ comme prioritaire et ignore $add(a)$. L'état obtenu à l'issue de cet exemple par chaque noeud est donc $\{\}$.

Il est à noter que si la modification $rmv(a)$ du noeud B avait eu lieu plus tôt dans notre exemple, l'état final obtenu aurait été $\{a\}$. Ainsi, des exécutions reproduisant le même ensemble de modifications produiront des résultats différents en fonction de l'ordre créé

par les estampilles associées à chaque modification. Ces estampilles étant des métadonnées du mécanisme de résolution de conflits, elles sont dissimulées aux utilisateur-rices. Le comportement de cette sémantique peut donc être perçu comme aléatoire et s'avérer perturbant pour les utilisateur-rices.

La sémantique LWW repose sur l'horloge de chaque noeud pour attribuer une estampille à chacune de leurs modifications. Les horloges physiques étant sujettes à des imprécisions et notamment des décalages, utiliser les estampilles qu'elles fournissent peut provoquer des anomalies vis-à-vis de la relation *happens-before*. Les systèmes distribués préfèrent donc généralement utiliser des horloges logiques [40]. *Matthieu: TODO : Ajouter refs des horloges logiques plus intelligentes (Interval Tree Clock, Hybrid Clock...)*

Sémantique *Multi-Value*

Une seconde sémantique générique[7] est la sémantique *Multi-Value* (MV). Cette approche propose de gérer les conflits de la manière suivante : plutôt que de prioritiser une modification par rapport aux autres modifications concurrentes, la sémantique MV maintient l'ensemble des états résultant possibles. Nous présentons son application à l'Ensemble dans la Figure 2.4.

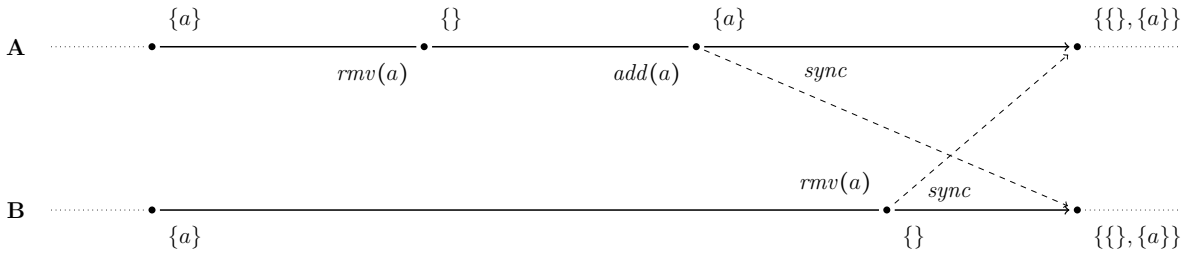


FIGURE 2.4 – Résolution du conflit en utilisant la sémantique MV

La Figure 2.4 présente la gestion du conflit entre les modifications concurrentes $add(a)$ et $rmv(a)$ par la sémantique MV. Devant ces modifications contraires, chaque noeud calcule chaque état possible, c.-à-d. un état sans l'élément a, $\{\}$, et un état avec ce dernier, $\{a\}$. Le CRDT maintient alors l'ensemble de ces états en parallèle. L'état obtenu est donc $\{\{\}, \{a\}\}$.

Ainsi, la sémantique MV expose les conflits aux utilisateur-rices lors de leur prochaine consultation de l'état du CRDT. Les utilisateur-rices peuvent alors prendre connaissance des intentions de chacun-e et résoudre le conflit manuellement. Dans la Figure 2.4, résoudre le conflit revient à re-effectuer une modification $add(a)$ ou $rmv(a)$ selon l'état choisi. Ainsi, si plusieurs personnes résolvent en concurrence le conflit de manière contraire, la sémantique MV exposera de nouveau les différents états proposés sous la forme d'un conflit.

Il est intéressant de noter que cette sémantique mène à un changement du domaine du CRDT considéré : en cas de conflit, la valeur retournée par le CRDT correspond à un Ensemble de valeurs du type initialement considéré. E.g. si nous considérons que le

7. Bien qu'uniquement associée au type *Registre* dans le domaine des CRDTs généralement.

type correspondant au CRDT dans la Figure 2.4 est le type $Set\langle V\rangle$, nous observons que la valeur finale obtenue a pour type $Set\langle Set\langle V\rangle\rangle$. Il s'agit à notre connaissance de la seule sémantique opérant ce changement.

Sémantiques *Add-Wins* et *Remove-Wins*

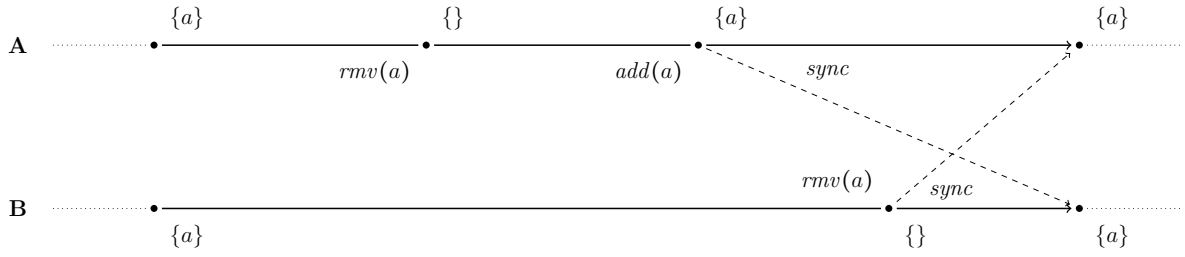
Comme évoqué précédemment, d'autres sémantiques sont spécifiques au type de données concerné. Ainsi, nous abordons à présent des sémantiques spécifiques au type de l'Ensemble.

Dans le cadre de l'Ensemble, un conflit est provoqué lorsque des modifications *add* et *rmv* d'un même élément sont effectuées en concurrence. Ainsi, deux approches peuvent être proposées pour résoudre le conflit :

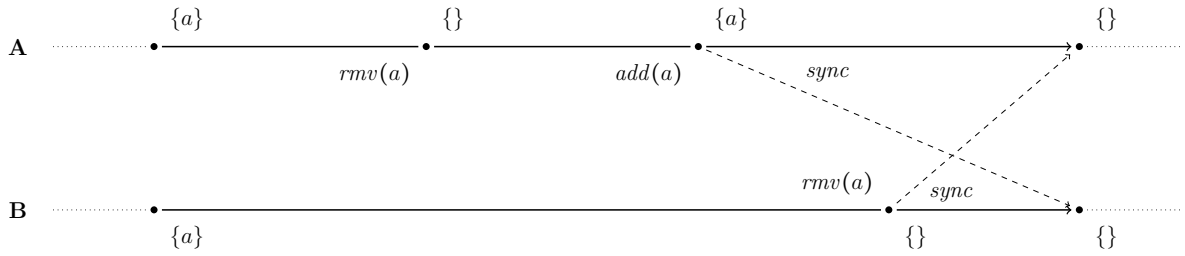
(i) Une sémantique où la modification *add* d'une élément prend la précédence sur les modifications concurrentes *rmv* du même élément, nommée *Add-Wins* (AW). L'élément est alors présent dans l'état obtenu à l'issue de la résolution du conflit.

(ii) Une sémantique où la modification *rmv* d'une élément prend la précédence sur les opérations concurrentes *add* du même élément, nommée *Remove-Wins* (RW). L'élément est alors absent de l'état obtenu à l'issue de la résolution du conflit.

La Figure 2.5 illustre l'application de chacune de ces sémantiques sur notre exemple.



(a) Application de la sémantique AW



(b) Application de la sémantique RW

FIGURE 2.5 – Résolution du conflit en utilisant soit la sémantique AW, soit la sémantique RW

Sémantique *Causal-Length*

Une nouvelle sémantique pour l'Ensemble fut proposée [47] récemment. Cette sémantique se base sur les observations suivantes :

(i) *add* et *rmv* d'un élément prennent place à tour de rôle, chaque modification invalidant la précédente.

(ii) *add* (resp. *rmv*) concurrents d'un même élément représentent la même intention. Prendre en compte une de ces modifications concurrentes revient à prendre en compte leur ensemble.

À partir de ces observations, Yu et al. [47] proposent de déterminer pour chaque élément la chaîne d'ajouts et retraits la plus longue. C'est cette chaîne, et précisément son dernier maillon, qui indique si l'élement est présent ou non dans l'ensemble final. La Figure 2.6 illustre son fonctionnement.

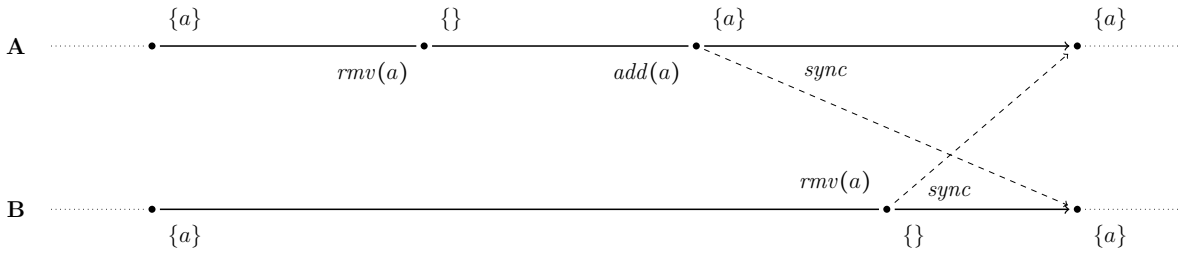


Figure 2.6 – Résolution du conflit en utilisant la sémantique CL

Dans notre exemple, la modification $rmv(a)$ effectuée par B est en concurrence avec une modification identique effectuée par A. La sémantique CL définit que ces deux modifications partagent la même intention. Ainsi, A ayant déjà appliqué sa propre modification préalablement, il ne prend pas en compte *de nouveau* cette modification lorsqu'il la reçoit de B. Son état reste donc inchangé.

À l'inverse, la modification $add(a)$ effectuée par A fait suite à sa modification $rmv(a)$. La sémantique CL définit alors qu'elle fait suite à toute autre modification $rmv(a)$ concurrente. Ainsi, B intègre cette modification lorsqu'il la reçoit de A. Son état évolue donc pour devenir $\{a\}$.

Synthèse

Dans cette section, nous avons mis en lumière l'existence de solutions différentes pour résoudre un même conflit. Chacune de ces solutions correspond à une sémantique spécifique de résolution de conflits. Ainsi, pour un même type de données, différents CRDTs peuvent être spécifiés. Chacun de ces CRDTs est spécifié par la combinaison de sémantiques qu'il adopte, chaque sémantique servant à résoudre un des types de conflits du type de données.

Il est à noter qu'aucune sémantique n'est intrinsèquement meilleure et préférable aux autres. Il revient aux concepteur-rices d'applications de choisir les CRDTs adaptés en fonction des besoins et des comportements attendus en cas de conflits.

Par exemple, pour une application collaborative de listes de courses, l'utilisation d'un MV-Registre pour représenter le contenu de la liste se justifie : cette sémantique permet d'exposer les modifications concurrentes aux utilisateur-rices. Ainsi, les personnes peuvent détecter et résoudre les conflits provoquées par ces éditions concurrentes, e.g. l'ajout

de l'élément *lait* à la liste, pour cuisiner des crêpes, tandis que les *oeufs* nécessaires à ces mêmes crêpes sont retirés. En parallèle, cette même application peut utiliser un LWW-Registre pour représenter et indiquer aux utilisateur-rices la date de la dernière modification effectuée.

2.2.2 Modèles de synchronisation

Dans le modèle de réplication optimiste, les noeuds divergent momentanément lorsqu'ils effectuent des modifications locales. Pour ensuite converger vers des états équivalents, les noeuds doivent propager et intégrer l'ensemble des modifications. La Figure 2.7 illustre ce point.

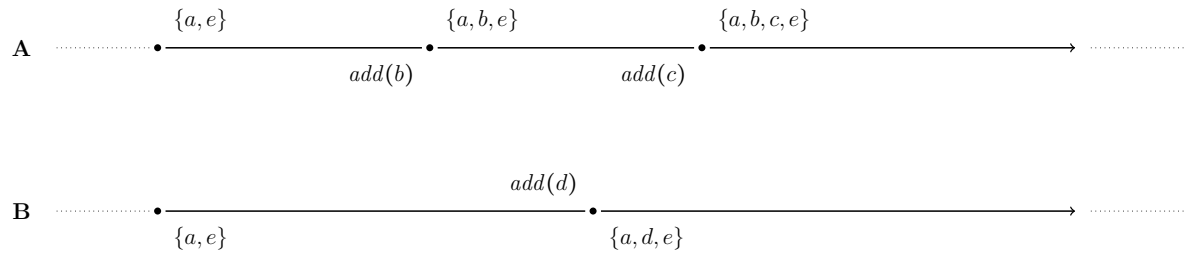


FIGURE 2.7 – Modifications en concurrence d'un Ensemble répliqué par les noeuds A et B

Dans cet exemple, deux noeuds A et B partagent et éditent un même Ensemble à l'aide d'un CRDT. Les deux noeuds possèdent le même état initial : $\{a, e\}$.

Le noeud A effectue les modifications $add(b)$ puis $add(c)$. Il obtient ainsi l'état $\{a, b, c, e\}$. De son côté, le noeud B effectue la modification suivante : $add(d)$. Son état devient donc $\{a, d, e\}$. Ainsi, les noeuds doivent encore s'échanger leur modifications pour converger vers l'état souhaité [8], c.-à-d. $\{a, b, c, d, e\}$.

Dans le cadre des CRDTs, le choix de la méthode pour synchroniser les noeuds n'est pas anodin. En effet, ce choix impacte la spécification même du CRDT et ses prérequis.

Initialement, deux approches ont été proposées : une méthode de synchronisation par états [22, 48] et une méthode de synchronisation par opérations [22, 48, 49, 50]. Une troisième approche, nommée synchronisation par différence d'états [51, 52], fut spécifiée par la suite. Le but de cette dernière est d'allier le meilleur des deux approches précédentes.

Dans la suite de cette section, nous présentons ces approches ainsi que leurs caractéristiques respectives. Pour les illustrer, nous complétons l'exemple décrit ici. Cependant, nous nous focalisons uniquement sur les messages envoyés par les noeuds et n'évoquons seulement les métadonnées introduites par chaque modèle de synchronisation, par soucis de clarté et de simplicité.

8. Le scénario ne comportant uniquement des modifications *add*, aucun conflit n'est produit malgré la concurrence des modifications.

Synchronisation par états

L'approche de la synchronisation par états propose que les noeuds diffusent leurs modifications en transmettant leur état. Les CRDTs adoptant cette approche doivent définir une fonction `merge`. Cette fonction correspond à la fonction de fusion mentionnée précédemment (cf. Définition 17, page 16) : elle prend en paramètres une paire d'états et génère en retour leur LUB, c.-à-d. l'état correspondant à la borne supérieure des deux états en paramètres. Cette fonction doit être associative, commutative et idempotente.

Ainsi, lorsqu'un noeud reçoit l'état d'un autre noeud, il fusionne ce dernier avec son état courant à l'aide de la fonction `merge`. Il obtient alors un nouvel état intégrant l'ensemble des modifications ayant été effectuées sur les deux états.

La nature croissante des états des CRDTs couplée aux propriétés d'associativité, de commutativité et d'idempotence de la fonction `merge` permettent de reposer sur la couche de livraison sans lui imposer de contraintes fortes : les messages peuvent être perdus, réordonnés ou même dupliqués. Les noeuds convergeront tant que la couche de livraison garantit que les noeuds seront capables de transmettre leur état aux autres à terme. Il s'agit là de la principale force des CRDTs synchronisés par états.

Néanmoins, la définition de la fonction `merge` offrant ces propriétés peut s'avérer complexe et a des répercussions sur la spécification même du CRDT. Notamment, les états doivent conserver une trace de l'existence des éléments et de leur suppression afin d'éviter qu'une fusion d'états ne les fassent ressurgirent. Ainsi, les CRDTs synchronisés par états utilisent régulièrement des pierres tombales.

Définition 18 (Pierre tombale)**.** Une pierre tombale est un marqueur de la présence passée d'un élément.

Dans le contexte des CRDTs, un identifiant est généralement associé à chaque élément. Dans ce contexte, l'utilisation de pierres tombales correspond au comportement suivant : la suppression d'un élément peut supprimer de manière effective ce dernier, mais doit cependant conserver son identifiant dans la structure de données.

En plus de l'utilisation de pierres tombales, la taille de l'état peut croître de manière non-bornée dans le cas de certains types de donnés, e.g. l'Ensemble ou la Séquence. Ainsi, ces structures peuvent atteindre à terme des tailles conséquentes. Dans de tels cas, diffuser l'état complet à chaque modification induirait alors un coût rédhibitoire. L'approche de la synchronisation par états s'avère donc inadaptée aux systèmes nécessitant une diffusion et intégration instantanée des modifications, c.-à-d. les systèmes temps réel. Ainsi, les systèmes utilisant des CRDTs synchronisés par états reposent généralement sur une synchronisation périodique des noeuds, c.-à-d. chaque noeud diffuse périodiquement son état.

Nous illustrons le fonctionnement de cette approche avec la Figure 2.8. Dans cet exemple, après que les noeuds aient effectués leurs modifications respectives, le mécanisme de synchronisation périodique de chaque noeud se déclenche. Le noeud A (resp. B) diffuse alors son état $\{a, b, c, e\}$ (resp. $\{a, d, e\}$) à B (resp. A).

À la réception de l'état, chaque noeud utilise la fonction `merge` pour intégrer les modifications de l'état reçu dans son propre état. Dans le cadre de l'Ensemble répliqué, cette fonction consiste généralement à faire l'union des états, en prenant en compte l'estampille

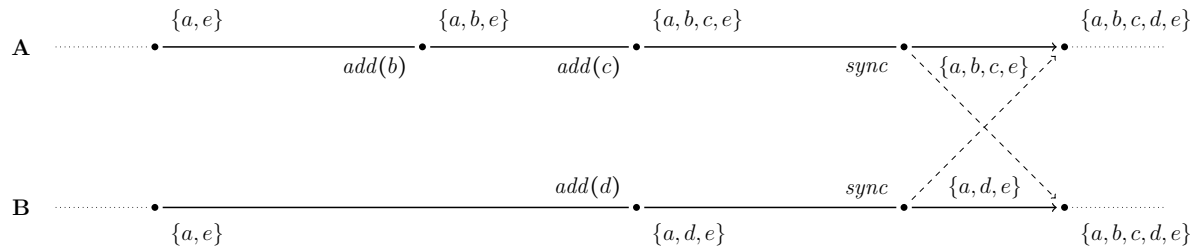


FIGURE 2.8 – Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par états

et le statut (présent ou non) associé à chaque élément. Ainsi la fusion de leur état respectif, $\{a, b, c, e\} \cup \{a, d, e\}$, permet aux noeuds de converger à l'état souhaité : $\{a, b, c, d, e\}$.

Avant de conclure, il est intéressant de noter que les CRDTs adoptant ce modèle de synchronisation respectent de manière intrinsèque le modèle de cohérence causale [53].

Définition 19 (Modèle de cohérence causale)**.** Le modèle de cohérence causale définit que, pour toute paire de modifications m_1 et m_2 d'une exécution, si $m_1 \rightarrow m_2$, alors l'ensemble des noeuds doit intégrer la modification m_1 avant d'intégrer la modification m_2.

En effet, ce modèle de synchronisation assure l'intégration soit de toutes les modifications connues d'un noeud, soit d'aucune. Par exemple, dans la Figure 2.8, le noeud B ne peut pas recevoir et intégrer l'élément c sans l'élement b. Ainsi, ce modèle permet naturellement d'éviter ce qui pourrait être interprétées comme des anomalies par les utilisateur-rices.

Synchronisation par opérations

L'approche de la synchronisation par opérations propose quant à elle que les noeuds diffusent leurs modifications sous la forme d'opérations. Pour chaque modification possible, les CRDTs synchronisés par opérations doivent définir deux fonctions : `prepare` et `effect` [50].

La fonction `prepare` a pour but de générer une opération correspondant à la modification effectuée, et commutative avec les potentielles opérations concurrentes. Elle prend en paramètres la modification ainsi que ses paramètres, et l'état courant du noeud. Cette fonction n'a pas d'effet de bord, c.-à-d. ne modifie pas l'état courant, et génère en retour l'opération à diffuser à l'ensemble des noeuds.

Une opération est un message. Son rôle est d'encoder la modification sous la forme d'un ou plusieurs éléments irréductibles du sup-demi-treillis.

Définition 20 (Élément irréductible)**.** Un élément irréductible d'un sup-demi-treillis est un élément atomique de ce dernier. Il ne peut être obtenu par la fusion d'autres états.

Il est à noter que dans le cas des CRDTs purs synchronisés par opérations [50], les modifications estampillées avec leur information de causalité correspondent à des éléments

irréductibles, c.-à-d. à des opérations. La fonction `prepare` peut donc être omise pour cette sous-catégorie de CRDTs synchronisés par opérations.

La fonction `effect` permet quant à elle d'intégrer les effets d'une opération générée ou reçue. Elle prend en paramètre l'état courant et l'opération, et retourne un nouvel état. Ce nouvel état correspond à la LUB entre l'état courant et le ou les éléments irréductibles encodés par l'opération.

La diffusion des modifications par le biais d'opérations présentent plusieurs avantages. Tout d'abord, la taille des opérations est généralement fixe et inférieure à la taille de l'état complet du CRDT, puisque les opérations servent à encoder un de ses éléments irréductibles. Ensuite, l'expressivité des opérations permet de proposer plus simplement des algorithmes efficaces pour leur intégration par rapport aux modifications équivalentes dans les CRDTs synchronisés par états. Par exemple, la suppression d'un élément dans un Ensemble se traduit en une opération de manière presque littérale, tandis que pour les CRDTs synchronisés par états, c'est l'absence de l'élément dans l'état qui va rendre compte de la suppression effectuée. Ces avantages rendent possible la diffusion et l'intégration une à une des modifications et rendent ainsi plus adaptés les CRDTs synchronisés par opérations pour construire des systèmes temps réels.

Il est à noter que la seule contrainte imposée aux CRDTs synchronisés par opérations est que leurs opérations concurrentes soient commutatives [22]. Ainsi, il n'existe aucune contrainte sur la commutativité des opérations liées causalement. De la même manière, aucune contrainte n'est définie sur l'idempotence des opérations. Ces libertés impliquent qu'il peut être nécessaire que les opérations soient livrées au CRDT en respectant un ordre donné et en garantissant leur livraison en exactement une fois pour garantir la convergence. Ainsi, un intergiciel chargé de la diffusion et de la livraison des opérations est usuellement associé aux CRDTs synchronisés par opérations pour respecter ces contraintes. Il s'agit de la couche de livraison de messages que nous avons introduit dans le cadre de notre modèle du système (cf. section 2.1, page 12).

Généralement, les CRDTs synchronisés par opérations sont présentés dans la littérature comme nécessitant une livraison causale des opérations.

Définition 21 (Modèle de livraison causale)**.** Le modèle de livraison causale définit que, pour toute paire de messages m_1 et m_2 d'une exécution, si $m_1 \rightarrow m_2$, alors la couche de livraison de l'ensemble des noeuds doit livrer le message m_1 à l'application avant de livrer le message m_2.

Ce modèle de livraison permet de respecter le modèle de cohérence causale et ainsi de simplifier le raisonnement sur les exécutions.

Ce modèle de livraison introduit néanmoins plusieurs effets négatifs. Tout d'abord, ce modèle peut provoquer un délai dans l'intégration des modifications. En effet, la perte d'une opération par le réseau provoque la mise en attente de la livraison des opérations suivantes. Les opérations mises en attente ne pourront en effet être livrées qu'une fois l'opération perdue re-diffusée et livrée.

De plus, il nécessite que des informations de causalité précises soient attachées à chaque opération. Pour cela, les systèmes reposent généralement sur l'utilisation de vecteurs de version [54, 55]. Or, la taille de cette structure de données croît de manière linéaire avec le nombre de noeuds du système. Les métadonnées de causalité peuvent ainsi représenter la

majorité des données diffusées sur le réseau [9]. Cependant, nous observons que la livraison dans l'ordre causal de toutes les opérations n'est pas toujours nécessaire. Par exemple, l'ordre d'intégration de deux opérations d'ajout d'éléments différents dans un Ensemble n'a pas d'importance. Nous pouvons alors nous affranchir du modèle de livraison causale pour accélérer la vitesse d'intégration des modifications et pour réduire les métadonnées envoyées.

Pour compenser la perte d'opérations par le réseau et ainsi garantir la livraison à terme des opérations, la couche de livraison des opérations doit mettre en place un mécanisme d'anti-entropie, c.-à-d. un mécanisme permettant de détecter et ré-échanger les messages perdus. Plusieurs mécanismes de ce type ont été proposés dans la littérature [57, 58, 59, 60] *Matthieu: TODO : Ajouter refs Scuttlebutt si applicable à Op-based* et proposent des compromis variés entre complexité en temps, complexité spatiale et consommation réseau.

Nous illustrons le modèle de synchronisation par opérations à l'aide de la Figure 2.9. Dans ce nouvel exemple, les noeuds diffusent les modifications qu'ils effectuent sous la forme d'opérations. Nous considèrons que le CRDT utilisé est un CRDT pur synchronisé par opérations, c.-à-d. que les modifications et opérations sont confondues, et qu'il autorise une livraison dans le désordre des opérations *add*.

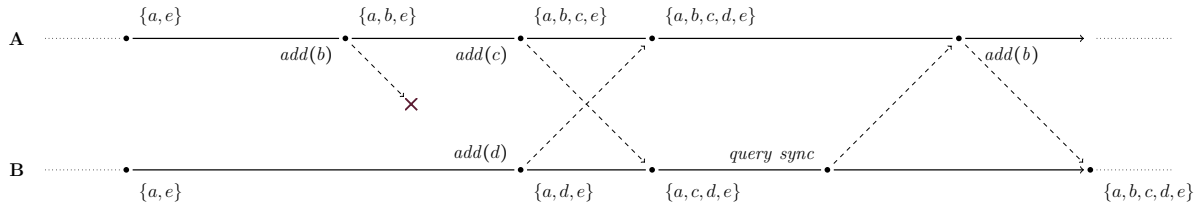


FIGURE 2.9 – Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par opérations

Le noeud A diffuse donc les opérations $add(b)$ et $add(c)$. Il reçoit ensuite l'opération $add(d)$ de B, qu'il intègre à sa copie. Il obtient alors l'état $\{a, b, c, d, e\}$.

De son côté, le noeud B ne reçoit initialement pas l'opération $add(b)$ suite à une défaillance réseau. Il génère et diffuse $add(d)$ puis reçoit l'opération $add(c)$. Comme indiqué précédemment, nous considérons que la livraison causale des opérations *add* n'est pas obligatoire dans cet exemple, cette opération est alors intégrée sans attendre. Le noeud B obtient alors l'état $\{a, c, d, e\}$.

Ensuite, le mécanisme d'anti-entropie du noeud B se déclenche. Le noeud B envoie alors à A une demande de synchronisation contenant un résumé de son état, e.g. son vecteur de version. À partir de cette donnée, le noeud A détermine que B n'a pas reçu l'opération $add(a)$. Il génère alors une réponse contenant cette opération et lui envoie. À la réception de l'opération, le noeud B l'intègre. Il obtient l'état $\{a, b, c, d, e\}$ et converge ainsi avec A.

9. La relation de causalité étant transitive, les opérations et leurs relations de causalité forment un DAG. [56] propose d'ajouter en dépendances causales d'une opération seulement les opérations correspondant aux extremités du DAG au moment de sa génération. Ce mécanisme plus complexe permet de réduire la consommation réseau, mais induit un surcoût en calculs et en mémoire utilisée.

Avant de conclure, nous noterons qu'il est nécessaire pour les noeuds de maintenir leur journal d'opérations. En effet, les noeuds l'utilisent pour renvoyer les opérations manquées lors de l'exécution du mécanisme d'anti-entropie évoqué ci-dessus. Ceci se traduit par une augmentation perpétuelle des métadonnées des CRDTs synchronisés par opérations. Pour y pallier, des travaux [50, 61] proposent de tronquer le journal des opérations pour en supprimer les opérations connues de tous. Les noeuds reposent alors sur la notion de stabilité causale [62] pour déterminer les opérations supprimables de manière sûre.

Définition 22 (Stabilité causale)**.** Une opération est stable causalement lorsqu'elle a été intégrée par l'ensemble des noeuds du système. Ainsi, toute opération future dépend causalement des opérations causalement stables, c.-à-d. les noeuds ne peuvent plus générer d'opérations concurrentes aux opérations causalement stables.

Un mécanisme d'instantané *Matthieu: TODO : Ajouter refs* doit néanmoins être associé au mécanisme de troncature du journal pour générer un état équivalent à la partie tronquée. Ce mécanisme est en effet nécessaire pour permettre un nouveau noeud de rejoindre le système et d'obtenir l'état courant à partir de l'instantané et du journal tronqué.

Pour résumer, cette approche permet de mettre en place un système en composant un CRDT synchronisé par opérations avec une couche de livraison des messages. Mais comme illustré ci-dessus, chaque CRDT synchronisé par opérations établit les propriétés de ses différentes opérations et délègue potentiellement des responsabilités à la couche de livraison. Une partie de la complexité de cette approche réside ainsi dans l'ajustement du couple $\langle CRDT, \textit{couche livraison} \rangle$ pour régler finement et optimiser leur fonctionnement en tandem. Des travaux [50, 61] ont proposé un patron de conception pour modéliser ces deux composants et leurs interactions. Cependant, ce patron repose sur l'hypothèse d'une livraison causale des opérations et n'est donc pas optimal. *Matthieu: TODO : Vérifier que c'est bien le cas dans [61]*

Synchronisation par différences d'états

Almeida et al. [51] introduisent un nouveau modèle de synchronisation pour CRDTs. La proposition de ce modèle est nourrie par les observations suivantes :

(i) Les CRDTs synchronisés par opérations sont sujets aux défaillances du réseau et nécessitent généralement pour pallier à cette une livraison des opérations respectant le modèle de livraison causal.

(ii) Les CRDTs synchronisés par états pâtissent du surcoût induit par la diffusion de leurs états complets, généralement croissant de manière monotone.

Pour pallier aux faiblesses de chaque approche et allier le meilleur des deux mondes, les auteurs proposent les CRDTs synchronisés par différences d'états [51, 52, 63]. Il s'agit en fait d'une sous-famille des CRDTs synchronisés par états. Ainsi, comme ces derniers, ils disposent d'une fonction `merge` associative, commutative et idempotente qui permet de produire la LUB de deux états, c.-à-d. l'état correspond à la borne supérieure de ces deux états.

La spécificité des CRDTs synchronisés par différences d'états est qu'une modification locale produit en retour un delta. Un delta encode la modification effectuée sous la forme d'un état du lattice. Les deltas étant des états, ils peuvent être diffusés puis intégrés par les autres noeuds à l'aide de la fonction `merge`. Ceci permet de bénéficier des propriétés d'associativité, de commutativité et d'idempotence offertes par cette fonction. Les CRDTs synchronisés par différences d'états offrent ainsi :

(i) Une diffusion des modifications avec un surcoût pour le réseau proche de celui des CRDTs synchronisés par opérations.

(ii) Une résistance aux défaillances réseaux similaire celle des CRDTs synchronisés par états.

Cette définition des CRDTs synchronisés par différences d'états, introduite dans [51, 52], fut ensuite précisée dans [63]. Dans cet article, les auteurs précisent qu'utiliser des éléments irréductibles (cf. Définition 20, page 23) comme deltas est optimal du point de vue de la taille des deltas produits.

Concernant la diffusion des modifications, les CRDTs synchronisés par différences d'états autorisent un large éventail de possibilités. Par exemple, les deltas peuvent être diffusés et intégrés de manière indépendante. Une autre approche possible consiste à tirer avantage du fait que les deltas sont des états : il est possible d'agréger plusieurs deltas à l'aide de la fonction `merge`, éliminant leurs éventuelles redondances. Ainsi, la fusion de deltas permet ensuite de diffuser un ensemble de modifications par le biais d'un seul et unique delta, minimal. Et en dernier recours, les CRDTs synchronisés par différences d'états peuvent adopter le même schéma de diffusion que les CRDTs synchronisés par états, c.-à-d. diffuser leur état complet de manière périodique. Chacune de ces approches proposent un compromis entre délai d'intégration des modifications, surcoût en métadonnées, calculs et bande-passante [63]. Ainsi, il est possible pour un système utilisant des CRDTs synchronisés par différences d'états de sélectionner la technique de diffusion des modifications la plus adaptée à ses besoins, ou même d'alterner entre plusieurs en fonction de son état courant.

Nous illustrons cette approche avec la Figure 2.10. Dans cet exemple, nous considérons que les noeuds adoptent la seconde approche évoquée, c.-à-d. que périodiquement les noeuds aggrégent les deltas issus de leurs modifications et diffusent le delta résultant.

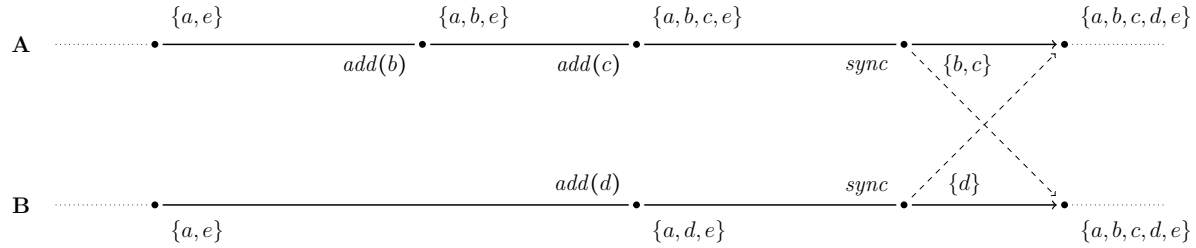


FIGURE 2.10 – Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par différences d'états

Le noeud A effectue les modifications $add(b)$ et $add(c)$, qui retournent respectivement les deltas $\{b\}$ et $\{c\}$. Le noeud A aggrége ces deltas et diffuse donc le delta suivant $\{b, c\}$.

Quant au noeud B, il effectue la modification $add(d)$ qui produit le delta $\{d\}$. S'agissant de son unique modification, il diffuse ce delta inchangé.

Quand A (resp. B) reçoit le delta $\{d\}$ (resp. $\{b, c\}$), il l'intègre à sa copie en utilisant la fonction `merge`. Les deux noeuds convergent alors à l'état $\{a, b, c, d, e\}$.

La synchronisation par différences d'états permet donc de réduire la taille des messages diffusés sur le réseau par rapport à la synchronisation par états. Cependant, il est important de noter que la décomposition en deltas entraîne la perte d'une des propriétés intrinsèques des CRDTs synchronisés par états : le respect du modèle de cohérence causale. En effet, sans mécanisme supplémentaire, la perte ou le ré-ordonnement de deltas par le réseau peut mener à une livraison dans le désordre des modifications à l'un des noeuds. S'ils souhaitent assurer une intégration causale des modifications, les CRDTs synchronisés par différences d'états doivent donc définir et ajouter à leur spécification un mécanisme similaire à la couche de livraison des CRDTs synchronisés par opérations.

Ainsi, les CRDTs synchronisés par différences d'états sont une évolution prometteuse des CRDTs synchronisés par états. Ce modèle de synchronisation rend ces CRDTs utilisables dans les systèmes temps réels sans introduire de contraintes sur la fiabilité du réseau. Mais pour cela, il ajoute une couche supplémentaire de complexité à la spécification des CRDTs synchronisés par états, c.-à-d. le mécanisme dédié à la livraison des deltas.

Synthèse

Ainsi, plusieurs modèles de synchronisation ont été proposées pour permettre aux noeuds utilisant un CRDT pour répliquer une donnée de diffuser leurs modifications et d'intégrer celles des autres. Nous récapitulons dans cette section les principales propriétés et différences entre ces modèles.

Tout d'abord, rappelons que chaque approche repose sur l'utilisation d'un sup-demi-treillis pour assurer la convergence forte. Dans le cadre des CRDTs synchronisés par états et des CRDTs synchronisés par différences d'états, ce sont les états du CRDTs même qui forment un sup-demi-treillis.

Ce n'est pas exactement le cas dans le cadre des CRDTs synchronisés par opérations. Comme indiqué précédemment, les CRDTs synchronisés par opérations demandent à la couche de livraison des messages qui leur est associée qu'elle satisfasse un ensemble de contraintes. Si la couche de livraison ne garantit pas ces contraintes, e.g. les opérations sont livrées dans le désordre, l'état des noeuds peut diverger définitivement. Ainsi, pour être précis, c'est le couple $\langle \textit{etats du CRDT}, \textit{couche livraison} \rangle$ qui forme un sup-demi-treillis dans le cadre de ce modèle de synchronisation.

La principale différence entre les modèles de synchronisation proposés réside dans l'unité utilisée lors d'une synchronisation. Le modèle de synchronisation par états, de manière équivoque, utilise les états complets. L'intégration des modifications effectuées par un noeud dans la copie locale d'un second se fait alors en diffusant l'état du premier au second et en fusionnant cet état avec l'état du second.

Le modèle de synchronisation par opérations repose sur des opérations pour diffuser les modifications. Les opérations encodent les modifications sous la forme d'un ou plusieurs états spécifiques du sup-demi-trellis : les éléments irréductibles (cf. Définition 20, page

23). L'intégration des modifications d'un noeud par un second se fait alors en diffusant les opérations correspondant aux modifications et en intégrant chacune d'entre elle à la copie locale du second.

Le modèle de synchronisation par différences d'états permet quant à lui d'intégrer les modifications soit par le biais d'éléments irréductibles, soit par le biais d'états complets. Dans les deux cas, les CRDTs synchronisés par différences d'états reposent sur la fonction de fusion du sup-demi-treillis pour intégrer les modifications.

De cette différence d'unité de synchronisation découle l'ensemble des différences entre ces modèles. La capacité d'intégrer les modifications par le biais d'une fusion d'états permet aux CRDTs synchronisé par états et différences d'états de résister aux défaillances du réseau. En effet, la perte, le ré-ordonnement ou la duplication de messages, c.-à-d. d'états ou de différences d'états, n'empêche pas la convergence des noeuds. Tant que deux noeuds peuvent à terme échanger leur états respectifs et les fusionner, la fonction de fusion garantit qu'ils obtiendront à terme des états équivalents.

À l'inverse, la perte, le ré-ordonnement ou la duplication de messages, c.-à-d. d'opérations, peut entraîner une divergence des noeuds dans le cadre du modèle de synchronisation par opérations. Pour éviter ce problème, la couche de livraison de messages associée au CRDT doit satisfaire le modèle de livraison requis par ce dernier.

Un autre aspect impacté par l'unité de synchronisation est la fréquence de synchronisation. La synchronisation par états nécessite de diffuser son état complet pour diffuser ses modifications. En fonction du type de données, le coût réseau pour diffuser chaque modification dès qu'elle est effectuée peut s'avérer prohibitif. Ce modèle de synchronisation repose donc généralement sur une synchronisation périodique, c.-à-d. chaque noeud diffuse son état périodiquement.

À l'inverse, la synchronisation par éléments irréductibles, que ça soit sous la forme d'opérations ou leur forme primaire, induit un coût réseau raisonnable : les éléments sont généralement petits et de taille fixe. Les modèles de synchronisation par opérations et par différences d'états permettent donc de diffuser des modifications dès leur génération. Ceci permet aux noeuds du système d'intégrer les modifications effectuées par les autres noeuds de manière plus fréquente, voire en temps réel.

Finalement, la dernière différence entre ces modèles concerne le modèle de cohérence causale (cf. Définition 19, page 23). Par nature, le modèle de synchronisation par états garantit le respect du modèle de cohérence causale. En effet, un état correspond à l'intégration d'un ensemble de modifications. De manière similaire, le résultat de la fusion de deux états correspond à l'intégration de l'union de leur ensemble respectif de modifications. Ce modèle de synchronisation empêche donc l'intégration d'une modification sans avoir intégré aussi les modifications l'ayant précédé d'après la relation *happens-before*.

À l'inverse, par défaut, les modèles de synchronisation par opérations ou différences d'états permettent l'intégration d'un élément irréductible sans avoir intégré au préalable les éléments irréductibles l'ayant précédé d'après la relation *happens-before*. Pour satisfaire le modèle de cohérence causale, les CRDTs adoptant ces modèles de synchronisation doivent être associés à une couche de livraison de messages garantissant leur livraison causale (cf. Définition 21, page 24).

Nous récapitulons le contenu de cette discussion sous la forme du Tableau 2.1.

TABLE 2.1 – Récapitulatif comparatif des différents modèles de synchronisation pour CRDTs

	Sync. par états	Sync. par opérations	Sync. par diff. d'états
Forme un sup-demi-treillis	✓	✓	✓
Intègre modifications par fusion d'états	✓	✗	✓
Intègre modifications par élts irréductibles	✗	✓	✓
Résiste nativ. aux défaillances réseau	✓	✗	✓
Adapté pour systèmes temps réel	✗	✓	✓
Offre nativ. modèle de cohérence causale	✓	✗	✗

2.3 Séquences répliquées sans conflits

Dans le cadre des travaux de cette thèse, nous nous sommes focalisés sur les CRDTs pour un type de donnée précis : la *Séquence*.

La Séquence, aussi appelée *Liste*, est un type abstrait de données représentant une collection ordonnée et de taille dynamique d'éléments. Dans une séquence, un même élément peut apparaître à de multiples reprises. Chacune des occurrences de cet élément est alors considérée comme distincte.

Dans le cadre de ce manuscrit, nous travaillons sur des séquences de caractères. Cette restriction du domaine se fait sans perte en généralité. Nous illustrons par la Figure 2.11 notre représentation des séquences.

H	E	L	L	O
0	1	2	3	4

FIGURE 2.11 – Représentation de la séquence "HELLO"

Dans la Figure 2.12, nous présentons la spécification algébrique du type Séquence que nous utilisons.

payload
$S \in Seq\langle E \rangle$

constructor
$emp \quad : \quad \longrightarrow S$

mutators
$ins \quad : \quad S \times \mathbb{N} \times E \quad \longrightarrow S$
$rmv \quad : \quad S \times \mathbb{N} \quad \longrightarrow S$

queries
$len \quad : \quad S \quad \longrightarrow \mathbb{N}$
$rd \quad : \quad S \quad \longrightarrow Arr\langle E \rangle$

FIGURE 2.12 – Spécification algébrique du type abstrait usuel Séquence

Celle-ci définit deux modifications :

(i) $ins(s, i, e)$, qui permet d'insérer un élément donné e à un index donné i dans une séquence s de taille m. Cette modification renvoie une nouvelle séquence construite de la manière suivante :

$$\forall s \in S, e \in E, i \in [0, m] \mid m = len(s), s = \langle e_0, ..., e_{i-1}, e_i, ..., e_{m-1} \rangle \cdot ins(s, i, e) = \langle e_0, ..., e_{i-1}, e, e_i, ..., e_{m-1} \rangle$$

(ii) $rmv(s, i)$, qui permet de retirer l'élément situé à l'index i dans une séquence s de taille m. Cette modification renvoie une nouvelle séquence construite de la manière suivante :

$$\forall s \in S, e \in E, i \in [0, m[\mid m = len(s), s = \langle e_0, ..., e_{i-1}, e_i, ei + 1, ..., e_{m-1} \rangle \cdot rmv(s, i) = \langle e_0, ..., e_{i-1}, e_{i+1}, ..., e_{m-1} \rangle$$

Les modifications définies dans Figure 2.12, *ins* et *rmv*, ne permettent respectivement que l'insertion ou la suppression d'un élément à la fois. Cette simplification du type se fait cependant sans perte de généralité, la spécification pouvant être étendue pour insérer successivement plusieurs éléments à partir d'un index donné ou retirer plusieurs éléments consécutifs.

La spécification définit aussi deux observateurs :

(i) $len(s)$, qui permet de récupérer le nombre d'éléments présents dans une séquence s.

(ii) $rd(s)$, qui permet de consulter l'état d'une séquence s. L'état de la séquence est retournée sous la forme d'un Tableau, c.-à-d. une collection ordonnée de taille fixe d'éléments. Comme pour le type Ensemble, nous considérons que *rd* est utilisé de manière implicite après chaque modification dans nos exemples.

Cette spécification du type Séquence est une spécification séquentielle. Les modifications sont définies pour être effectuées l'une après l'autre. Si plusieurs noeuds répliquent une même séquence et la modifient en concurrence, l'intégration de leurs opérations respectives dans des ordres différents résulte en des états différents. Nous illustrons ce point avec la Figure 2.13.

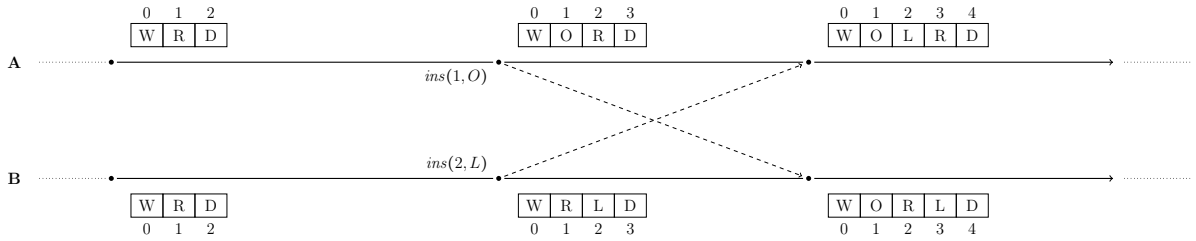


FIGURE 2.13 – Modifications concurrentes d'une séquence

Dans cet exemple, deux noeuds A et B partagent et éditent collaborativement une même séquence. Celle-ci correspond initialement à la chaîne de caractères "WRD". Le

noeud A insère le caractère "O" à l'index 1, obtenant ainsi la séquence "WORD". En concurrence, le noeud B insère lui le caractère "L" à l'index 2 pour obtenir "WRLD".

Les deux noeuds diffusent ensuite leur opération respective puis intègre celle de leur pair. Nous constatons alors une divergence. En effet, l'intégration de la modification $ins(2, L)$ par le noeud A ne produit pas l'effet escompté, c.-à-d. produire la chaîne "WORLD", mais la chaîne "WOLRD".

Cette divergence est dûe à la non-commutativité de la modification *ins* avec elle-même. En effet, celle-ci se base sur un index pour déterminer où placer le nouvel élément. Cependant, les index sont eux-mêmes modifiés par *ins*. Ainsi, l'intégration dans des ordres différents de modifications *ins* sur un même état initial résulte en des états différents. Plus généralement, nous observons que chaque paire possible de modifications du type Séquence, c.-à-d. $\langle ins, ins\rangle$, $\langle ins, del\rangle$ et $\langle del, del\rangle$, n'est pas commutative.

La non-commutativité des modifications du type Séquence fut l'objet de nombreux travaux de recherche dans le domaine de l'édition collaborative. Pour résoudre ce problème, l'approche Operational Transformation (OT) [64, 65] fut initialement proposée. Cette approche propose de transformer une modification par rapport aux modifications concurrentes intégrées pour tenir compte de leur effet. Elle se décompose en deux parties :

(i) Un algorithme de contrôle [66, 67, 68], qui définit par rapport à quelles modifications une nouvelle modification distante doit être transformée avant d'être intégrée à la copie.

(ii) Des fonctions de transformations [64, 66, 69, 70], qui définissent comment une modification doit être transformée par rapport à une autre modification pour tenir compte de son effet.

Cependant, bien que de nombreuses fonctions de transformations pour le type Séquence ont été proposées, seule la correction des Tombstone Transformation Functions (TTF) [70] a été éprouvée pour les systèmes P2P à notre connaissance. De plus, les algorithmes de contrôle compatibles reposent sur une livraison causale des modifications, et donc l'utilisation de vecteurs d'horloges. Cette approche est donc inadaptée aux systèmes P2P dynamiques.

Néanmoins, une contribution importante de l'approche OT fut la définition d'un modèle de cohérence que doivent respecter les systèmes d'édition collaboratif : le modèle Convergence, Causality preservation, Intention preservation (CCI) [71].

Définition 23 (Modèle CCI)**.** Le modèle de cohérence CCI définit qu'un système d'édition collaboratif doit respecter les critères suivants :

Définition 23.1 (Convergence)**.** Le critère de *Convergence* indique que des noeuds ayant intégrés le même ensemble de modifications convergent à un état équivalent.

Définition 23.2 (Préservation de la causalité)**.** Le critère de *Préservation de la causalité* indique que si une modification m_1 précède une autre modification m_2 d'après la relation *happens-before*, c.-à-d. $m_1 \rightarrow m_2$, m_1 doit être intégrée avant m_2 par les noeuds du système.

Définition 23.3 (Préservation de l'intention)**.** Le critère de *Préservation de l'intention* indique que l'intégration d'une modification par un noeud distant doit reproduire l'effet de la modification sur la copie du noeud d'origine, indépendamment des modifications concurrentes intégrées.

De manière similaire à [72], nous ajoutons le critère de *Passage à l'échelle* à ces critères :

Définition 24 (Passage à l'échelle)**.** Le critère de *Passage à l'échelle* indique que le nombre de noeuds du système ne doit avoir qu'un impact sous-linéaire sur les complexités en temps, en espace et sur le nombre et la taille des messages.

Nous constatons cependant que les critères 23.2 et 24 peuvent être contraires. En effet, pour respecter le modèle de cohérence causale, un système peut nécessiter une livraison causale des modifications, e.g. un CRDT synchronisé par opérations dont seules les opérations concurrentes sont commutatives. La livraison causale implique un surcoût computationnel, en métadonnées et en taille des messages qui est fonction du nombre de participants du système [73]. Ainsi, dans le cadre de nos travaux, nous cherchons à nous affranchir du modèle de livraison causale des modifications, ce qui peut nécessiter de relaxer le modèle de cohérence causale.

C'est dans une optique similaire que fut proposé WOOT [74], un modèle de séquence répliquée qui pose les fondations des CRDTs. Depuis, plusieurs CRDTs pour Séquence furent définies [75, 76, 72]. Ces CRDTs peuvent être répartis en deux approches : l'approche à pierres tombales [74, 75] et l'approche à identifiants densément ordonnés [76, 72]. L'état d'une séquence pouvant croître de manière infinie, ces CRDTs sont synchronisés par opérations pour limiter la taille des messages diffusés. À notre connaissance, seul [29] propose un CRDT pour Séquence synchronisé par différence d'états.

Dans la suite de cette section, nous présentons les différents CRDTs pour Séquence de la littérature.

2.3.1 Approche à pierres tombales

WOOT

WOOT [74] fait suite aux travaux présentés dans [70]. Il est considéré a posteriori comme le premier CRDT synchronisé par opérations pour Séquence [10]. Conçu pour l'édition collaborative P2P, son but est de surpasser les limites de l'approche OT évoquées précédemment, c.-à-d. le coût du mécanisme de livraison causale.

L'intuition de WOOT est la suivante : WOOT modifie la sémantique de la modification *ins* pour qu'elle corresponde à l'insertion d'un nouvel élément entre deux autres, et non plus à l'insertion d'un nouvel élément à une position donnée. Par exemple, l'insertion de l'élément "B" dans la séquence "AC" pour obtenir l'état "ABC", c.-à-d. $ins(1, B)$, devient $ins(A \prec B \prec C)$.

Ce changement, qui est compatible avec l'intention des utilisateur-rices, n'est cependant pas anodin. En effet, il permet à WOOT de rendre *ins* commutative avec les modifications concurrentes, en exprimant la position du nouvel élément de manière relative à d'autres éléments et non plus via un index qui est spécifique à un état donné.

Afin de préciser quels éléments correspondent aux prédécesseur et successeur de l'élément inséré, WOOT repose sur un système d'identifiants. WOOT associe ainsi un identifiant unique à chaque élément de la séquence.

10. [21] n'ayant formalisé les CRDTs qu'en 2007.

Définition 25 (Identifiant WOOT)**.** Un identifiant WOOT est un couple $\langle nodeId, nodeSeq \rangle$ avec

(i) *nodeId*, l'identifiant du noeud qui génère cet identifiant WOOT. Est supposé unique.

(ii) *seq*, un entier propre au noeud, servant d'horloge logique. Est incrémenté à chaque génération d'identifiant WOOT.

Dans le cadre de ce manuscrit, nous utiliserons pour former les identifiants WOOT le nom de du noeud (e.g. A) comme *nodeId* et un entier naturel, en démarrant à 1, comme *nodeSeq*. Nous les représenterons de la manière suivante *nodeId nodeSeq*, e.g. $A1$.

Avant de continuer, notons qu'un identifiant WOOT est bel et bien unique, deux noeuds ne pouvant utiliser le même *nodeId* et un noeud n'utilisant jamais deux fois le même *nodeSeq*. Finalement, précisons que cette structure d'identifiant et son usage possible pour le suivi de la causalité furent ensuite mis en évidence par [77] sous le nom de *Dot*.

Les modifications *ins* et *rmv* sont dès lors redéfinies pour devenir des opérations en tirant profit des identifiants. Par exemple, considérons une séquence WOOT représentant "AC" et qui associe respectivement les identifiants $A1$ et $A2$ aux éléments "A" et "C". L'insertion de l'élement de l'élement "B" dans cette séquence pour obtenir l'état "ABC", c.-à-d. $ins(A \prec B \prec C)$, devient par exemple $ins(A1 \prec \langle B1, B \rangle \prec A2)$. De manière similaire, la suppression de l'élément "B" dans cette séquence pour obtenir l'état "AC", c.-à-d. $rmv(1)$, devient $rmv(B1)$.

WOOT utilise des pierres tombales pour rendre commutative *ins*, qui nécessite la présence des deux éléments entre lesquels nous insérons un nouvel élément, avec *rmv*. Ainsi, lorsqu'un élément est retiré, une pierre tombale est conservée dans la séquence pour indiquer sa présence passée. Les données de l'élément sont elles supprimées. Dans le cadre d'une Séquence WOOT, *rmv* a donc pour effet de masquer l'élément.

Finalement, WOOT définit $<_{id}$, un ordre strict total sur les identifiants associés aux éléments. En effet, la relation $\prec$ n'est pas suffisante pour rendre les opérations *ins* commutatives, puisqu'elle ne spécifie qu'un ordre partiel entre les éléments. Plus précisément, $\prec$ ne permet pas d'ordonner les éléments insérés en concurrence et possédant les mêmes prédecesseur et successeur, e.g. $ins(a \prec 1 \prec b)$ et $ins(a \prec 2 \prec b)$. Pour que tous les noeuds convergent, ils doivent choisir comment ordonner ces éléments de manière déterministe et indépendante de l'ordre de réception des modifications. Ils utilisent pour cela $<_{id}$.

Définition 26 (Relation $<_{id}$)**.** La relation $<_{id}$ définit que, étant donné deux identifiants $id_1 = \langle nodeId_1, nodeSeq_1 \rangle$ et $id_2 = \langle nodeId_2, nodeSeq_2 \rangle$, nous avons :

$$id_1 <_{id} id_2 \quad \textit{iff} \quad (nodeId_1 < nodeId_2) \quad \vee \\ (nodeId_1 = nodeId_2 \wedge nodeSeq_1 < nodeSeq_2)$$

Notons que l'ordre définit par $<_{id}$ correspond à l'ordre lexicographique sur les composants des identifiants.

De cette manière, WOOT offre une spécification de la Séquence dont les opérations sont commutatives, c.-à-d. ne génèrent pas de conflits. Nous récapitulons son fonctionnement à l'aide de la Figure 2.14.

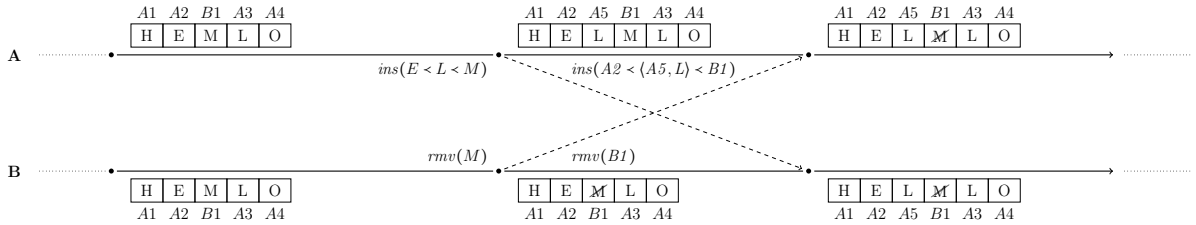


FIGURE 2.14 – Modifications concurrentes d'une séquence répliquée WOOT

Dans cet exemple, deux noeuds A et B partagent et éditent collaborativement une séquence répliquée WOOT. Initialement, ils possèdent le même état : la séquence contient les éléments "HEMLO", et à chaque élément est associé un identifiant, e.g. $A1$, $B1$, $A2$...

Le noeud A insère l'élément "L" entre les éléments "E" et "M", c.-à-d. $ins(E \prec L \prec M)$. WOOT convertit cette modification en opération $ins(A2 \prec \langle A5, L\rangle \prec B1)$. L'opération est intégrée à la copie locale, ce qui produit l'état "HELMLO", puis diffusée sur le réseau.

En concurrence, le noeud B supprime l'élément "M" de la séquence, c.-à-d. $rmv(M)$. De la même manière, WOOT génère l'opération correspondante $rmv(B1)$. Comme expliqué précédemment, l'intégration de cette opération ne supprime pas l'élément "M" de l'état mais se contente de le masquer. L'état produit est donc "HE~~M~~LO". L'opération est ensuite diffusée.

A (resp. B) reçoit ensuite l'opération de B, $rmv(B1)$ (resp. A, $ins(A2 \prec \langle A5, L\rangle \prec B1)$), et l'intègre à sa copie. Les opérations de WOOT étant commutatives, les noeuds obtiennent le même état final : "HEL~~M~~LO".

Grâce à la commutativité de ses opérations, WOOT s'affranchit du modèle de livraison causale nécessitant l'utilisation coûteuse de vecteurs d'horloges. WOOT met en place un modèle de livraison sur-mesure basé sur les pré-conditions des opérations :

Définition 27 (Modèle de livraison WOOT)**.** Le modèle de livraison WOOT définit que :

(i) Une opération doit être livrée exactement une fois à chaque noeud [11].

(ii) Une opération $ins(predId \prec \langle id, elt\rangle \prec succId)$ ne peut être livrée à un noeud qu'après la livraison des opérations d'insertion des éléments associés à $predId$ et $succId$.

(iii) L'opération $rmv(id)$ ne peut être livrée à un noeud qu'après la livraison de l'opération d'insertion de l'élément associé à id.

Ce modèle de livraison ne requiert qu'une quantité fixe de métadonnées associées à chaque opération pour être respecté. WOOT est donc adapté aux systèmes P2P dynamiques.

WOOT souffre néanmoins de plusieurs limites. La première d'entre elles correspond à l'utilisation de pierres tombales dans la séquence répliquée. En effet, comme indiqué précédemment, la modification rmv ne supprime que les données de l'élément concerné. L'identifiant qui lui a été associé reste lui présent dans la séquence à son emplacement.

11. Néanmoins, les algorithmes d'intégration des opérations, notamment celui pour l'opération *ins*, pourraient être aisément modifiés pour être idempotents. Ainsi, la livraison répétée d'une même opération deviendrait possible, ce qui permettrait de relaxer cette contrainte en *une livraison au moins une fois*.

Une séquence WOOT ne peut donc que croître, ce qui impacte négativement sa complexité en espace ainsi que ses complexités en temps.

Oster et al. [74] font cependant le choix de ne pas proposer de mécanisme pour purger les pierres tombales. En effet, leur motivation est d'utiliser ces pierres tombales pour proposer un mécanisme d'*undo*, une fonctionnalité importante dans le domaine de l'édition collaborative. *Matthieu: TODO : Ajouter refs, celles utilisées dans [74].* Cette piste de recherche est développée dans [78].

Une seconde limite de WOOT concerne la complexité en temps de l'algorithme d'intégration des opérations d'insertion. En effet, celle-ci est en $\mathcal{O}(H^3)$ avec H le nombre de modifications ayant été effectuées sur le document [79]. Plusieurs évolutions de WOOT sont proposées pour mitiger cette limite : WOOTO [80] et WOOTH [79].

Weiss et al. [80] remanient la structure des identifiants associés aux éléments. Cette modification permet un algorithme d'intégration des opérations *ins* avec une meilleure complexité en temps, $\mathcal{O}(H^2)$. Ahmed-Nacer et al. [79] se basent sur WOOTO et proposent l'utilisation de structures de données améliorant la complexité des algorithmes d'intégration des opérations, au détriment des métadonnées stockées localement par chaque noeud. Cependant, cette évolution ne permet ici pas de réduire l'ordre de grandeur des opérations *ins*.

Néanmoins, l'évaluation expérimentale des différentes approches pour l'édition collaborative P2P en temps réel menée dans [79] a montré que les CRDTs de la famille WOOT n'étaient pas assez efficaces. Dans le cadre de cette expérience, des utilisateur-rices effectuaient des tâches d'édition collaborative données. Les traces de ces sessions d'édition collaboratives furent ensuite rejouées en utilisant divers mécanismes de résolution de conflits, dont WOOT, WOOTO et WOOTH. Le but était de mesurer les performances de ces mécanismes, notamment leurs temps d'intégration des modifications et opérations. Dans le cas de la famille WOOT, Ahmed-Nacer et al. ont constaté que ces temps dépassaient parfois 50ms. Il s'agit là de la limite des délais acceptables par les utilisateur-rices d'après [81, 82]. Ces performances disqualifient donc les CRDTs de la famille WOOT comme approches viables pour l'édition collaborative P2P temps réel.

Replicated Growable Array

Replicated Growable Array (RGA) [75] est le second CRDT pour Séquence appartenant à l'approche à pierres tombales. Il a été spécifié dans le cadre d'un effort pour établir les principes nécessaires à la conception de Replicated Abstract Data Types (RADTs).

Dans cet article, les auteurs définissent et se basent sur 2 principes pour concevoir RADTs. Le premier d'entre eux est l'Commutativité des Opérations (OC).

Définition 28 (Commutativité des Opérations)**.** La Commutativité des Opérations (OC) définit que toute paire possible d'opérations concurrentes du RADT doit être commutative.

Ce principe permet de garantir que l'intégration par différents noeuds d'une même séquence d'opérations concurrentes, mais dans des ordres différents, résultera en un état équivalent.

Le second principe sur lequel reposent les RADTs est la Transitivité de la Précédence (PT).

Définition 29 (Transitivité de la Précédence)**.** La Transitivité de la Précédence (PT) se base sur une relation de précédence, notée $\dashrightarrow$.

Définition 29.1 (Relation de précédence)**.** La relation de précédence définit qu'étant donné deux opérations, o_1 et o_2, l'intention de o_2 doit être préservée par rapport à celle de o_1, noté $o_1 \dashrightarrow o_2$, si et seulement si :

(i) $o_1 \rightarrow o_2$ ou

(ii) $o_1 \parallel o_2$ et o_2 a une priorité supérieure à o_1.

PT définit qu'étant donné trois opérations, o_1, o_2 et o_3, si $o_1 \dashrightarrow o_2$ et $o_2 \dashrightarrow o_3$, alors nous avons $o_1 \dashrightarrow o_3$.

Ce second principe offre une méthode pour concevoir un ensemble d'opérations commutatives. Il permet aussi d'exprimer la priorité des opérations par rapport aux opérations dont elles dépendent causalement.

À partir de ces principes, les auteurs proposent plusieurs RADTs : Replicated Fixed-Size Array (RFA), Replicated Hash Table (RFT) et Replicated Growable Array (RGA), qui nous intéresse ici.

Dans RGA, l'intention de l'insertion est définit comme l'insertion d'un nouvel élément directement après un élément existant. Ainsi, RGA se base sur le prédecesseur d'un élément pour déterminer où l'insérer. De fait, tout comme WOOT, RGA repose sur un système d'identifiants qu'il associe aux éléments pour pouvoir s'y référer par la suite.

Les auteurs proposent le modèle de données suivant comme identifiants :

Définition 30 (Identifiants S4Vector)**.** Les identifiants S4Vector sont de la forme $\langle ssid, sum, ssn, seq \rangle$ avec :

(i) *ssid*, l'identifiant de la session de collaboration.

(ii) *sum*, la somme du vecteur d'horloges courant du noeud auteur de l'élément.

(iii) *ssn*, l'identifiant du noeud auteur de l'élément.

(iv) *seq*, le numéro de séquence de l'auteur de l'élément à son insertion.

Cependant, dans les présentations suivantes de RGA [22, 83], les auteurs utilisent des horloges de Lamport [40] en lieu et place des identifiants S4Vector. Nous procédons donc ici à la même simplification, et abstrayons la structure des identifiants utilisée avec le symbole t.

À l'aide des identifiants, RGA redéfinit les modifications de la séquence de la manière suivante :

(i) *ins* devient $ins(predId \prec \langle id, elt \rangle)$.

(ii) *rmv* devient $rmv(id)$.

Puisque plusieurs éléments peuvent être insérés en concurrence à la même position, c.-à-d. avec le même prédecesseur, il est nécessaire de définir une relation d'ordre strict total pour ordonner les éléments de manière déterministe et indépendante de l'ordre de réception des modifications. Pour cela, RGA définit $<_{id}$:

Définition 31 (Relation $<_{id}$)**.** La relation $<_{id}$ définit un ordre strict total sur les identifiants en se basant sur l'ordre lexicographique leurs composants. Par exemple, étant donné deux identifiants $t_1 = \langle ssid_1, sum_1, ssn_1, seq_1 \rangle$ et $t_2 = \langle ssid_2, sum_2, ssn_2, seq_2 \rangle$, nous avons :

$$\begin{aligned} t_1 <_{id} t_2 \quad \textit{iff} \quad & (ssid_1 < ssid_2) \quad \vee \\ & (ssid_1 = ssid_2 \wedge sum_1 < sum_2) \quad \vee \\ & (ssid_1 = ssid_2 \wedge sum_1 = sum_2 \wedge ssn_1 < ssn_2) \quad \vee \\ & (ssid_1 = ssid_2 \wedge sum_1 = sum_2 \wedge ssn_1 = ssn_2 \wedge seq_1 = seq_2) \end{aligned}$$

L'utilisation de $<_{id}$ comme stratégie de résolution de conflits permet de rendre commutative les modifications *ins* concurrentes.

Concernant les suppressions, RGA se comporte de manière similaire à WOOT : la séquence conserve une pierre tombale pour chaque élément supprimé, de façon à pouvoir insérer à la bonne position un élément dont le prédecesseur a été supprimé en concurrence. Cette stratégie rend commutative les modifications *ins* et *rmv*.

Nous récapitulons le fonctionnement de RGA à l'aide de la Figure 2.15.

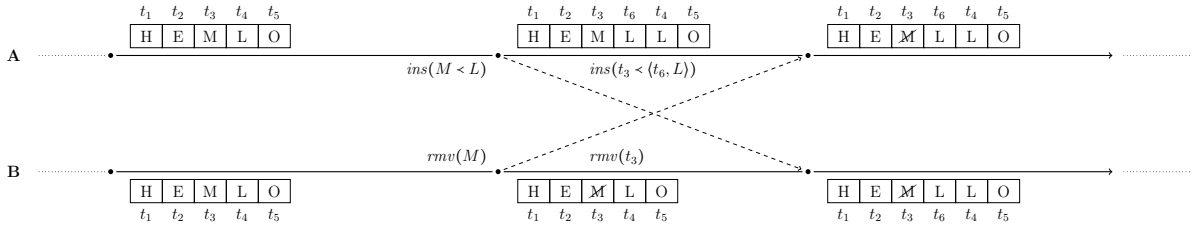


FIGURE 2.15 – Modifications concurrentes d'une séquence répliquée RGA

Dans cet exemple, deux noeuds A et B partagent et éditent collaborativement une séquence répliquée RGA. Initialement, ils possèdent le même état : la séquence contient les éléments "HEMLO", et à chaque élément est associé un identifiant, e.g. t_1, t_2, t_3...

Le noeud A insère l'élément "L" après l'élément et "M", c.-à-d. $ins(M \prec L)$. RGA convertit cette modification en opération $ins(t_3 \prec \langle t_6, L \rangle)$. L'opération est intégrée à la copie locale, ce qui produit l'état "HEMLLO", puis diffusée sur le réseau.

En concurrence, le noeud B supprime l'élément "M" de la séquence, c.-à-d. $rmv(M)$. De la même manière, RGA génère l'opération correspondante $rmv(t_3)$. Comme expliqué précédemment, l'intégration de cette opération ne supprime pas l'élément "M" de l'état mais se contente de le masquer. L'état produit est donc "HE$\not{M}$LO". L'opération est ensuite diffusée.

A (resp. B) reçoit ensuite l'opération de B, $rmv(t_3)$ (resp. A, $ins(t_3 \prec \langle t_6, L \rangle)$), et l'intègre à sa copie. Les opérations de RGA étant commutatives, les noeuds obtiennent le même état final : "HE$\not{M}$LLO".

À la différence des auteurs de WOOT, ROH et al. [75] jugent le coût des pierres tombales trop élévé. Ils proposent alors un mécanisme de GC des pierres tombales. Ce mécanisme repose sur deux conditions :

(i) La stabilité causale de l'opération *rmv*, c.-à-d. l'ensemble des noeuds a intégré la suppression de l'élément et ne peut émettre d'opérations utilisant l'élément supprimé comme prédecesseur.

(ii) L'impossibilité pour l'ensemble des noeuds de générer un identifiant inférieur à celui de l'élément suivant la pierre tombale d'après $<_{id}$.

L'intuition de la condition (i) est de s'assurer qu'aucune opération *ins* concurrente à l'exécution du mécanisme ne peut utiliser la pierre tombale comme prédecesseur, les opérations *ins* ne pouvant reposer que sur les éléments. L'intuition de la condition (ii) est de s'assurer que l'intégration d'une opération *ins*, concurrente à l'exécution du mécanisme et devant résulter en l'insertion de l'élément avant la pierre tombale, ne sera altérée par la suppression de cette dernière.

Concernant le modèle de livraison adopté, RGA repose sur une livraison causale des opérations. Cependant, [75] indique que ce modèle de livraison pourrait être relaxé, de façon à ne plus dépendre de vecteurs d'horloges. Ce point est néanmoins laissé comme piste de recherche future. À notre connaissance, cette dernière n'a pas été explorée dans la littérature. Néanmoins Elvinger [29] indique que RGA pourrait adopter un modèle de livraison similaire à celui de WOOT. Ce modèle consisterait :

Définition 32 (Modèle livraison RGA)**.** Le modèle de livraison RGA définit que :

(i) Une opération doit être livrée exactement une fois à chaque noeud.

(ii) Une opération $ins(predId \prec \langle id, elt \rangle)$ ne peut être livrée à un noeud qu'après la livraison de l'opération d'insertion de l'élément associé à $predId$.

(iii) Une opération $rmv(id)$ ne peut être livrée à un noeud qu'après la livraison de l'opération d'insertion de l'élément associé à id.

Nous secondons cette observation.

Un des avantages de RGA est son efficacité. En effet, son algorithme d'intégration des insertions offre une meilleure complexité en temps que celui de WOOT : $\mathcal{O}(H)$, avec H le nombre de modifications ayant été effectuées sur le document [79]. De plus, [83, 84] montrent que le modèle de données de RGA est optimal d'un point de vue complexité en espace comme CRDT pour Séquence par élément sans mécanisme de GC. RGA est ainsi utilisé dans plusieurs implémentations [85].

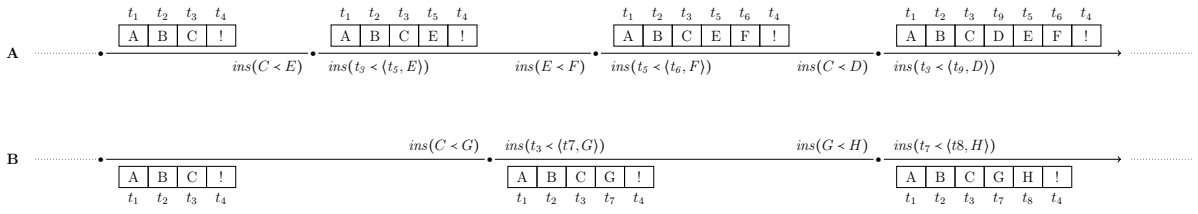
Plusieurs extensions de RGA ont par la suite été proposées. Briot et al. [86] indiquent que les pauvres performances des modifications locales [12] des CRDTs pour Séquence constituent une de leurs limites. Il s'agit en effet des performances impactant le plus l'expérience utilisateur, celleux-ci s'attendant à un retour immédiat de la part de l'application. Les auteurs souhaitent donc réduire la complexité en temps des modifications locales à une complexité logarithmique.

Pour cela, ils proposent l'*identifier structure*, une structure de données auxiliaire utilisable par les CRDTs pour Séquence. Cette structure permet de retrouver plus efficacement l'identifiant d'un élément à partir de son index, au pris d'un surcoût en métadonnées. Les auteurs combinent cette structure de données à un mécanisme d'aggrégation des élements en blocs [13] tels que proposés par [87, 28], qui permet de réduire la quantité de métadonnées stockées par la séquence répliquée. Cette combinaison aboutit à la définition d'un nouveau

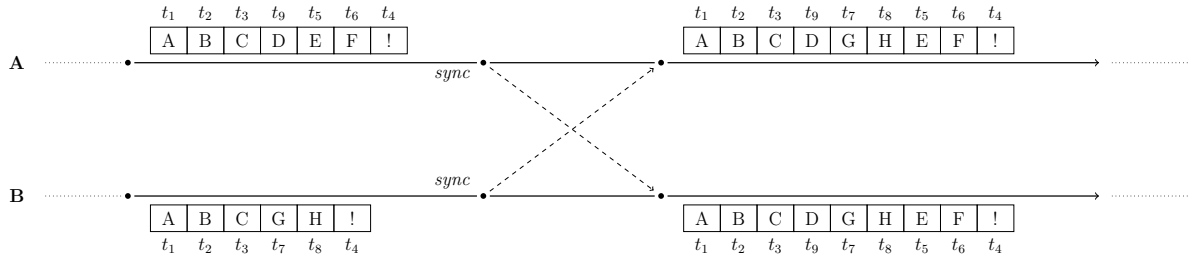
12. Relativement par rapport aux algorithmes de l'approche OT.
13. Nous détaillerons ce mécanisme par la suite.

CRDT pour Séquence, *RGATreeSplit*, qui offre une meilleure complexité en temps et en espace.

Dans [88], les auteurs mettent en lumière un problème récurrent des CRDTs pour Séquence : lorsque des séquences de modifications sont effectuées en concurrence par des noeuds, les CRDTs assurent la convergence des répliques mais pas la correction du résultat. Notamment, il est possible que les éléments insérés en concurrence se retrouvent entrelacés. La Figure 2.16 présente un tel cas de figure :



(a) Modifications concurrentes d'une séquence répliquée RGA



(b) Synchronisation résultant en un entrelacement des modifications concurrentes

FIGURE 2.16 – Entrelacement d'éléments insérés de manière concurrente

Dans la Figure 2.16a, deux noeuds A et B partagent et éditent collaborativement une séquence répliquée RGA. Initialement, ils possèdent le même état : la séquence contient les éléments "ABC!", et à chaque élément est associé un identifiant, e.g. t_1, t_2, t_3 et t_4.

Le noeud A insère après l'élément "C" les éléments "E" et F. RGA génère les opérations $ins(t_3 \prec \langle t_5, E\rangle)$ et $ins(t_5 \prec \langle t_6, F\rangle)$. En concurrence, le noeud B insère les éléments "G" et "H" de manière similaire, produisant les opérations $ins(t_3 \prec \langle t_7, G\rangle)$ et $ins(t_7 \prec \langle t_8, H\rangle)$. Finalement, toujours en concurrence, le noeud A insère un nouvel élément après l'élément "C", l'élément "D", ce qui résulte en l'opération $ins(t_9 \prec \langle t_3, D\rangle)$. Pour la suite de notre exemple, nous supposons que $t_5 <_{id} t_6 <_{id} t_7 <_{id} t_8 <_{id} t_9$.

Nous poursuivons notre exemple dans la Figure 2.16b. Dans cette figure, les noeuds A et B se synchronisent et échangent leurs opérations respectives. À la réception de l'opération de B $ins(t_3 \prec \langle t_7, G\rangle)$, le noeud A compare t_7 avec les identifiants des éléments se trouvant après t_3. Il place l'élément "G" qu'après les éléments ayant des identifiants supérieurs à t_7. Ainsi, il insère "G" après "D" (t_9), mais avant "E" (t_5). L'élément "H" (t_7) est inséré de manière similaire avant "E" (t_5).

Le noeud B procède de manière similaire. Les noeuds A et B convergent alors à un état équivalent : "ABCDGHEF!". Nous remarquons ainsi que les modifications de B, la chaîne "GH", s'est intercalée dans la chaîne insérée par A en concurrence, "DHEF".

Pour remédier à ce problème, les auteurs définissent une nouvelle spécification que

doivent respecter les approches pour la mise en place de séquences répliquées : *la spécification forte sans entrelacement des séquences répliquées*. Basée sur la spécification forte des séquences répliquées spécifiée dans [83, 84], cette nouvelle spécification précise que les éléments insérés en concurrence ne doivent pas s'entrelacer dans l'état final. Kleppmann et al. [88] proposent ensuite une évolution de RGA respectant cette spécification.

Pour cela, les auteurs ajoutent à l'opération *ins* un paramètre, *samePredIds*, un ensemble correspondant à l'ensemble des identifiants connus utilisant le même *predId* que l'élément inséré. En maintenant en plus un exemplaire de cet ensemble pour chaque élément de la séquence, il est possible de déterminer si deux opérations *ins* sont concurrentes ou causalement liées et ainsi déterminer comment ordonner leurs éléments. Cependant, les auteurs ne prouvent pas dans [88] que cette extension empêche tout entrelacement [14].

2.3.2 Approche à identifiants densément ordonnés

Treedoc

[21, 76] proposent une nouvelle approche pour CRDTs pour Séquence. La particuliarité de cette approche est de se baser sur des identifiants de position, respectant un ensemble de propriétés :

Définition 33 (Propriétés des identifiants de position)**.** Les propriétés que les identifiants de position doivent respecter sont les suivantes :

(i) Chaque identifiant est attribué à un élément de la séquence.

(ii) Aucune paire d'éléments ne partage le même identifiant.

(iii) L'identifiant d'un élément est immuable.

(iv) Il existe un ordre total strict sur les identifiants, $<_{id}$, cohérent avec l'ordre des éléments dans la séquence.

(v) Les identifiants sont tirés d'un ensemble dense, que nous notons $\mathbb{I}$.

Intéressons-nous un instant à la propriété (v). Cette propriété signifie que :

$$\forall predId, succId \in \mathbb{I}, \exists id \in \mathbb{I} \mid predId <_{id} id <_{id} succId$$

Cette propriété garantit donc qu'il sera toujours possible de générer un nouvel identifiant de position entre deux autres, c.-à-d. qu'il sera toujours possible d'insérer un nouvel élément entre deux autres (d'après la propriété (iv)).

L'utilisation d'identifiants de position permet de redéfinir les modifications de la séquence :

(i) $ins(pred \prec elt \prec succ)$ devient alors $ins(id, elt)$, avec $predId <_{id} id <_{id} succId$.

(ii) $rmv(elt)$ devient $rmv(id)$.

14. Un travail en cours [89] indique en effet qu'une séquence répliquée empêchant tout entrelacement est impossible.

Ces redéfinitions permettent de proposer une spécification de la séquence avec des modifications commutatives.

À partir de cette spécification, Preguica et al. propose un CRDT pour Séquence : *Treedoc* Ce dernier tire son nom de l'approche utilisée pour émuler un ensemble dense pour générer les identifiants de position : Treedoc utilise pour cela les chemins d'un arbre binaire.

La Figure 2.17 illustre le fonctionnement de cette approche. La racine de l'arbre binaire,

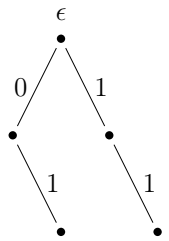


Figure 2.17 – Identifiants de positions

notée ϵ, correspond à l'identifiant de position du premier élément inséré dans la séquence répliquée. Pour générer les identifiants des éléments suivants, Treedoc utilise l'identifiant de leur prédecesseur ou successeur : Treedoc concatène (noté $\oplus$) à ce dernier le chiffre 0 (resp. 1) en fonction de si l'élément doit être placé à gauche (resp. à droite) de l'identifiant utilisé comme base. Par exemple, pour insérer un nouvel élément à la fin de la séquence dont les identifiants de position sont représentés par la Figure 2.17, Treedoc lui associerait l'identifiant $id = \epsilon \oplus 1 \oplus 1 \oplus 1$. Ainsi, Treedoc suit l'ordre du parcours infixe de l'arbre binaire pour ordonner les identifiants de position.

Ce mécanisme souffre néanmoins d'un écueil : en l'état, plusieurs noeuds du système peuvent associer un même identifiant à des éléments insérés en concurrence, contravenant alors à la propriété (ii). Pour corriger cela, Treedoc ajoute à chaque noeud de l'arbre un désambiguateur par élément : un *Dot* (cf. Définition 25). Nous représentons ces derniers avec la notation d_i.

Ainsi, un noeud de l'arbre des identifiants peut correspondre à plusieurs éléments, ayant tous le même identifiant à l'exception de leur désambiguateur. Ces éléments sont alors ordonnés les uns par rapport aux autres en respectant l'ordre défini sur leur désambiguateur.

Afin de réduire le surcoût des désambiguateurs, ces derniers ne sont ajoutés au chemin formant un identifiant qu'uniquement lorsqu'ils sont nécessaires, c.-à-d. :

(i) Le noeud courant est le noeud final de l'identifiant.

(ii) Le noeud courant nécessite désambiguation, c.-à-d. plusieurs éléments utilisent l'identifiant correspondant à ce noeud.

La Figure 2.18 présente un exemple de cette situation. Dans cet exemple, deux identifiants furent insérés en concurrence en fin de séquence : $id_4 = \epsilon \oplus \langle 1, d_4 \rangle$ et $id_5 = \epsilon \oplus \langle 1, d_5 \rangle$. Pour développer cet exemple, Treedoc générerait les identifiants :

(i) $id_6 = \epsilon \oplus 1 \oplus \langle 1, d_6 \rangle$ à l'insertion d'un nouvel élément en fin de liste.

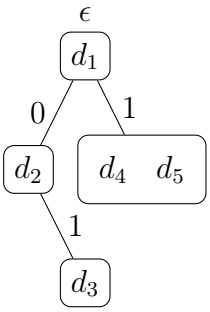


FIGURE 2.18 – Identifiants de position avec désambiguateurs

(ii) $id_7 = \epsilon \oplus \langle 1, d_4 \rangle \oplus \langle 1, d_7 \rangle$ à l'insertion d'un nouvel élément entre les éléments ayant pour identifiants id_4 et id_5.

Nous récapitulons le fonctionnement complet de Treedoc dans la Figure 2.19. Par souci de cohésion, nous utilisons ici à la fois l'arbre binaire pour représenter les identifiants de position des éléments et les éléments eux-mêmes. Nous omettons aussi le chemin vide ϵ dans la représentation des identifiants lorsque non-nécessaire.

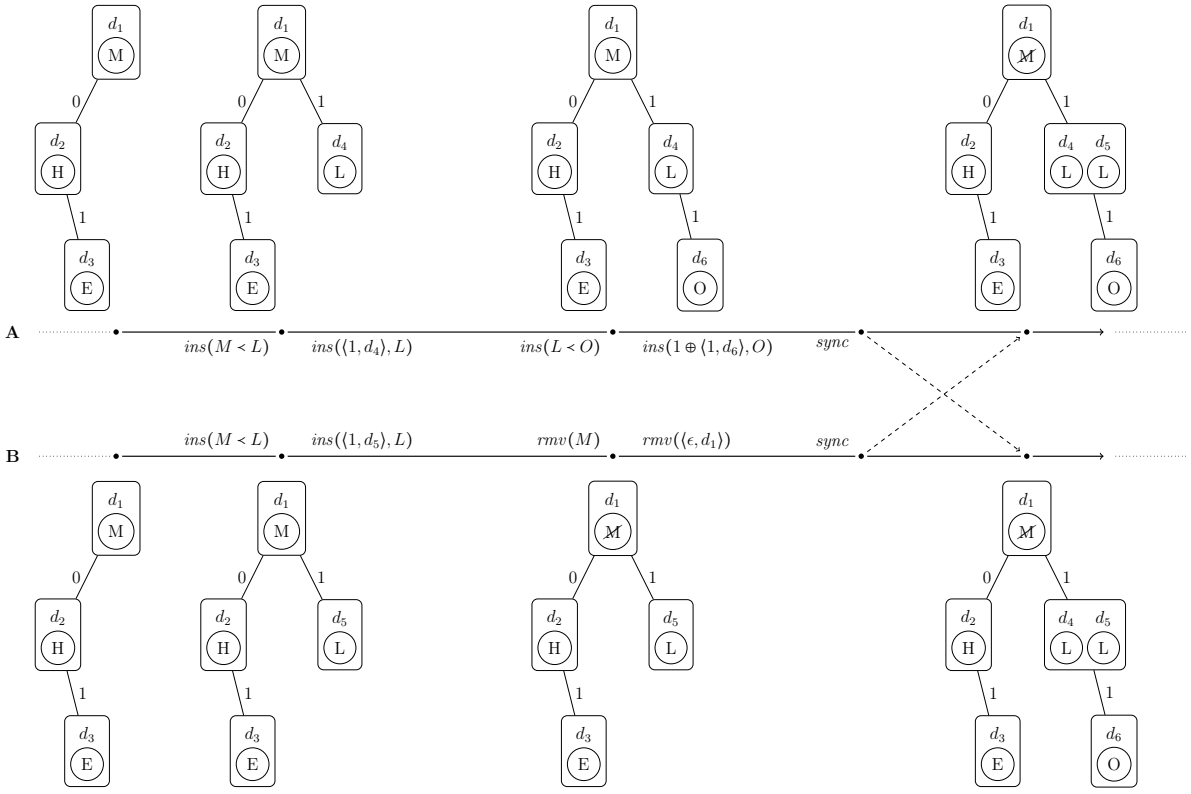


FIGURE 2.19 – Modifications concurrentes d'une séquence répliquée Treedoc

Dans cet exemple, deux noeuds A et B partagent et éditent collaborativement une séquence répliquée Treedoc. Initialement, ils possèdent le même état : la séquence contient les éléments "HEM".

Le noeud A insère l'élément "L" en fin de séquence, c.-à-d. $ins(M \prec L)$. Treedoc génère l'opération correspondante, $ins(\langle 1, d_4\rangle, L)$, et l'intègre à sa copie locale. Puis A insère l'élément "O", toujours en fin de séquence. La modification $ins(L \prec O)$ est convertie en opération $ins(1 \oplus \langle 1, d_6\rangle, O)$ et intégrée.

En concurrence, le noeud B insère aussi un élément "L" en fin de séquence. Cette modification résulte en l'opération $ins(\langle 1, d_5\rangle, L)$, qui est intégrée. Le noeud B supprime ensuite l'élément "M" de la séquence, ce qui produit l'opération $rmv(\langle \epsilon, d_1\rangle)$. Cette dernière est intégrée à sa copie locale. Notons ici que le noeud de l'arbre des identifiants n'est pas supprimé suite à cette opération : l'élément associé est supprimé mais le noeud est conservé et devient une pierre tombale. Nous détaillons ci-après le fonctionnement des pierres tombales dans Treedoc.

Les deux noeuds procèdent ensuite à une synchronisation, échangeant leurs opérations respectives. Lorsque A (resp. B) intègre $ins(\langle 1, d_5\rangle, L)$ (resp. $ins(\langle 1, d_4\rangle, L)$), il ajoute cet élément avec son désambiguateur dans noeud de chemin 1, après (resp. avant) l'élément existant (on considère que $d_4 < d_5$).

B intègre ensuite $ins(1 \oplus \langle 1, d_6\rangle, O)$. Il existe cependant une ambiguité sur la position de "O" : cet élément doit-il être placé après l'élément "L" ayant pour identifiant $\langle 1, d_4\rangle$, ou l'élément "L" ayant pour identifiant $\langle 1, d_5\rangle$? Treedoc résout de manière déterministe cette ambiguité en insérant l'élément en tant qu'enfant droit du noeud 1 et de ses éléments. Ainsi, les noeuds A et B convergent à l'état "HELLO".

Intéressons-nous dorénavant au modèle de livraison requis par Treedoc. Dans [76], les auteurs indiquent reposer sur le modèle de livraison causal. En pratique, nous pouvons néanmoins relaxer le modèle de livraison comme expliqué dans [29] :

Définition 34 (Modèle livraison Treedoc)**.** Le modèle de livraison Treedoc définit que :

(i) Une opération doit être livrée exactement une fois à chaque noeud.

(ii) Les opérations *ins* peuvent être livrées dans un ordre quelconque.

(iii) L'opération $rmv(id)$ ne peut être livrée qu'après la livraison de l'opération d'insertion de l'élément associé à id.

Treedoc souffre néanmoins de plusieurs limites. Tout d'abord, le mécanisme d'identifiants de positions proposé est couplé à la structure d'arbre binaire. Cependant, les utilisateur-rices ont tendance à écrire de manière séquentielle, c.-à-d. dans le sens d'écriture de la langue utilisée. Les nouveaux identifiants forment donc généralement une liste chaînée, qui déséquilibre l'arbre.

Ensuite, comme illustré dans la Figure 2.19, Treedoc doit conserver un noeud de l'arbre des identifiants malgré sa suppression lorsque ce dernier possède des enfants. Ce noeud de l'arbre devient alors une pierre tombale. Comparé à l'approche à pierres tombales, Treedoc a pour avantage que son mécanisme de GC ne repose pas sur la stabilité causale d'opérations. En effet, Treedoc peut supprimer définitivement un noeud de l'arbre binaire des identifiants dès lors que celui-ci est une pierre tombale et une feuille de l'arbre. Ainsi, Treedoc ne nécessite pas de coordination asynchrone avec l'ensemble des noeuds du système pour purger les pierres tombales. Néanmoins, l'évaluation de [76] a montré que les pierres tombales pouvait représenter jusqu'à 95% des noeuds de l'arbre.

Finalement, Treedoc souffre du problème de l'entrelacement d'éléments insérés de manière concurrente, contrairement à ce qui est conjecturé dans [88]. En effet, nous présentons un contre-exemple correspondant dans l'Annexe A.

Logoot

En parallèle à Treedoc [76], Weiss et al. [72] proposent Logoot. Ce nouvel CRDT pour Séquence repose sur idée similaire à celle de Treedoc : il associe un identifiant de position, provenant d'un espace dense, à chaque élément de la séquence. Ainsi, ces identifiants ont les mêmes propriétés que celles décrites dans Définition 33.

Les identifiants de position utilisés par Logoot sont spécifiés de manière différente dans [72] et [90]. Dans ce manuscrit, nous nous basons sur la spécification de [90] :

Définition 35 (Identifiant Logoot)**.** Un identifiant Logoot est une liste de tuples Logoot. Les tuples Logoot sont définis de la manière suivante :

Définition 35.1 (Tuple Logoot)**.** Un tuple Logoot est un triplet $\langle pos, nodeId, seq\rangle$ avec

(i) *pos*, un entier représentant la position relative du tuple dans l'espace dense,

(ii) *nodeId*, l'identifiant du noeud auteur de l'élément,

(iii) *seq*, le numéro de séquence courant du noeud auteur de l'élément.

Dans le cadre de cette section, nous nous basons sur cette dernière spécification. Nous utiliserons la notation suivante $pos^{nodeId\ seq}$ pour représenter un tuple Logoot. Sans perdre en généralité, nous utiliserons des lettres minuscules comme valeurs pour *pos*, des lettres majuscules pour *nodeId* et des entiers naturels pour *seq*. Par exemple, l'identifiant $\langle\langle i, A, 1\rangle\langle f, B, 1\rangle\rangle$ est représenté par $i^{A1}f^{B1}$.

Logoot définit un ordre strict total $<_{id}$ sur les identifiants de position. Cet ordre lui permet de les ordonner relativement les uns aux autres, et ainsi ordonner les éléments associés. Pour définir $<_{id}$, Logoot se base sur l'ordre lexicographique.

Définition 36 (Relation $<_{id}$)**.** Étant donné deux identifiants $id = t_1 \oplus t_2 \oplus ... \oplus t_n$ et $id' = t'_1 \oplus t'_2 \oplus ... \oplus t'_m$, nous avons :

$$id <_{id} id' \quad \textit{iff} \quad \begin{array}{l} (n < m \wedge \forall i \in [1, n] \cdot t_i = t'_i) \quad \vee \\ (\exists j \leq m \cdot \forall i < j \cdot t_i = t'_i \wedge t_j <_t t'_j) \end{array}$$

avec $<_t$ défini de la manière suivante :

Définition 36.1 (Relation $<_t$)**.** Étant donné deux tuples $t = \langle pos, nodeId, seq\rangle$ et $t' = \langle pos', nodeId', seq'\rangle$, nous avons :

$$t <_t t' \quad \textit{iff} \quad \begin{array}{l} (pos < pos') \quad \vee \\ (pos = pos' \wedge nodeId < nodeId') \quad \vee \\ (pos = pos' \wedge nodeId = nodeId' \wedge seq < seq') \end{array}$$

Logoot spécifie une fonction `generateId`. Cette fonction permet de générer un nouvel identifiant de position, *id*, entre deux identifiants donnés, *predId* et *succId*, tel que $predId \prec id \prec succId$. Plusieurs algorithmes peuvent être utilisés pour cela. Notamment, [72] présente un algorithme permettant de générer N identifiants de manière aléatoire entre des identifiants *predId* et *succId*, mais reposant sur une représentation efficace des tuples en mémoire. Par souci de simplicité, nous présentons dans Algorithme 1 un algorithme naïf pour `generateId`.

Algorithme 1 Algorithme de génération d'un nouvel identifiant

```
 1: function GENERATEID(predId ∈ 𝕀, succId ∈ 𝕀, nodeId ∈ ℕ, seq ∈ ℕ*)
                                          ▷ precondition : predId <_id succId
 2:     if succId = predId ⊕ ⟨pos_j, nodeId_j, seq_j⟩ ⊕ ... then
                                          ▷ predId is a prefix of succId
 3:         pos ← random ∈ ]⊥_ℕ, pos_j[
 4:         id ← predId ⊕ ⟨pos, nodeId, seq⟩
 5:     else if predId = common ⊕ ⟨pos_i, nodeId_i, seq_i⟩ ⊕ ...∧
                 succId = common ⊕ ⟨pos_j, nodeId_j, seq_j⟩ ⊕ ...∧
                 pos_j − pos_i ≤ 1                                then
                                          ▷ Not enough space between predId and succId
                                          ▷ to insert new id with same length
                                          ▷ common may be empty
 6:         pos ← random ∈ ]pos_{i+1}, ⊤_ℕ]
 7:         id ← common ⊕ ⟨pos_i, nodeId_i, seq_i⟩ ⊕ ⟨pos, nodeId, seq⟩
 8:     else
                                          ▷ predId = common ⊕ ⟨pos_i, nodeId_i, seq_i⟩ ⊕ ...∧
                                          ▷ succId = common ⊕ ⟨pos_j, nodeId_j, seq_j⟩ ⊕ ...∧
                                          ▷ pos_j − pos_i > 1
                                          ▷ common may be empty
 9:         pos ← random ∈ ]pos_i, pos_j[
10:         id ← common ⊕ ⟨pos, nodeId, seq⟩
11:     end if
12:     return id                        ▷ postcondition : predId <_id id <_id succId
13: end function
```

Pour illustrer cet algorithme, considérons son exécution avec :

(i) $predId = e^{A1}$, $nextId = m^{B1}$, $nodeId = C$ et $seq = 1$. `generateId` commence par déterminer où fini le préfixe commun entre les deux identifiants. Dans cet exemple, *predId* et *succId* n'ont aucun préfixe commun, c.-à-d. $common = \varnothing$. `generateId` compare donc les valeurs de *pos* de leur premier tuple respectifs, c.-à-d. e et m, pour déterminer si un nouvel identifiant de taille 1 peut être inséré dans cet intervalle. S'agissant du cas ici, `generateId` choisit une valeur aléatoire dans $]e, m[$, e.g. l, et renvoie un identifiant composé de cette valeur pour *pos* et avec les valeurs de *nodeId* et *seq*, c.-à-d. $id = l^{C1}$ (lignes 8-10).

(ii) $predId = i^{A1}f^{A2}$, $succId = i^{A1}g^{B1}$, $nodeId = C$ et $seq = 1$. De manière similaire à précédemment, `generateId` détermine le préfixe commun entre *predId* et *succId*. Ici, $common = i^{A1}$. `generateId` compare ensuite les valeurs de *pos* de leur second tuple respectifs, c.-à-d. f et g, pour déterminer si un nouvel identifiant de taille 2 peut être inséré dans cet intervalle. Ce n'est point le cas ici, `generateId` doit donc

recopier le second tuple de *predId* pour former *id* et y concaténer un nouveau tuple. Pour générer ce nouveau tuple, `generateId` choisit une valeur aléatoire entre la valeur de *pos* du troisième tuple de *predId* et la valeur maximale notée $\top_{\mathbb{N}}$. *predId* n'ayant pas de troisième tuple, `generateId` utilise la valeur minimale pour *pos*, $\bot_{\mathbb{N}}$. `generateId` choisit donc une valeur aléatoire dans $]\bot_{\mathbb{N}}, \top_{\mathbb{N}}]$ [15], e.g. m, et renvoie un identifiant composé du préfixe commun, du tuple suivant de *predId* et d'un tuple formé à partir de cette valeur pour *pos* et avec les valeurs de *nodeId* et *seq*, c.-à-d. $id = i^{A1} f^{A2} m^{C1}$ (lignes 5-7).

Comme pour Treedoc, l'utilisation d'identifiants de position permet de redéfinir les modifications :

(i) $ins(pred \prec elt \prec succ)$ devient alors $ins(id, elt)$, avec $predId <_{id} id <_{id} succId$.

(ii) $rmv(elt)$ devient $rmv(id)$.

Les auteurs proposent ainsi une séquence répliquée avec des opérations commutatives.

Nous illustrons cela à l'aide de la Figure 2.20.

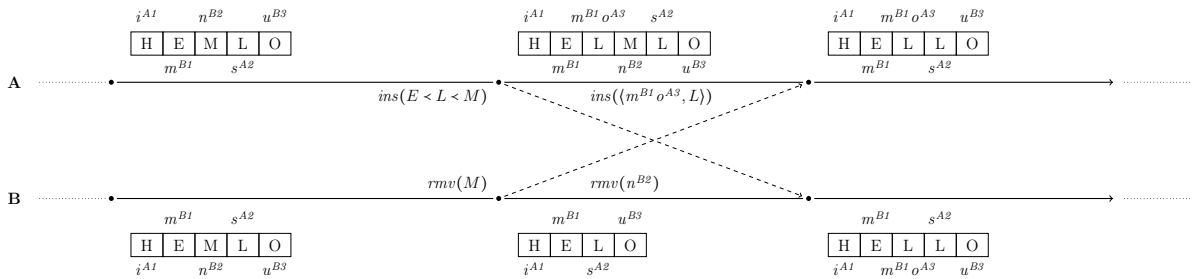


FIGURE 2.20 – Modifications concurrentes d'une séquence répliquée Logoot

Dans cet exemple, deux noeuds A et B partagent et éditent collaborativement une séquence répliquée Logoot. Les deux noeuds possèdent le même état initial : une séquence contenant les éléments "HEMLO", avec leur identifiants respectifs.

Le noeud A insère l'élément "L" entre les éléments "E" et "M", c.-à-d. $ins(E \prec L \prec M)$. Logoot doit alors associer à cet élément un identifiant *id* tel que $m^{B1} \prec id \prec n^{B2}$. Dans cet exemple, Logoot choisit l'identifiant $m^{B1}\,o^{A3}$. L'opération correspondante à l'insertion, $ins(m^{B1}\,o^{A3}, L)$, est générée, intégrée à la copie locale et diffusée.

En concurrence, le noeud B supprime l'élément "M" de la séquence. Logoot retrouve l'identifiant de cet élément, n^{B2} et produit l'opération $rmv(n^{B2})$. Cette dernière est intégrée à sa copie locale et diffusée.

À la réception de l'opération $rmv(n^{B2})$, le noeud A parcourt sa copie locale. Il identifie l'élément possèdant cet identifiant, "M", et le supprime de sa séquence. De son côté, le noeud B reçoit l'opération $ins(m^{B1}\,o^{A3}, L)$. Il parcourt sa copie locale jusqu'à trouver un identifiant supérieur à celui de l'opération : s^{B2}. Il insère alors l'élément reçu avant ce dernier. Les noeuds convergent alors à l'état "HELLO".

15. Il est important d'exclure $\bot_{\mathbb{N}}$ des valeurs possibles pour *pos* du dernier tuple d'un identifiant *id* afin de garantir que l'espace reste dense, notamment pour garantir qu'un noeud sera toujours en mesure de générer un nouvel identifiant id' tel que $id' <_{id} id$.

Concernant le modèle de livraison de Logoot, [72] indique se reposer sur le modèle de livraison causal. Nous constatons cependant que nous pouvons proposer un modèle de livraison moins contraint :

Définition 37 (Modèle livraison Logoot)**.** Le modèle de livraison Logoot définit que :

(i) Une opération doit être livrée exactement une fois à chaque noeud.

(ii) Les opérations *ins* peuvent être livrées dans un ordre quelconque.

(iii) L'opération $rmv(id)$ ne peut être livrée qu'après la livraison de l'opération d'insertion de l'élément associé à id.

Ainsi, Logoot peut adopter le même modèle de livraison que Treedoc, comme indiqué dans [29].

En contrepartie, Logoot souffre d'un problème de croissance de la taille des identifiants. Comme mis en lumière dans la Figure 2.20, Logoot génère des identifiants composés de plus en plus de tuples au fur et à mesure que l'espace des identifiants pour une taille donnée se sature. La croissance des identifiants a cependant plusieurs impacts négatifs :

(i) Les identifiants sont stockés au sein de la séquence répliquée. Leur croissance augmente donc le surcoût en métadonnées du CRDT.

(ii) Les identifiants sont diffusés sur le réseau par le biais des opérations. Leur croissance augmente donc le surcoût en bande-passante du CRDT.

(iii) Les identifiants sont comparés entre eux lors de l'intégration des opérations. Leur croissance augmente donc le surcoût en calculs du CRDT.

Un objectif de l'algorithme `generateId` est donc de limiter le plus possible la vitesse de croissance des identifiants.

Plusieurs extensions furent proposées pour Logoot. Weiss et al. [90] proposent une nouvelle stratégie d'allocation des identifiants pour `generateId`. Cette stratégie consiste à limiter la distance entre deux identifiants insérés au cours de la même modification *ins*, au lieu des les répartir de manière aléatoire entre *predId* et *succId*. Ceci permet de regrouper les identifiants des éléments insérés par une même modification et de laisser plus d'espace pour les insertions suivantes. Les expérimentations présentées montrent que cette stratégie permet de ralentir la croissance des identifiants en fonction du nombre d'insertions. Ce résultat est confirmé par la suite dans [79]. Ainsi, en réduisant la vitesse de croissance des identifiants, ce nouvel algorithme permet de réduire le surcoût en métadonnées, calculs et bande-passante du CRDT.

Toujours dans [90], les auteurs introduisent *Logoot-Undo*, une version de Logoot dotée d'un mécanisme d'undo. Ce mécanisme prend la forme d'une nouvelle modification, *undo*, qui permet d'annuler l'effet d'une ou plusieurs modifications passées. Cette modification, et l'opération en résultant, est spécifiée de manière à être commutative avec toutes autres opérations concurrentes, c.-à-d. *ins*, *rmv* et *undo* elle-même.

Pour définir *undo*, une notion de *degrée de visibilité* d'un élément est introduite. Elle permet à Logoot-Undo de déterminer si l'élément doit être affiché ou non. Pour cela, Logoot-Undo maintient une structure auxiliaire, le *Cimetière*, qui référence les identifiants

des éléments dont le degrée est inférieur à 0 [16]. Ainsi, Logoot-Undo ne référence qu'un nombre réduit de pierres tombales. Qui plus est, ces pierres tombales sont stockées en dehors de la structure représentant la séquence et n'impactent donc pas les performances des modifications ultérieures.

De plus, il convient de noter que l'ajout du degrée de visibilité des éléments permet de rendre commutatives l'opération *ins* avec l'opération *rmv* d'un même élément. Ainsi, Logoot-Undo ne nécessite pour son modèle de livraison qu'une *livraison en exactement un exemplaire à chaque noeud.*

Finalement, André et al. [28] introduisent *LogootSplit*. Reprenant les idées introduites par [87], ce travail présente un mécanisme d'aggrégation dynamiques des élements en blocs. Ceci permet de réduire la granularité des éléments stockés dans la séquence, et ainsi de réduire le surcoût en métadonnées, calculs et bande-passante du CRDT. Nous utilisons ce CRDT pour séquence comme base pour les travaux présentés dans ce manuscrit. Nous dédions donc la section 2.4 à sa présentation en détails.

Matthieu: TODO : Autres Sequence CRDTs à considérer : String-wise CRDT [87], Chronofold [91]

2.3.3 Synthèse

Depuis l'introduction des CRDTs, deux approches différentes pour la résolution de conflits ont été proposées pour le type Séquence : l'*approche basée sur des pierres tombales* et l'*approche basée à identifiants densément ordonnés*. Chacune de ces approches visent à permettre l'édition concurrente tout en minimisant le surcoût du type répliqué, que ce soit d'un point de vue métadonnées, calculs et bande-passante. Au fil des années, chacune de ces approches a été raffinée avec de nouveaux CRDTs de plus en plus en efficaces.

Cependant, une faiblesse de la littérature est à notre sens le couplage entre mécanismes de résolution de conflits et choix d'implémentations : plusieurs travaux [76, 72, 28, 86] ne séparent pas l'approche proposée pour rendre les modifications concurrentes commutatives des structures de données et algorithmes choisis pour représenter et manipuler la séquence et les identifiants, e.g. tableau dynamique, liste chaînée, liste chaînée + table de hachage + arbre binaire de recherche... *Matthieu: TODO : Revoir refs utilisées ici* Il en découle que les évaluations proposées par la communautée comparent au final des efforts d'implémentations plutôt que les approches elles-mêmes. En conséquence, la littérature ne permet pas d'établir la supériorité d'une approche sur l'autre.

Nous conjecturons que le surcoût des pierres tombales et le surcoût des identifiants densément ordonnés ne sont que les facettes d'une même pièce, c.-à-d. le surcoût inhérent à un mécanisme de résolution de conflits pour séquence répliquée. Ce surcoût s'exprime sous la forme de compromis différents selon l'approche choisie. Nous proposons donc une comparaison de ces approches se focalisant sur leurs différences pour indiquer plus clairement le compromis que chacune d'entre elle propose.

La principale différence entre les deux approches porte sur les identifiants. Chaque approche repose sur des identifiants attachés aux éléments, mais leurs rôles et utilisations

16. Nous pouvons dès lors inférer le degrée des identifiants restants en fonction de s'ils se trouvent dans la séquence (1) ou s'ils sont absents à la fois de la séquence et du cimetière (0).

diffèrent :

(i) Dans l'approche à pierres tombales, les identifiants servent à référencer de manière unique et immuable les éléments, c.-à-d. de manière indépendante de leur index courant. Ils sont aussi utilisés pour ordonner les éléments insérés de manière concurrente à une même position.

(ii) Dans l'approche à identifiants densément ordonnés, les identifiants incarnent les positions uniques et immuables des éléments dans un espace dense, avec l'ordre entre les positions des éléments dans cet espace qui correspond avec l'intention des insertions effectuées.

Ainsi, les contraintes qui pèsent sur les identifiants sont différentes. Nous les présentons ci-dessous.

Définition 38 (Propriétés des identifiants dans approche à pierres tombales)**.** Les propriétés que doivent respecter les identifiants dans l'approche à pierres tombales sont les suivantes :

(i) Chaque identifiant est attribué à un élément de la séquence.

(ii) Aucune paire d'éléments ne partage le même identifiant.

(iii) L'identifiant d'un élément est immuable.

(iv) Il existe un ordre total strict sur les identifiants, $<_{id}$, qui permet d'ordonner les éléments insérés en concurrence à une même position.

Définition 39 (Propriétés des identifiants dans approche à identifiants densément ordonnés)**.** Les propriétés que doivent respecter les identifiants dans l'approche à identifiants densément ordonnés sont les suivantes :

(i) Chaque identifiant est attribué à un élément de la séquence.

(ii) Aucune paire d'éléments ne partage le même identifiant.

(iii) L'identifiant d'un élément est immuable.

(iv) Il existe un ordre total strict sur les identifiants, $<_{id}$, qui permet d'ordonner les éléments insérés dans la séquence de manière cohérente avec l'ordre souhaité.

(v) Les identifiants sont tirés d'un ensemble dense.

Les identifiants des deux approches partagent donc les propriétés (i), (ii) et (iii).

Pour respecter les propriétés (i) et (ii), les CRDTs reposent généralement sur des dots (cf. Définition 25, page 34). Ainsi, un couple de taille fixe, $\langle nodeId, seq \rangle$, permet de respecter la contrainte d'unicité des identifiants.

Le rôle des identifiants diffère entre les approches au niveau des propriétés (iv) et (v) : les identifiants dans l'approche à pierres tombales doivent permettre d'ordonner un élément par rapport aux éléments insérés en concurrence uniquement, tandis que ceux de la seconde approche doivent permettre d'ordonner un élément par rapport à l'ensemble des éléments insérés. Cette nuance se traduit dans la structure des identifiants, notamment leur taille.

Pour ordonner un identifiant par rapport à ceux générés en concurrence, l'approche à pierres tombales peut définir une relation d'ordre total strict sur leur dot respectif, e.g.

en se basant sur l'ordre lexicographique. Un élément tiers peut y être ajouté si nécessaire, e.g. RGA et son horloge de Lamport [40]. Ainsi, les identifiants de cette approche peuvent être définis tout en ayant une taille fixe, c.-à-d. un nombre de composants fixe.

D'après (iv), l'approche à identifiants densément ordonnés doit elle définir une relation d'ordre total strict sur l'ensemble de ses identifiants. Il en découle qu'elle doit aussi permettre de générer un nouvel identifiant de position entre deux autres, c.-à-d. la propriété (v). Ainsi, cette propriété requiert de l'ensemble des identifiants d'émuler l'ensemble des réels. La précision étant finie en informatique, la seule approche proposée à notre connaissance pour répondre à ce besoin consiste à permettre à la taille des identifiants de varier et de baser la relation d'ordre $<_{id}$ sur l'ordre lexicographique.

L'augmentation non-bornée de la taille des identifiants se répercute sur plusieurs aspects du surcoût de l'approche à identifiants densément ordonnés :

(i) Les métadonnées attachées par élément, c.-à-d. le surcoût mémoire.

(ii) Les métadonnées transmises par message, les identifiants étant intégrés dans les opérations, c.-à-d. le surcoût en bande-passante.

(iii) Le nombre de comparaisons effectuées lors d'une recherche ou manipulation de la séquence, les identifiants étant comparés pour déterminer où trouver ou placer un élément, c.-à-d. le surcoût en calculs.

En contrepartie, les identifiants densément ordonnés permettent l'intégration chaque élément de manière indépendante des autres. Les identifiants de l'approche à pierres tombales, eux, n'offrent pas cette possibilité puisque la relation d'ordre associée, $<_{id}$, ne correspond pas à l'ordre souhaité des éléments. Pour respecter cet ordre souhaité, l'approche à pierres tombales repose sur l'utilisation du prédecesseur et/ou successeur du nouvel élément inséré. Ce mécanisme implique la nécessité de conserver des pierres tombales dans la séquence, tant qu'elles peuvent être utilisées par une opération encore inconnue, c.-à-d. tant que l'opération de suppression correspondante n'est pas causalement stable.

La présence de pierres tombales dans la séquence impacte aussi plusieurs aspects du surcoût de l'approche à pierres tombales :

(i) Les métadonnées de la séquence ne dépendent pas de son nombre courant d'éléments, mais du nombre d'insertions effectuées, c.-à-d. le surcoût mémoire.

(ii) Le nombre de comparaisons effectuées lors d'une recherche ou manipulation de la séquence, les identifiants des pierres tombales étant aussi comparés lors de la recherche ou insertion d'un élément, c.-à-d. le surcoût en calculs.

Pour compléter notre étude de ces approches, intéressons nous au modèle de livraison requis par ces dernières. Contrairement à ce qui peut être conjecturé après une lecture de la littérature, nous notons qu'aucune de ces approches ne requiert de manière intrinsèque une livraison causale de ses opérations. Ces deux approches sont donc adaptées aux systèmes distribués P2P.

Finalement, nous notons que l'ensemble des CRDTs pour Séquence proposés souffrent du problème de l'entrelacement présenté dans [88]. Nous conjecturons cependant que les CRDTs pour Séquence à pierres tombales sont moins sujets à ce problème. En effet, dans cette approche, l'algorithme d'intégration des nouveaux éléments repose généralement sur l'élément précédent. Ainsi, une séquence d'insertions séquentielles produit une

sous-chaîne d'éléments. L'algorithme d'intégration permet ensuite d'intégrer sans entrelacement de telles sous-chaînes générées en concurrence, e.g. dans le cadre de sessions de travail asynchrones. Cependant, il s'agit d'une garantie offerte par l'approche à pierres tombales dont nous ne retrouvons pas d'équivalent dans l'approche à identifiants densément ordonnés. Pour confirmer notre conjecture et évaluer son impact sur l'expérience utilisateur, il conviendrait de mener un ensemble d'expériences utilisateurs dans la lignée de [79, 92, 93].

Nous récapitulons cette discussion dans le Tableau 2.2.

TABLE 2.2 – Récapitulatif comparatif des différents approches pour CRDTs pour Séquence

	Pierres tombales	Identifiants densément ordonnés
Performances en fct. de la taille de la séq.	✗	✗
Identifiants de taille fixe	✓	✗
Taille des messages fixe	✓	✗
Eléments réellement supprimés de la séq.	✗	✓
Livraison causale non-nécessaire	✓	✓
Sujet à l'entrelacement	✓	✓

Pour la suite de ce manuscrit, nous prenons LogootSplit comme base de travail. Nous détaillons donc son fonctionnement dans la section suivante.

2.4 LogootSplit

LogootSplit [28] est l'état de l'art des séquences répliquées à identifiants densément ordonnés. Son intuition est de proposer un mécanisme permettant d'aggréger de manière dynamique des éléments en blocs d'élements. Ce mécanisme permet de factoriser les métadonnées des éléments aggrégés et de réduire la granularité de la séquence, réduisant ainsi le surcoût en métadonnées, calculs et bande-passante.

2.4.1 Identifiants

Pour ce faire, LogootSplit associe aux éléments des identifiants définis de la manière suivante :

Définition 40 (Identifiant LogootSplit)**.** Un identifiant LogootSplit est une liste de tuples LogootSplit. Les tuples LogootSplit sont définis de la manière suivante :

Définition 40.1 (Tuple LogootSplit)**.** Un tuple LogootSplit est un quadruplet $\langle pos, nodeId, seq, offset \rangle$ avec :

(i) *pos*, un entier représentant la position relative du tuple dans l'espace dense,

(ii) *nodeId*, l'identifiant du noeud auteur de l'élément,

(iii) *seq*, le numéro de séquence courant du noeud auteur de l'élément.

(iv) *offset*, la position de l'élément au sein d'un bloc. Nous reviendrons plus en détails sur ce composant dans la sous-section 2.4.2.

Dans ce manuscrit, nous représentons les tuples LogootSplit par le biais de la notation suivante : $position_{offset}^{nodeId\ nodeSeq}$. Sans perdre en généralité, nous utiliserons des lettres minuscules comme valeurs pour *pos*, des lettres majuscules pour *nodeId* et des entiers naturels pour *seq* et *offset*. Par exemple, nous représentons l'identifiant $\langle\langle i, A, 1, 1\rangle\langle f, B, 1, 1\rangle\rangle$ par $i_1^{A1} f_1^{B1}$.

LogootSplit utilise les identifiants de position pour ordonner relativement les éléments les uns par rapport aux autres. LogootSplit définit une relation d'ordre strict total sur les identifiants : $<_{id}$. Cette relation repose sur l'ordre lexicographique.

Définition 41 (Relation $<_{id}$)**.** Étant donné deux identifiants $id = t_1 \oplus t_2 \oplus ... \oplus t_n$ et $id' = t'_1 \oplus t'_2 \oplus ... \oplus t'_m$, nous avons :

$$id <_{id} id' \quad iff \quad (n < m \wedge \forall i \in [1, n] \cdot t_i = t'_i) \quad \vee \quad (\exists j \leq m \cdot \forall i < j \cdot t_i = t'_i \wedge t_j <_t t'_j)$$

avec $<_t$ défini de la manière suivante :

Définition 41.1 (Relation $<_t$)**.** Étant donné deux tuples $t = \langle pos, nodeId, seq, offset\rangle$ et $t' = \langle pos', nodeId', seq', offset'\rangle$, nous avons :

$$\begin{aligned} t <_t t' \quad iff \quad & (pos < pos') \quad \vee \\ & (pos = pos' \wedge nodeId < nodeId') \quad \vee \\ & (pos = pos' \wedge nodeId = nodeId' \wedge seq < seq') \\ & (pos = pos' \wedge nodeId = nodeId' \wedge seq = seq' \wedge offset = offset') \end{aligned}$$

Par exemple, nous avons :

(i) $i_1^{A1} <_{id} i_1^{B1}$ car le tuple composant le premier est inférieur au tuple composant le second, c.-à-d. $i_0^{A1} <_t i_0^{B0}$.

(ii) $i_0^{B0} <_{id} i_0^{B0} f_0^{A0}$ car le premier est un préfixe du second.

Il est intéressant de noter que le triplet $\langle nodeId, seq, offset\rangle$ du dernier tuple d'un identifiant permet de l'identifier de manière unique.

2.4.2 Aggrégation dynamique d'élements en blocs

Afin de réduire le surcoût de la séquence, LogootSplit propose d'aggréger de façon dynamique les éléments et leur identifiants dans des blocs. Pour cela, LogootSplit introduit la notion d'intervalle d'identifiants :

Définition 42 (Intervalle d'identifiants)**.** Un intervalle d'identifiants est un couple $\langle idBegin, offsetEnd\rangle$ avec :

(i) *idBegin*, l'identifiant du premier élément de l'interval.

(ii) *offsetEnd*, l'offset du dernier identifiant de l'interval.

Les intervalles d'identifiants permettent à LogootSplit d'assigner logiquement un identifiant à un ensemble d'éléments, tout en ne stockant de manière effective 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 intervalle.

Définition 43 (Identifiants contigus)**.** Deux identifiants sont contigus si et seulement si les deux identifiants sont identiques à l'exception de leur dernier offset et que leur derniers offsets sont consécutifs.

De manière plus formelle, étant donné deux identifiants $id = t_1 \oplus t_2 \oplus ... \oplus t_{n-1} \oplus \langle pos, nodeId, seq, offset \rangle$ et $id' = t'_1 \oplus t'_2 \oplus ... \oplus t'_{n-1} \oplus \langle pos', nodeId', seq', offset' \rangle$, nous avons :

$$\begin{aligned} contigus(id, id') \quad iff \quad & (\forall i \in [1, n[\cdot t_i = t'_i) \quad \wedge \\ & (pos = pos' \wedge nodeId = nodeId' \wedge seq = seq') \quad \wedge \\ & (offset + 1 = offset' \vee offset - 1 = offset') \end{aligned}$$

Nous représentons un intervalle d'identifiants à l'aide du formalisme suivant : $position^{nodeId\ nodeSeq}_{begin..end}$ où *begin* est l'offset du premier identifiant de l'intervalle et *end* du dernier.

Matthieu: TODO : Ajouter MàJ de generateId ici. "Pour profiter de cette fonctionnalité, LogootSplit propose une nouvelle fonction generateId. Le principal ajout est un cas supplémentaire favorisant la génération d'un id contigu dans le cas où predId est le dernier élément d'un intervalle d'identifiants de l'auteur. Le booléen isAppendable est nécessaire pour éviter de re-générer un identifiant avec un triplet nodeId,seq,offset déjà utilisé".

LogootSplit utilise une structure de données pour associer un intervalle d'identifiants aux éléments correspondants : les blocs.

Définition 44 (Bloc)**.** Un bloc est un triplet $\langle idInterval, elts, isAppendable \rangle$ avec :

(i) *idInterval*, l'intervalle d'identifiants associés au bloc.

(ii) *elts*, les éléments contenus dans le bloc.

(iii) *isAppendable*, un booléen indiquant si l'auteur du bloc peut ajouter de nouveaux éléments en fin de bloc [17].

Nous représentons un exemple de séquence LogootSplit dans la Figure 2.21. Dans la Figure 2.21a, les identifiants i_1^{B1}, i_2^{B1} et i_3^{B1} forment une chaîne d'identifiants contigus. LogootSplit est donc capable de regrouper ces éléments en un bloc représentant l'intervalle d'identifiants $i_{1..3}^{B1}$ pour minimiser les métadonnées stockées, comme illustré dans la Figure 2.21b.

(a) Éléments avec leur identifiant correspondant (b) Éléments regroupés en un bloc

FIGURE 2.21 – Représentation d'une séquence LogootSplit contenant les éléments "HLO"

Cette fonctionnalité permet d'améliorer les performances de la structure de données sur plusieurs aspects :

17. De manière similaire, il est possible de permettre à l'auteur du bloc d'ajouter de nouveaux éléments en début de bloc à l'aide d'un booléen *isPrependable*. Cette fonctionnalité est cependant incompatible avec le mécanisme que nous proposons dans le **??**. Nous faisons donc le choix de la retirer.

(i) Elle réduit le nombre d'identifiants stockés au sein de la structure de données. En effet, les identifiants sont désormais conservés à l'échelle des blocs plutôt qu'à l'échelle de chaque élément. Ceci permet de réduire le surcoût en métadonnées du CRDT.

(ii) L'utilisation de blocs comme niveau de granularité, en lieu et place des éléments, permet de réduire la complexité en temps des manipulations de la structure de données.

(iii) L'utilisation de blocs permet aussi d'effectuer des modifications à l'échelle de plusieurs éléments, et non plus un par un seulement. Ceci permet de réduire la taille des messages diffusés sur le réseau.

Il est intéressant de noter que la paire $\langle nodeId, seq \rangle$ du dernier tuple d'un identifiant permet d'identifier de manière unique la partie commune des identifiants de l'intervalle d'identifiants auquel il appartient. Ainsi, nous pouvons identifier de manière unique un intervalle d'identifiants avec le quadruplet $\langle nodeId, seq, offsetBegin, offsetEnd \rangle$. Par exemple, l'intervalle d'identifiants $i_2^{B1} f_{2..4}^{A1}$ peut être référencé à l'aide du quadruplet $\langle A, 1, 2, 4 \rangle$.

2.4.3 Modèle de données

En nous basant sur André et al. [28], nous proposons une définition du modèle de données de LogootSplit dans la Figure 2.22 :

payload
$S \in Seq\langle IdInterval, Arr\langle E \rangle, Bool \rangle$

constructor

emp	:		$\longrightarrow S$

prepare

ins	:	$S \times \mathbb{N} \times Arr\langle E \rangle \times \mathbb{I} \times \mathbb{N}^*$	$\longrightarrow Id \times Arr\langle E \rangle$
rmv	:	$S \times \mathbb{N} \times \mathbb{N}$	$\longrightarrow Arr\langle IdInterval \rangle$

effect

ins	:	$S \times Id \times Arr\langle E \rangle$	$\longrightarrow S$
rmv	:	$S \times Arr\langle IdInterval \rangle$	$\longrightarrow S$

queries

len	:	S	$\longrightarrow \mathbb{N}$
rd	:	S	$\longrightarrow Arr\langle E \rangle$

Figure 2.22 – Spécification algébrique du type abstrait LogootSplit

Une séquence LogootSplit est une séquence de blocs. Concernant les modifications définies sur le type, nous nous inspirons de [50] et les séparons en deux étapes :

(i) **prepare**, l'étape qui consiste à générer l'opération correspondant à la modification à partir de l'état courant et de ses éventuels paramètres. Cette étape ne modifie pas l'état.

(ii) **effect**, l'étape qui consiste à intégrer l'effet d'une opération générée précédemment, par le noeud lui-même ou un autre. Cette étape met à jour l'état à partir des données fournies par l'opération.

La séquence LogootSplit autorise deux types de modifications :

(i) $ins(s, i, elts, nodeId, seq)$, qui génère l'opération permettant d'insérer les éléments $elts$ dans la séquence s à l'index i. Cette fonction génère et associe un intervalle d'identifiants aux éléments à insérer en utilisant les valeurs pour $nodeId$ et seq fournies. Elle retourne les données nécessaires pour l'opération ins, c.-à-d. le premier identifiant de l'intervalle d'identifiants alloué et les éléments. Par soucis de simplicité, nous noterons cette modification $ins(pred \prec elts \prec succ)$ et utiliserons l'état courant de la séquence comme valeur pour s, l'identifiant du noeud auteur de la modification comme valeur pour $nodeId$ et le nombre de blocs que le noeud a créé comme valeur pour seq dans nos exemples.

(ii) *rmv(s, i, nbElts)*, qui génère l'opération permettant de supprimer $nbElts$ dans la séquence s à partir de l'index i. Elle retourne les données nécessaires pour l'opération rmv, c.-à-d. les intervalles d'identifiants supprimés. Par soucis de simplicité, nous noterons cette modifications $rmv(elts)$ dans nos exemples.

Nous présentons dans la Figure 2.23 un exemple d'utilisation de cette séquence répliquée.

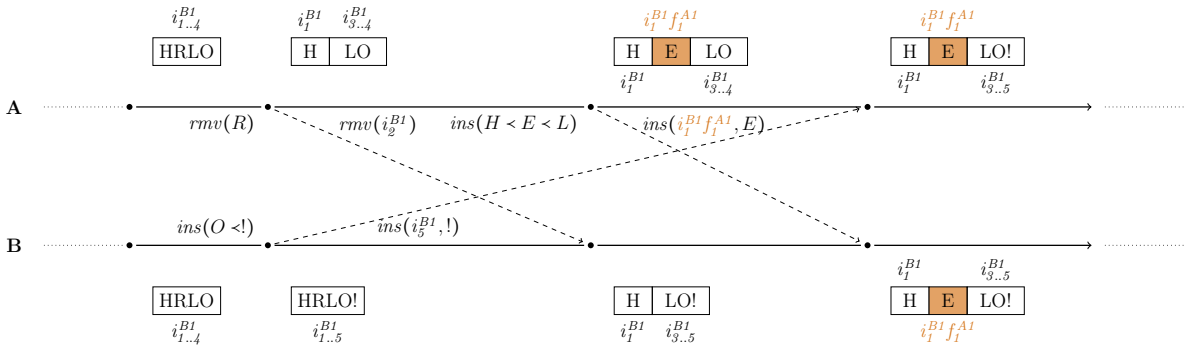


FIGURE 2.23 – Modifications concurrentes d'une séquence répliquée LogootSplit

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 $i_{1..4}^{B1}$ 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 rmv correspondante en utilisant l'identifiant de l'élément supprimé : i_2^{B1}. Cette opération est intégrée à sa copie locale et envoyée au noeud B pour qu'il intègre cette modification à son tour.

Le noeud A insère ensuite l'élément "E" dans la séquence entre les éléments "H" et "L". Le noeud A doit alors générer un identifiant id à associer à ce nouvel élément respectant la contrainte suivante :

$$i_1^{B1} <_{id} id <_{id} i_3^{B1}$$

L'espace des identifiants de taille 1 étant saturé entre ces deux identifiants, A génère id en reprenant le premier tuple de l'identifiant du prédecesseur et en y concaténant un nouveau tuple : $id = i_1^{B1} \oplus f_1^{A1}$. LogootSplit génère l'opération ins correspondante, indiquant l'élément à insérer et sa position grâce à son identifiant. Il intègre cette opération et la diffuse sur le réseau.

En parallèle, le noeud B insère l'élément " !" en fin de la séquence. Comme le noeud B est l'auteur du bloc $i_{1..4}^{B1}$, il peut y ajouter de nouveaux éléments. B associe donc l'identifiant i_5^{B1} à l'élément " !" pour l'ajouter au bloc existant. Il génère l'opération ins correspondante, l'intègre puis la diffuse.

Les noeuds se synchronisent ensuite. Le noeud A reçoit l'opération $ins(i_5^{B1}, L)$. Le noeud A détermine que cet élément doit être inséré à la fin de la séquence, puisque $i_4^{B1} <_{id} i_5^{B1}$. Ces deux identifiants étant contigus, il ajoute cet élément au bloc existant.

De son côté, le noeud B reçoit tout d'abord l'opération $rmv(i_2^{B1})$. Le noeud B supprime donc l'élément correspondant de son état, "R".

Il reçoit ensuite l'opération $ins(i_1^{B1} f_1^{A1}, E)$. Le noeud B insère cet élément entre les éléments "H" et "L", puisqu'on a :

$$i_1^{B1} <_{id} i_1^{B1} f_1^{A1} <_{id} i_3^{B1}$$

L'intention de chaque noeud est donc préservée et les copies convergent.

2.4.4 Modèle de livraison

Afin de garantir son bon fonctionnement, LogootSplit doit être associé à une couche de livraison de messages. Cette couche de livraison doit respecter un modèle de livraison adapté, c.-à-d. offrir des garanties sur l'ordre de livraison des opérations. Dans cette section, nous présentons des exemples d'exécutions en l'absence de modèle de livraison pour illustrer la nécessité de ces différentes garanties.

Livraison des opérations en exactement un exemplaire

Ce premier exemple, représenté par la Figure 2.24, a pour but d'illustrer la nécessité de la propriété de livraison en *exactement un exemplaire* des opérations.

Dans cet exemple, deux noeuds A et B répliquent et éditent collaborativement une séquence. La séquence répliquée contient initialement les éléments "ABCD", qui sont associés à l'intervalle d'identifiants $p_{1..4}^{A1}$.

Le noeud A commence par insérer l'élément "X" dans la séquence entre les éléments "B" et "C". A intègre l'opération résultante, $ins(p_2^{A1} m_1^{A2}, X)$ puis la diffuse au noeud B.

À la réception de l'opération ins, le noeud B l'intègre à son état. Puis il supprime dans la foulée l'élément "X" nouvellement inséré. B intègre l'opération $rmv(p_2^{A1} m_1^{A2})$ puis l'envoie au noeud A.

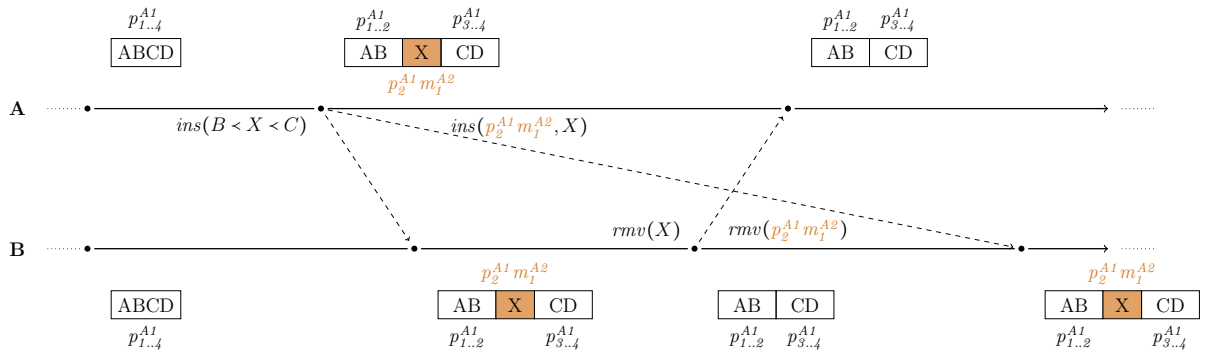


FIGURE 2.24 – Résurgence d'un élément supprimé suite à la relivraison de son opération *ins*

Le noeud A intègre l'opération *rmv*, ce qui a pour effet de supprimer l'élément "X" associé à l'identifiant $p_2^{A1} m_1^{A2}$. Il obtient alors un état équivalent à celui du noeud B.

Cependant, l'opération *ins* insérant l'élément "X" à la position $p_2^{A1} m_1^{A2}$ est de nouveau envoyée au noeud B. De multiples raisons peuvent être à l'origine de ce nouvel envoi : 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 *ins*, ce qui fait revenir l'élément "X" 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 *ins* correspondante.

Livraison de l'opération *rmv* après les opérations *ins* correspondantes

La Figure 2.25 présente un second exemple illustrant la nécessité de la contrainte de livraison d'une opération *rmv* qu'après la livraison des opérations *ins* correspondantes.

Dans cet exemple, trois noeuds A, B et C répliquent et éditent collaborativement une séquence. La séquence répliquée contient initialement les éléments "ABCD", qui sont associés à l'intervalle d'identifiants $p_{1..4}^{A1}$.

Le noeud A commence par insérer l'élément "X" dans la séquence entre les éléments "B" et "C". A intègre l'opération résultante, $ins(p_2^{A1} m_1^{A2}, X)$ puis la diffuse.

À la réception de l'opération *ins*, le noeud B l'intègre à son état. Puis il supprime dans la foulée l'élément "X" nouvellement inséré. B intègre l'opération $rmv(p_2^{A1} m_1^{A2})$ puis la diffuse.

Toutefois, suite à un aléa du réseau, l'opération *rmv* supprimant l'élément "X" est reçue par le noeud C en première. Ainsi, le noeud C intègre cette opération : il parcourt son état à la recherche de l'élément "X" pour le supprimer. Celui-ci n'est pas présent dans son état courant, l'intégration de l'opération s'achève sans effectuer de modification.

Le noeud C reçoit ensuite l'opération *ins*. Le noeud C intègre ce nouvel élément dans la séquence en utilisant son identifiant.

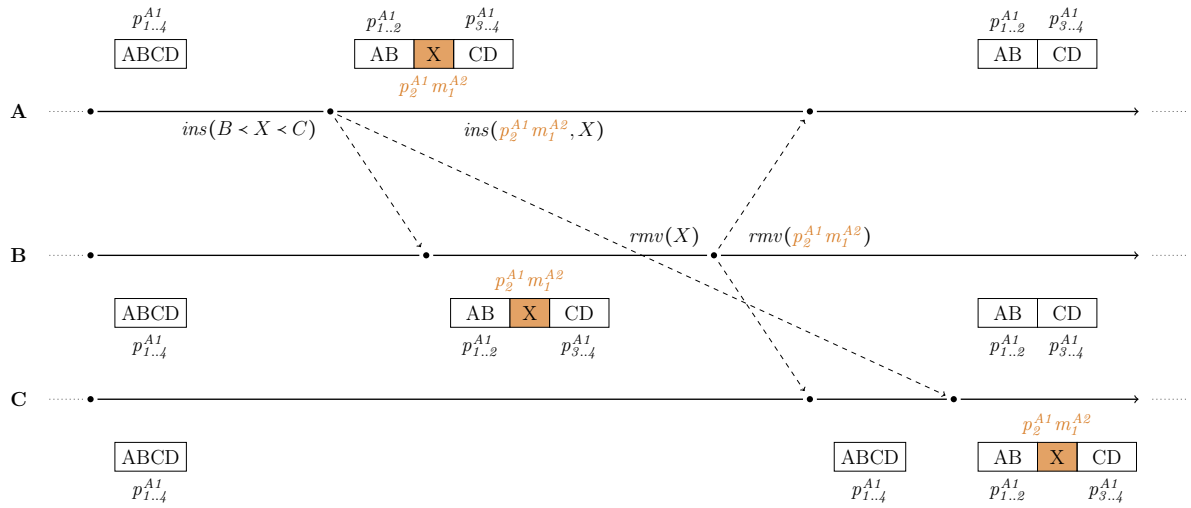


FIGURE 2.25 – Non-effet de l'opération *rmv* car reçue avant l'opération *ins* correspondante

Nous constatons alors que l'état à terme du noeud C diverge de celui des noeuds A et B, et cela malgré que les noeuds A, B et C aient intégré le même ensemble d'opérations. Ce résultat transgresse la propriété 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 *rmv* par rapport aux opérations *ins* correspondantes.

Définition du modèle de livraison

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

Définition 45 (Modèle de livraison LogootSplit)**.** Le modèle de livraison LogootSplit définit que :

(i) Une opération doit être livrée exactement une fois à chaque noeud.

(ii) Les opérations *ins* peuvent être livrées dans un ordre quelconque.

(iii) L'opération *rmv*(*idIntervals*) ne peut être livrée qu'après la livraison des opération d'insertions des éléments formant les *idIntervals*.

Il est à noter que ELVINGER [29] a récemment proposé dans ses travaux de thèse Dotted LogootSplit, un nouveau CRDT pour Séquence dont la synchronisation est basée sur les différences d'états. 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.

2.4.5 Limites

Intéressons-nous désormais aux limites de LogootSplit. Nous en identifions deux que nous détaillons ci-dessous : la croissance non-bornée de la taille des identifiants, et la fragmentation de la séquence en blocs courts.

Croissance non-bornée de la taille des identifiants

La première limite de ce CRDT, héritée de l'approche auquel il appartient, est la taille non-bornée de ses identifiants de position. Comme indiqué précédemment, LogootSplit génère des identifiants composés de plus en plus de tuples au fur et à mesure que l'espace dense des identifiants se sature.

Cependant, LogootSplit introduit un mécanisme favorisant la croissance des identifiants : les intervalles d'identifiants. Considérons l'exemple présenté dans la Figure 2.26.

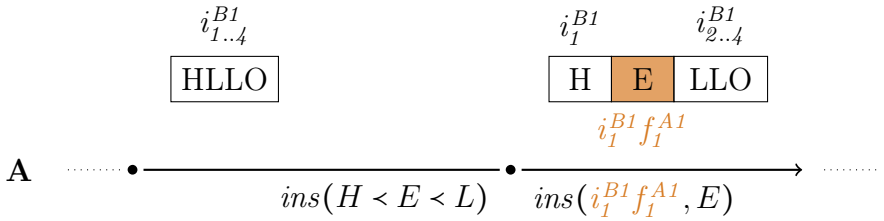


FIGURE 2.26 – Insertion menant à une augmentation de la taille des identifiants

Dans cet exemple, le noeud A insère un nouvel élément dans un intervalle d'identifiants existant, c.-à-d. entre deux identifiants contigus : i^{B1}_1 et i^{B1}_2. Ces deux identifiants étant contigus, il n'est pas possible de générer id, un identifiant de même taille tel que $i^{B1}_1 <_{id} id <_{id} i^{B1}_2$. Pour respecter l'ordre souhaité, LogootSplit génère donc un identifiant à partir de l'identifiant du prédecesseur et en y ajoutant un nouveau tuple, e.g. $i^{B1}_1 f^{A1}_1$.

Par conséquent, la taille des identifiants croît à chaque fois qu'un intervalle d'identifiants est scindé. Comme présenté précédemment (cf. sous-section 2.3.3, page 49), cette croissance augmente le surcoût en métadonnées, en calculs et en bande-passante du CRDT.

Fragmentation de la séquence en blocs courts

La seconde limite de LogootSplit est la fragmentation de l'état en une multitude de blocs courts. En effet, plusieurs contraintes sur la génération d'identifiants empêchent les noeuds d'ajouter des nouveaux éléments aux blocs existants :

Définition 46 (Contraintes sur l'ajout d'éléments à un bloc existant)**.** L'ajout d'éléments à un bloc existant doit respecter les règles suivantes :

(i) Seul le noeud qui a généré l'intervalle d'identifiants du bloc, c.-à-d. qui est l'auteur du bloc, peut ajouter des éléments à ce dernier.

(ii) L'ajout d'éléments à un bloc ne peut se faire qu'à la fin de ce dernier.

(iii) La suppression du dernier élément d'un bloc interdit tout ajout futur à ce bloc.

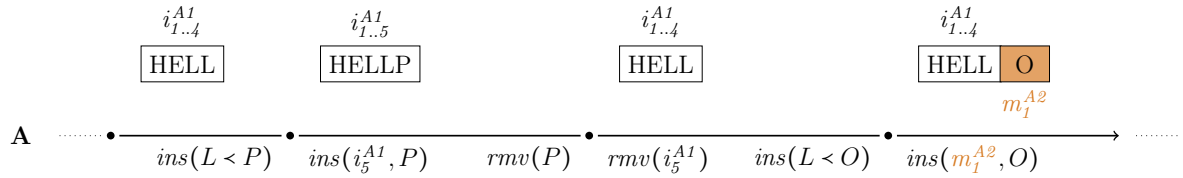


FIGURE 2.27 – Insertion menant à une augmentation de la taille des identifiants

La figure Figure 2.27 illustre ces règles.

Ainsi, ces limitations conduisent à la génération de nouveau blocs au fur et à mesure de la manipulation de la séquence. Nous conjecturons que, dans un cadre d'utilisation standard, la séquence est à terme fragmentée en de nombreux blocs de seulement quelques caractères chacun. Les blocs étant le niveau de granularité de la séquence, chaque nouveau bloc entraîne un surcoût en métadonnées et en calculs. Cependant, aucun mécanisme pour fusionner les blocs *a posteriori* n'est proposé. L'efficacité de la structure décroît donc au fur et à mesure que l'état se fragmente.

Synthèse

Les performances d'une séquence LogootSplit diminuent au fur et à mesure que celle-ci est manipulée et que des modifications sont effectuées dessus. Cette perte d'efficacité est dûe à la taille des identifiants de position qui croît de manière non-bornée, ainsi qu'au nombre généralement croissant de blocs.

Initialement, nous nous sommes focalisés sur un aspect du problème : la croissance du surcoût en métadonnées de la structure. Afin de quantifier ce problème, nous avons évalué par le biais de simulations [18] l'évolution de la taille de la séquence. La Figure 2.28 présente le résultat obtenu.

Sur cette figure, nous représentons l'évolution au fur et à mesure que des modifications sont effectuées sur une séquence LogootSplit de la taille de son contenu, sous la forme d'une ligne pointillée bleu, et de la taille de la séquence LogootSplit complète, sous la forme d'une ligne continue rouge. Nous constatons 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, c.-à-d. la taille des identifiants, les blocs composant la séquence LogootSplit, mais aussi la structure de données utilisée en interne pour représenter la séquence de manière efficace.

Nous jugeons donc nécessaire de proposer des mécanismes et techniques afin de mitiger le surcoût des CRDTs pour Séquence et sa croissance.

Matthieu: TODO : Serait plus intéressant de proposer des stats sur la taille des ids, le nombre de blocs composant la séquence, le nombre d'éléments par blocs et la proportion de la taille du contenu sur la taille de la structure de données en fonction du nombre d'opérations jouées. Pose la question de quand introduire le protocole suivi pour générer les traces.

18. Nous détaillons le protocole expérimental que nous avons défini pour ces simulations dans le **??**.

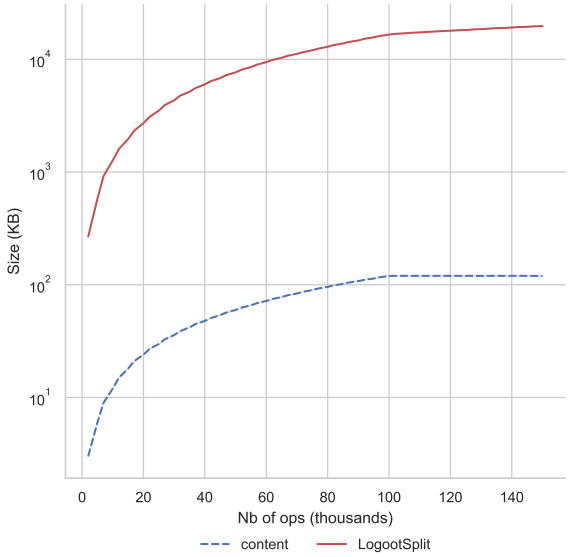


FIGURE 2.28 – Taille du contenu comparé à la taille de la séquence LogootSplit

2.5 Mitigation du surcoût des séquences répliquées sans conflits

L'augmentation du surcoût des CRDTs pour Séquence, qu'il soit dû à des pierres tombales ou à des identifiants de taille non-bornée, est un problème bien identifié dans la littérature [75, 76, 23, 24, 94, 95]. Plusieurs approches ont donc été proposées pour réduire sa croissance.

2.5.1 Mécanisme de GC des pierres tombales

Pour réduire l'impact des pierres tombales sur les performances de RGA, ROH et al. [75] proposent un mécanisme de GC des pierres tombales. Pour rappel, ce mécanisme nécessite qu'une pierre tombale ne puisse plus être utilisée comme prédecesseur par une opération *ins* reçue dans le futur pour pouvoir être supprimée définitivement. En d'autres termes, ce mécanisme repose sur la stabilité causale de l'opération de suppression pour supprimer la pierre tombale correspondante.

La stabilité causale est une contrainte forte, peu adaptée aux systèmes P2P dynamiques à large échelle. Notamment, la stabilité causale nécessite que chaque noeud du système fournisse régulièrement des informations sur son avancée, c.-à-d. quelles opérations il a intégré, pour progresser. Ainsi, il suffit qu'un noeud du système se déconnecte pour bloquer la stabilité causale, ce qui apparaît extrêmement fréquent dans le cadre d'un système P2P dynamique dans lequel nous n'avons pas de contrôle sur les noeuds.

À notre connaissance, il s'agit du seul mécanisme proposé pour l'approche à pierres tombales.

2.5.2 Ré-équilibrage de l'arbre des identifiants de position

Concernant l'approche à identifiants densément ordonnés, LETIA et al. [23] puis ZAWIRSKI et al. [24] proposent un mécanisme de ré-équilibrage de l'arbre des identifiants de position pour Treedoc [76]. Pour rappel, Treedoc souffre des problèmes suivants :

(i) Le déséquilibrage de son arbre des identifiants de position si les insertions sont effectuées de manière séquentielle à une position.

(ii) La présence de pierres tombales dans son arbre des identifiants de position lorsque des identifiants correspondants à des noeuds intermédiaires de l'arbre sont supprimés.

Pour répondre à ces problèmes, les auteurs présentent un mécanisme de ré-équilibrage de l'arbre supprimant par la même occasion les pierres tombales existantes, c.-à-d. un mécanisme réattribuant de nouveaux identifiants de position aux éléments encore présents. Ce mécanisme prend la forme d'une nouvelle opération, que nous notons *bal*.

Notons que l'opération *bal* contrevient à une des propriétés des identifiants de position densément ordonnés : leur *immutabilité* (cf. Définition 39, page 50). L'opération *bal* est donc intrinséquement non-commutative avec les opérations *ins* et *rmv* concurrentes. Pour assurer la convergence à terme des copies, les auteurs mettent en place un mécanisme de *catch-up*. Ce mécanisme consiste à transformer les opérations concurrentes aux opérations *bal* avant de les intégrer, de façon à prendre en compte les effets des ré-équilibrages.

Toutefois, l'opération *bal* n'est pas non plus commutative avec elle-même. Cette approche nécessite d'empêcher la génération d'opérations *bal* concurrentes. Pour cela, les auteurs proposent de reposer sur un protocole de consensus entre les noeuds pour la génération d'opérations *bal*.

De nouveau, l'utilisation d'un protocole de consensus est une contrainte forte, peu adaptée aux systèmes P2P dynamique à large échelle. Pour pallier à ce point, les auteurs proposent de répartir les noeuds dans deux groupes : le *core* et la *nebula*.

Le *core* est un ensemble, de taille réduite, de noeuds stables et hautement connectés tandis que la *nebula* est un ensemble, de taille non-bornée, de noeuds. Seuls les noeuds du *core* participent à l'exécution du protocole de consensus. Les noeuds de la *nebula* contribuent toujours au document par le biais des opérations *ins* et *rmv*.

Ainsi, cette solution permet d'adapter l'utilisation d'un protocole de consensus à un système P2P dynamique. Cependant, elle requiert de disposer de noeuds stables et bien connectés dans le système pour former le *core*. Cette condition est un obstacle pour le déploiement et la pérennité de cette solution.

2.5.3 Réduction de la croissance des identifiants de position

L'approche LSEQ [94, 95] est une approche visant à réduire la croissance des identifiants dans les Séquences CRDTs à identifiants densément ordonnés. Au lieu de réduire périodiquement la taille des métadonnées liées aux identifiants à l'aide d'un mécanisme de renommage coûteux, les auteurs définissent de nouvelles stratégies d'allocation des identifiants pour réduire leur vitesse de croissance.

Dans ces travaux, les auteurs notent que les stratégies d'allocation des identifiants proposées pour Logoot dans [72] et [90] ne sont adaptées qu'à un seul comportement

d'édition : l'édition séquentielle. Si les insertions sont effectuées en suivant d'autres comportements, les identifiants générés saturent rapidement l'espace des identifiants pour une taille donnée. Les insertions suivantes déclenchent alors une augmentation de la taille des identifiants. En conséquent, la taille des identifiants dans Logoot augmente de façon linéaire au nombre d'insertions, au lieu de suivre la progression logarithmique attendue.

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

Cette solution ne repose sur aucune coordination synchrone entre les noeuds. Sa complexité ne dépend pas non plus du nombre de noeuds du système. Elle nous apparaît donc adaptée aux systèmes P2P dynamique à large échelle.

Nous conjecturons cependant que cette approche perd ses bienfaits lorsqu'elle est couplée avec un CRDT pour Séquence à granularité variable. En effet, comme évoqué précédemment, toute insertion au sein d'un bloc provoque une augmentation de la taille de l'identifiant résultant (cf. section 2.4.5, page 60).

2.5.4 Synthèse

Ainsi, plusieurs approches ont été proposées dans la littérature pour réduire le surcoût des CRDTs pour Séquence. Cependant, aucune de ces approches ne nous apparaît adaptée pour les CRDTs pour Séquence à granularité variable dans le contexte de systèmes P2P dynamiques :

(i) Les approches présentées dans [75, 23, 24] reposent chacune sur des contraintes fortes dans les systèmes P2P dynamiques, c.-à-d. respectivement la stabilité causale des opérations et l'utilisation d'un protocole de consensus. Dans un système dans lequel nous n'avons aucun contrôle sur les noeuds et notamment leur disponibilité, ces contraintes nous apparaissent rédhibitoires.

(ii) L'approche présentée dans [94, 95] est conçue pour les CRDTs pour Séquence à identifiants densément ordonnés à granularité fixe. L'introduction de mécanismes d'aggrégation dynamique des éléments en blocs comme ceux présentés dans [28, 86], avec les contraintes qu'ils introduisent, nous semble contrarier les efforts effectués pour réduire la croissance des identifiants de position.

Nous considérons donc la problématique du surcoût des CRDTs pour Séquence à granularité variable toujours ouverte.

2.6 Synthèse

Les systèmes distribués adoptent le modèle de la réplication optimiste [16] pour offrir de meilleures garanties à leurs utilisateur-rices, en termes de disponibilité, latence et capacité de tolérance aux pannes [15].

Dans ce modèle, chaque noeud du système possède une copie de la donnée et peut la modifier sans coordination avec les autres noeuds. Il en résulte des divergences temporaires entre les copies respectives des noeuds. Pour résoudre les potentiels conflits provoqués par des modifications concurrentes et assurer la convergence à terme des copies, les systèmes utilisent les CRDTs [22] en place et lieu des types de données séquentiels.

Plusieurs CRDTs pour Séquence ont été proposés, notamment pour permettre la conception d'éditeurs collaboratifs pair-à-pair. Ces CRDTs peuvent être regroupés en deux catégories en fonction de leur mécanisme de résolution de conflits : l'approche à pierres tombales [74, 80, 79, 75, 86, 88] et l'approche à identifiants densément ordonnées [76, 72, 90, 28, 29].

Chacune de ces approches introduit néanmoins un surcoût croissant, au moins en termes de métadonnées et de calculs, pénalisant leurs performances à terme. Pour résoudre ce problème, plusieurs travaux ont été proposés, notamment [23, 24]. Cette approche présente un mécanisme de ré-équilibrage de l'arbre des identifiants de position pour les CRDTs pour Séquence à sur identifiants densément ordonnés.

Cette approche requiert cependant un protocole de consensus, des renommages concurrents provoquant un nouveau conflit. Cette contrainte empêche son utilisation dans les systèmes P2P ne disposant pas de noeuds suffisamment stables et bien connectés pour participer au protocole de consensus.

2.7 Proposition

Dans le cadre de cette thèse, nous proposons et présentons un nouveau mécanisme de réduction du surcoût pour les CRDTs pour Séquence à identifiants densément ordonnés et à granularité variable.

Ce mécanisme se distingue des travaux existants, notamment de [23, 24], par les aspects suivants :

(i) Il ne nécessite pas de coordination synchrone entre les noeuds.

(ii) Il ré-aggrége les éléments de la séquence en de nouveaux blocs pour réduire leur nombre.

Nous concevons ce mécanisme pour le CRDT LogootSplit. Toutefois, le principe de notre approche est générique. Ainsi, ce mécanisme peut être adapté pour proposer un équivalent pour d'autres CRDTs pour Séquence, e.g. RGASplit.

Nous présentons et détaillons ce mécanisme dans le chapitre suivant.

Chapitre 3

MUTE, un éditeur de texte web collaboratif P2P temps réel chiffré de bout en bout

Sommaire

3.1 Présentation 70
3.1.1 Objectifs 70
3.1.2 Fonctionnalités 70
3.1.3 Architecture système 72
3.1.4 Architecture logicielle 73
3.2 Couche interface utilisateur 75
3.3 Couche réplication 76
3.3.1 Modèle de données du document texte 76
3.3.2 Collaborateur-rices 77
3.3.3 Curseurs 81
3.4 Couche livraison 81
3.4.1 Livraison des opérations en exactement un exemplaire 82
3.4.2 Livraison de l'opération *remove* après l'opération *insert* 84
3.4.3 Livraison des opérations après l'opération *rename* introduisant leur époque 86
3.4.4 Livraison des opérations à terme 88
3.5 Couche réseau 89
3.5.1 Établissement d'un réseau P2P entre navigateurs 89
3.5.2 Topologie réseau 91
3.6 Couche sécurité 91
3.7 Conclusion 93

Les systèmes collaboratifs temps réels permettent à plusieurs utilisateur-rices de réaliser une tâche de manière coopérative. Ils permettent aux utilisateur-rices de consulter le contenu actuel, de le modifier et d'observer en direct les modifications effectuées par

les autres collaborateur-rices. L'observation en temps réel des modifications des autres favorise une réflexion de groupe et permet une répartition efficace des tâches. L'utilisation des systèmes collaboratifs se traduit alors par une augmentation de la qualité du résultat produit [96, 7].

Plusieurs outils d'édition collaborative centralisés basés sur l'approche OT [64] ont permis de populariser l'édition collaborative temps réel de texte [25, 97]. Ces approchent souffrent néanmoins de leur architecture centralisée. Notamment, ces solutions rencontrent des difficultés à passer à l'échelle [93, 98] et posent des problèmes de confidentialité [99, 100].

L'approche CRDT offre une meilleure capacité de passage à l'échelle et est compatible avec une architecture P2P [79]. Ainsi, de nombreux travaux [101, 102, 103] ont été entrepris pour proposer une alternative distribuée répondant aux limites des éditeurs collaboratifs centralisés. De manière plus globale, ces travaux s'inscrivent dans le nouveau paradigme d'application des *Local-First Softwares* [14, 104]. Ce paradigme vise le développement d'applications collaboratives, P2P, pérennes et rendant la souveraineté de leurs données aux utilisateurs.

Matthieu: TODO : Serait intéressant d'ajouter une catégorisation des éditeurs collaboratifs en fonction de leurs caractéristiques (décentralisé vs. p2p, pas de chiffrement vs. chiffrement serveur vs. chiffrement de bout en bout, OT vs CRDT vs mécanisme de résolution de conflits custom...) pour mettre en avant le caractère unique de MUTE

De manière semblable, l'équipe Coast conçoit depuis plusieurs années des applications avec ces mêmes objectifs et étudient les problématiques de recherche liées. Elle développe Multi User Text Editor (MUTE) [27] [19] [20], un éditeur collaboratif P2P temps réel chiffré de bout en bout. MUTE sert de plateforme d'expérimentation et de démonstration pour les travaux de l'équipe.

Ainsi, nous avons contribué à son développement dans le cadre de cette thèse. Notamment, nous avons participé à :

(i) L'implémentation des CRDTs LogootSplit [28] et RenamableLogootSplit [30] pour représenter le document texte.

(ii) L'implémentation de leur modèle de livraison de livraison respectifs.

(iii) L'implémentation d'un protocole d'appartenance au réseau, SWIM [33].

Dans ce chapitre, nous commençons par présenter le projet MUTE : ses objectifs, ses fonctionnalités et son architecture système et logicielle. Puis nous détaillons ses différentes couches logicielles : leur rôle, l'approche choisie pour leur implémentation et finalement leurs limites actuelles. Au cours de cette description, nous mettons l'emphase sur les composants auxquelles nous avons contribué, c.-à-d. les sous-sections 3.3.1, 3.4 et 3.3.2.Les systèmes collaboratifs temps réels permettent à plusieurs utilisateur-rices de réaliser une tâche de manière coopérative. Ils permettent aux utilisateur-rices de consulter le contenu actuel, de le modifier et d'observer en direct les modifications effectuées par les autres collaborateur-rices. L'observation en temps réel des modifications des autres favorise une

19. Disponible à l'adresse : `https://mutehost.loria.fr`
20. Code source disponible à l'adresse suivante : `https://github.com/coast-team/mute`

réflexion de groupe et permet une répartition efficace des tâches. L'utilisation des systèmes collaboratifs se traduit alors par une augmentation de la qualité du résultat produit [96, 7].

Plusieurs outils d'édition collaborative centralisés basés sur l'approche OT [64] ont permis de populariser l'édition collaborative temps réel de texte [25, 97]. Ces approchent souffrent néanmoins de leur architecture centralisée. Notamment, ces solutions rencontrent des difficultés à passer à l'échelle [93, 98] et posent des problèmes de confidentialité [99, 100].

L'approche CRDT offre une meilleure capacité de passage à l'échelle et est compatible avec une architecture P2P [79]. Ainsi, de nombreux travaux [101, 102, 103] ont été entrepris pour proposer une alternative distribuée répondant aux limites des éditeurs collaboratifs centralisés. De manière plus globale, ces travaux s'inscrivent dans le nouveau paradigme d'application des *Local-First Softwares* [14, 104]. Ce paradigme vise le développement d'applications collaboratives, P2P, pérennes et rendant la souveraineté de leurs données aux utilisateurs.

Matthieu: TODO : Serait intéressant d'ajouter une catégorisation des éditeurs collaboratifs en fonction de leurs caractéristiques (décentralisé vs. p2p, pas de chiffrement vs. chiffrement serveur vs. chiffrement de bout en bout, OT vs CRDT vs mécanisme de résolution de conflits custom...) pour mettre en avant le caractère unique de MUTE

De manière semblable, l'équipe Coast conçoit depuis plusieurs années des applications avec ces mêmes objectifs et étudient les problématiques de recherche liées. Elle développe Multi User Text Editor (MUTE) [27] [21] [22], un éditeur collaboratif P2P temps réel chiffré de bout en bout. MUTE sert de plateforme d'expérimentation et de démonstration pour les travaux de l'équipe.

Ainsi, nous avons contribué à son développement dans le cadre de cette thèse. Notamment, nous avons participé à :

(i) L'implémentation des CRDTs LogootSplit [28] et RenamableLogootSplit [30] pour représenter le document texte.

(ii) L'implémentation de leur modèle de livraison de livraison respectifs.

(iii) L'implémentation d'un protocole d'appartenance au réseau, SWIM [33].

Dans ce chapitre, nous commençons par présenter le projet MUTE : ses objectifs, ses fonctionnalités et son architecture système et logicielle. Puis nous détaillons ses différentes couches logicielles : leur rôle, l'approche choisie pour leur implémentation et finalement leurs limites actuelles. Au cours de cette description, nous mettons l'emphase sur les composants auxquelles nous avons contribué, c.-à-d. les sections 3.3, et 3.4.

21. Disponible à l'adresse : `https://mutehost.loria.fr`
22. Code source disponible à l'adresse suivante : `https://github.com/coast-team/mute`

3.1 Présentation

3.1.1 Objectifs

Comme indiqué dans l'introduction (cf. section 1.1, page 1), le but de ce projet est de proposer un éditeur de texte collaboratif Local-First Software (LFS), c.-à-d. un éditeur de texte collaboratif qui satisfait les propriétés suivantes :

(i) Toujours disponible, c.-à-d. qui permet à tout moment à un-e utilisateur-rice de consulter, créer ou éditer un document, même par exemple en l'absence de connexion internet.

(ii) Collaboratif, c.-à-d. qui permet à un-e utilisateur-rice de partager un document avec d'autres utilisateur-rices pour éditer à plusieurs le document, de manière synchrone et asynchrone. Nous considérons la capacité d'un-e utilisateur-rice à partager le document avec ses propres autres appareils comme un cas particulier de collaboration.

(iii) Performant, c.-à-d. qui garantit que le délai entre la génération d'une modification par un pair et l'intégration de cette dernière par un autre pair connecté soit assimilable à du temps réel et que ce délai ne soit pas impacté par le nombre de pairs dans la collaboration.

(iv) Pérenne, c.-à-d. qui garantit à ses utilisateur-rices qu'iels pourront continuer à utiliser l'application sur une longue période. Notamment, nous considérons la capacité des utilisateur-rices à configurer et déployer aisément leur propre instance du système comme un gage de pérennité du système.

(v) Garantissant la confidentialité des données, c.-à-d. qui permet à un-e utilisateur-rice de contrôler avec quelles personnes une version d'un document est partagée. Aussi, le système doit garantir qu'un adversaire ne doit pas être en mesure d'espionner les utilisateur-rices, e.g. en usurpant l'identité d'un-e utilisateur-rice ou en interceptant les messages diffusés sur le réseau.

(vi) Garantissant la souveraineté des données, c.-à-d. qui permet à un-e utilisateur-rice de maîtriser l'usage de ses données, e.g. pouvoir les consulter, modifier, partager ou encore exporter vers d'autres formats ou applications.

Ainsi, ces différentes propriétés nous conduisent à concevoir un éditeur de texte collaboratif P2P temps réel chiffré de bout en bout et qui est dépourvu d'autorités centrales.

3.1.2 Fonctionnalités

MUTE prend la forme d'une application web qui permet de créer et de gérer des documents textes. Chaque document se voit attribuer un identifiant, supposé unique. L'utilisateur-rice peut alors ouvrir et partager un document à partir de son URL.

L'application permet à l'utilisateur-rice d'être mis-e en relation avec les autres pairs actuellement connectés qui travaillent sur ce même document. Pour cela, l'application utilise le protocole WebRTC afin d'établir des connexions P2P avec ces derniers. Une fois les connexions P2P établies, le service fourni par le système pour mettre en relation les pairs n'est plus nécessaire.

Une fois connecté à un autre pair, l'utilisateur-rice récupère automatiquement les modifications effectuées par ses pairs de façon à obtenir la version courante du document. Iel peut alors modifier le document, c.-à-d. ajouter, supprimer du contenu ou encore modifier son titre. Ses modifications sont partagées en temps réel aux autres pairs connectés. À la réception de modifications, celles-ci sont intégrées à la copie locale du document. Figure 3.1 illustre l'interface utilisateur de l'éditeur de document de MUTE.

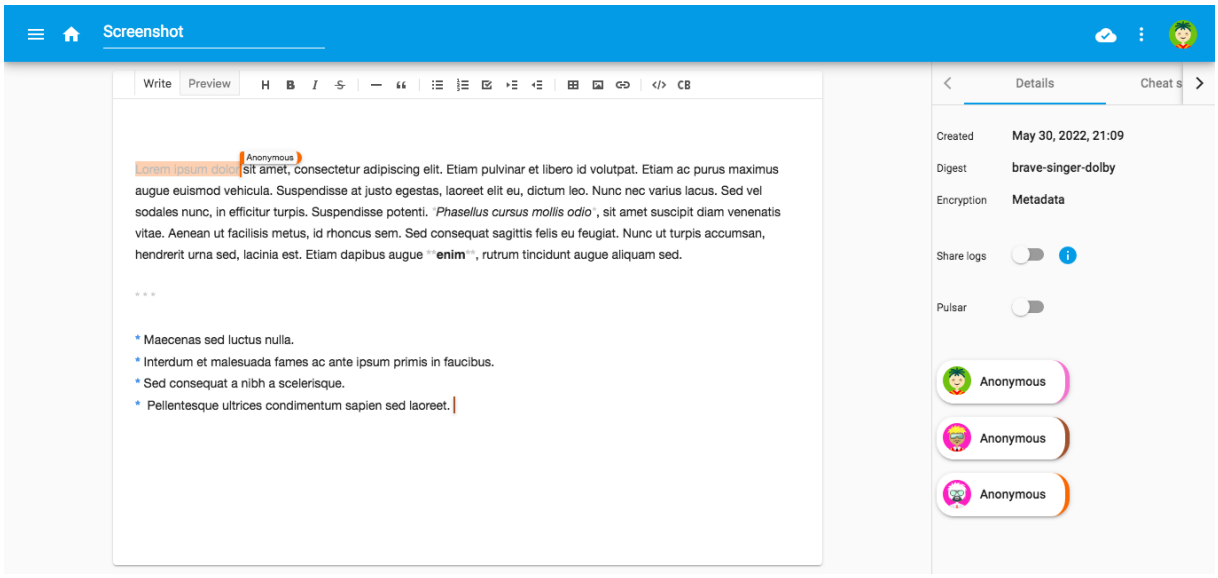


FIGURE 3.1 – Capture d'écran d'une session d'édition collaborative avec MUTE

Pour garantir la confidentialité des échanges, MUTE utilise un protocole de génération de clés de groupe. Ce protocole permet d'établir une clé de chiffrement connue seulement des pairs actuellement connectés, qui est ensuite utilisée pour chiffrer les messages entre pairs. Ce protocole permet de garantir les propriétés de *backward secrecy* et de *forward secrecy*.

Définition 47 (Backward Secrecy)**.** La *Backward Secrecy* est une propriété de sécurité garantissant qu'un nouveau noeud ne pourra pas déchiffrer avec la nouvelle clé de chiffrement les anciens messages chiffrés avec une clé de chiffrement précédente.

Définition 48 (Forward Secrecy)**.** La *Forward Secrecy* est une propriété de sécurité garantissant qu'un nouveau noeud ne pourra pas déchiffrer avec la nouvelle clé de chiffrement les futurs messages chiffrés avec une prochaine clé de chiffrement.

Une copie locale du document est sauvegardée dans le navigateur, avec l'ensemble des modifications. L'utilisateur-rice peut ainsi accéder à ses documents même sans connexion internet, pour les consulter ou modifier. Les modifications effectuées dans ce mode hors-ligne seront partagées aux collaborateur-rices à la prochaine connexion de l'utilisateur-rice.

Finalement, la page d'accueil de l'application permet aussi de lister ses documents. L'utilisateur-rice peut ainsi facilement parcourir ses documents, récupérer leur url pour les partager ou encore supprimer leur copie locale. Figure 3.2 illustre cette page de l'application.

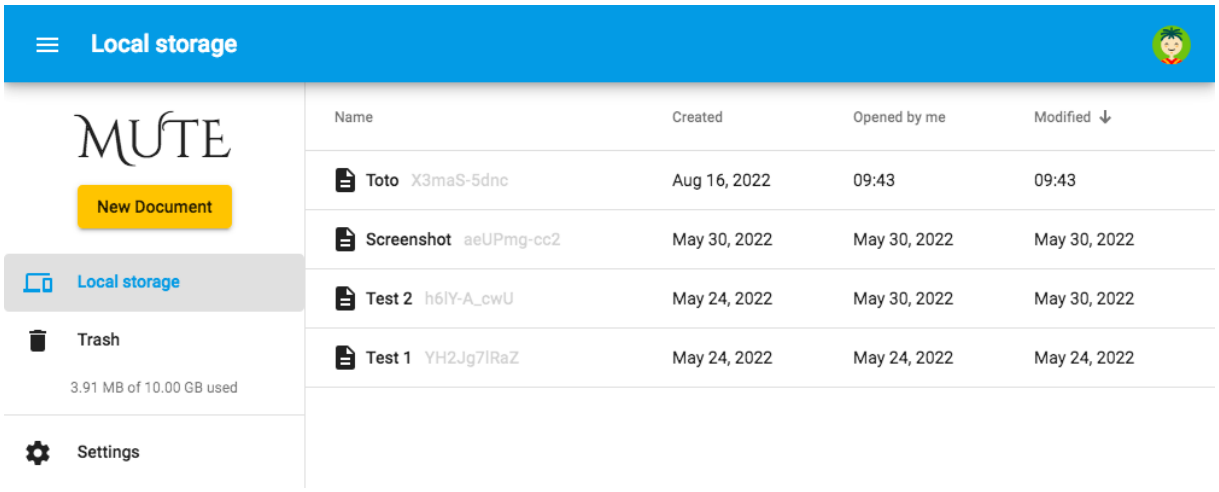


FIGURE 3.2 – Capture d'écran de la liste des documents.

3.1.3 Architecture système

Nous représentons l'architecture système d'une collaboration utilisant MUTE par la Figure 3.3.

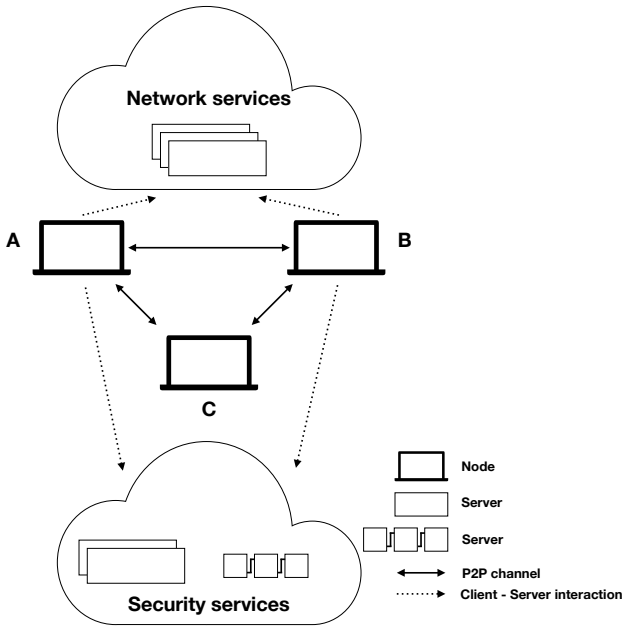


FIGURE 3.3 – Architecture système de l'application MUTE

Plusieurs types de noeuds composent cette architecture. Nous décrivons ci-dessous le type de chacun de ces noeuds ainsi que leurs rôles.

Pairs

Au centre de la collaboration se trouvent les noeuds qui correspondent aux utilisateurrices de l'application et à leurs appareils. Chaque noeud correspond à une instance de l'application MUTE, c.-à-d. l'éditeur collaboratif de texte. Chacun de ces noeuds peut donc consulter des documents et les modifier.

Ces noeuds forment un réseau P2P, qui leur permet d'échanger directement notamment pour diffuser les modifications effectuées sur le document. Les pairs interagissent aussi avec les autres types de noeuds, que nous décrivons dans les parties suivantes.

Notons qu'un noeud peut toutefois être déconnecté du système, c.-à-d. dans l'incapacité de se connecter aux autres pairs et d'interagir avec les autres types de noeuds. Cela ne l'empêche toutefois pas l'utilisateur-rice d'utiliser MUTE.

Services réseau

Nous décrivons par cette appelation l'ensemble des composants nécessaires à l'établissement et le bon fonctionnement du réseau P2P entre les appareils des utilisateur-rices.

Il s'agit de serveurs ayant pour buts de :

(i) Permettre à un pair d'obtenir les informations sur son propre état nécessaires pour l'établissement de connexions P2P.

(ii) Permettre à un pair de découvrir les autres pairs travaillant sur le même document et d'établir une connexion avec eux.

(iii) Permettre à des pairs de communiquer même si leur configurations réseaux respectives empêchent l'établissement d'une connection P2P directe.

Nous détaillons plus précisément chacun de ces services et les interactions entre les pairs et ces derniers dans la section 3.5.

Services sécurité

Nous décrivons par cette appelation l'ensemble des composants nécessaires à l'authentification des utilisateur-rices et à l'établissement de clés de groupe de chiffrement.

Il s'agit de serveurs ayant pour buts de :

(i) Permettre à un pair de s'authentifier.

(ii) Permettre à un pair de faire connaître sa clé publique de chiffrement.

(iii) Vérifier l'identité d'un pair.

(iv) Permetttre à un pair de vérifier le comportement honnête du ou des serveurs servant les clés publiques de chiffrement.

Nous dédions la section 3.6 à la description de ces différents services et les interactions des pairs avec ces derniers.

3.1.4 Architecture logicielle

Nous décrivons l'architecture logicielle d'un pair, c.-à-d. d'une instance de l'application MUTE dans un navigateur, dans la Figure 3.4.

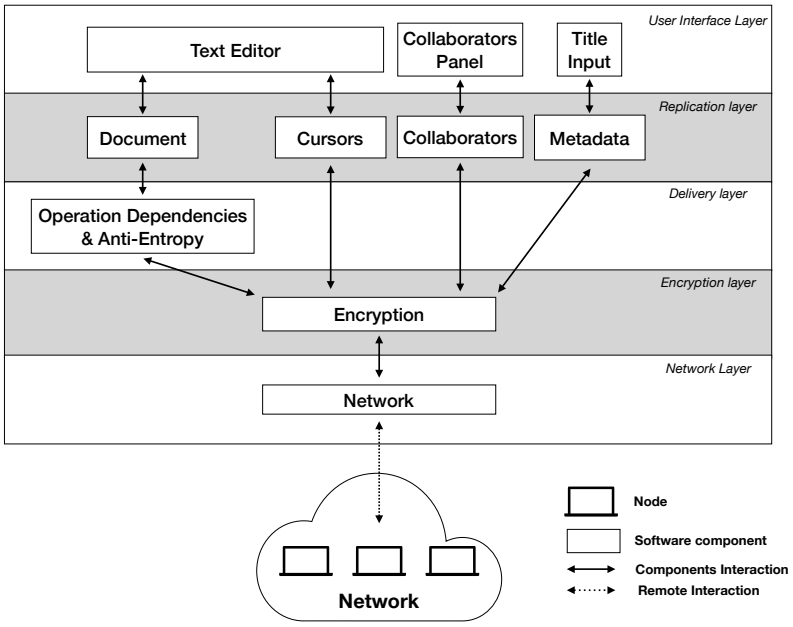


FIGURE 3.4 – Architecture logicielle de l'application MUTE

Cette architecture logicielle se compose de plusieurs composants, que nous regroupons par couche. Chacune de ces couches possède un rôle, que nous présentons brièvement ci-dessous avant de les décrire de manière plus détaillée dans leur section respective.

(i) La couche *interface utilisateur*, qui regroupe l'ensemble des composants permettant de communiquer des informations aux pairs et avec lesquelles ils peuvent interagir, c.-à-d. le document lui-même, son titre mais aussi la liste des collaborateur-rices actuellement connectés. Cette couche se charge de transmettre les actions de l'utilisateur-rice aux couches inférieures, et inversement de présenter à l'utilisateur-rice les modifications effectuées par ses pairs.

(ii) La couche *réplication*, qui regroupe l'ensemble des composants permettant de représenter les données répliquées entre pairs, c.-à-d. les CRDTs utilisés pour représenter le document, ses métadonnées (titre, date de création...), l'ensemble des collaborateurs et leur curseur. Cette couche se charge d'intégrer les modifications effectuées par l'utilisateur-rice et de transmettre les opérations correspondantes aux couches inférieures, et inversement d'intégrer les opérations effectuées par ses pairs et d'indiquer à la couche *interface utilisateur* les modifications correspondantes.

(iii) La couche *livraison*, qui est constitué d'un unique composant permettant de garantir les modèles de livraison requis par les différents CRDTs implémentés pour représenter le document. Cette couche se charge d'adjoindre aux opérations de l'utilisateur-rice leur(s) dépendance(s) avant de les transmettre aux couches inférieures, et de livrer les opérations de ses pairs une fois leur(s) dépendance(s) livrées au préalable, ou de les mettre en attente le cas échéant.

(iv) La couche *sécurité*, qui est constitué d'un unique composant gérant le chiffrement des messages. Cette couche se charge d'établir la clé de chiffrement de groupe, puis de

chiffrer les messages de l'utilisateur-rice avec cette dernière avant de les transmettre à la couche inférieure, et inversement de déchiffrer les messages chiffrés de ses pairs avant de les transmettre aux couches supérieures.

(v) La couche *réseau*, qui est constitué d'un unique composant permettant d'interagir avec le réseau P2P. Cette couche se charge d'établir les connexions P2P, puis permet de diffuser les messages chiffrés de l'utilisateur-rice à un ou plusieurs de ses pairs, et inversement de transmettre les messages chiffrés de ses pairs à la couche supérieure.

3.2 Couche interface utilisateur

Comme illustré par la Figure 3.1, l'interface de la page d'un document se compose principalement d'un éditeur de texte. Ce dernier supporte le langage de balisage Markdown [105]. Ainsi, l'éditeur permet d'inclure plusieurs éléments légers de style. Les balises du langage Markdown étant du texte, elles sont répliquées nativement par la structure de données utilisée en interne par MUTE.

L'interface de la page de l'éditeur de document est agrémentée de plusieurs mécanismes permettant d'établir une conscience de groupe entre les collaborateur-rices. L'indicateur en haut à droite de la page représente le statut de connexion de l'utilisateur-rice. Celui-ci permet d'indiquer à l'utilisateur-rice s'iel est actuellement connecté-e au réseau P2P, en cours de connexion, ou si un incident réseau a lieu.

De plus, MUTE affiche sur la droite de l'éditeur la liste des collaborateur-rices actuellement connecté-es. Un curseur ou une sélection distante est associée pour chaque membre de la liste. Ces informations permettent d'indiquer à l'utilisateur-rice dans quelles sections du document ses collaborateur-rices sont en train de travailler. Ainsi, iels peuvent se répartir la rédaction du document de manière implicite ou suivre facilement les modifications d'un-e collaborateur-rice.

Bien que fonctionnelle, cette interface souffre néanmoins de plusieurs limites. Notamment, nous n'avons pas encore pu étudier la littérature concernant les mécanismes de conscience pour supporter la collaboration, au-delà du système de curseurs distants.

Nous identifions ainsi plusieurs axes de travail pour ces mécanismes. Tout d'abord, l'axe des *mécanismes de conscience des changements*. Le but serait de proposer des mécanismes pour :

(i) Mettre en lumière de manière intelligible les modifications effectuées par les collabateur-rices dans le cadre de collaborations temps réel à large échelle. Un tel mécanisme représente un défi de part le débit important de changements, potentiellement à plusieurs endroits du document de manière quasi-simultanée, à présenter à l'utilisateur-rice.

(ii) Mettre en lumière de manière intelligible les modifications effectuées par les collaborateur-rices dans le cadre de collaborations asynchrones. De nouveau, ce mécanisme représente un défi de part la quantité massive de changements, une fois encore potentiellement à plusieurs endroits du document, à présenter à l'utilisateur-rice.

Une piste de travail potentiellement liée serait l'ajout d'une fonctionnalité d'historique du document, permettant aux utilisateur-rices de parcourir ses différentes versions obtenues au fur et à mesure des modifications. L'intégration d'une telle fonctionnalité dans un éditeur P2P pose cependant plusieurs questions : quel historique présenter aux utilisateur-rices, sachant que chacun-e a potentiellement observé un ordre différent des modifications ? Doit-on convenir d'une seule version de l'historique ? Dans ce cas, comment choisir et construire cet historique ?

Le second axe de travail sur les mécanismes de conscience concerne les *mécanismes de conscience de groupe*. Actuellement, nous affichons l'ensemble des collaborateur-rices actuellement connecté-es. Cette approche s'avère lourde voire entravante dans le cadre de collaborations à large échelle où le nombre de collaborateur-rices dépasse plusieurs centaines. Il convient donc de déterminer quelles informations présenter à l'utilisateur-rice dans cette situation, e.g. une liste compacte de pairs et leur curseur respectif, ainsi que le nombre de pairs total.

Ainsi, pour chacun de ses axes d'amélioration, il convient d'étudier leur littérature respective et de déterminer les solutions proposées qui sont adaptées à MUTE.

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 [23] (cf. section 2.4, page 52), soit de RenamableLogootSplit[23] (cf. **??**, page **??**) ou soit de Dotted LogootSplit [24] [29]. 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 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 livrée 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.

En plus du texte, MUTE maintient un ensemble de métadonnées par document. Par

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

exemple, les utilisateurs peuvent donner un titre au document. Pour représenter cette donnée additionnelle, nous associons un Last-Writer-Wins Register CRDT synchronisé par opérations [22] au document. De façon similaire, nous utilisons un First-Writer-Wins Register CRDT synchronisé par opérations pour représenter la date de création du document.

3.3.2 Collaborateur-rices

Pour assurer la qualité de la collaboration même à distance, il est important d'offrir des fonctionnalités de conscience de groupe aux utilisateurs. Une de ces fonctionnalités est de fournir la liste des collaborateur-rices actuellement connectés. Les protocoles d'appartenance au réseau sont une catégorie de protocoles spécifiquement dédiée à cet effet. Ainsi, nous devions en implémenter un dans MUTE.

MUTE présente cependant plusieurs contraintes liées à notre modèle du système que le protocole sélectionné doit respecter. Tout d'abord, le protocole doit être compatible avec un environnement P2P, où les noeuds partagent les mêmes droits et responsabilités. De plus, le protocole doit présenter une capacité de passage à l'échelle pour être adapté aux collaborations à large échelle.

En raison de ces contraintes, notre choix s'est porté sur le protocole SWIM [33]. Proposé par Das et al., ce protocole d'appartenance au réseau offre les propriétés intéressantes suivantes. Tout d'abord, le nombre de messages diffusés sur le réseau est proportionnel de façon linéaire au nombre de pairs. Pour être plus précis, le nombre de messages envoyés par un pair par période du protocole est constant. De plus, il fournit à chaque noeud une vue de la liste des collaborateur-rices cohérente à terme, même en cas de réception désordonnée des messages du protocoles. Finalement, il intègre un mécanisme permettant de réduire le taux de faux positifs, c.-à-d. le taux de pairs déclarés injustement comme défaillants.

Pour cela, SWIM découple les deux composants d'un protocole d'appartenance au réseau : le mécanisme de *détection des défaillances des pairs* et le mécanisme de *dissémination des mises à jour du groupe.*

Mécanisme de détection des défaillances des pairs

Le mécanisme de détection des défaillances des pairs est executé de manière périodique, toutes les T unités de temps, par chacun des noeuds du système de manière non-coordonnée. Son fonctionnement est illustré par la Figure 3.5.

Dans cet exemple, le réseau est composé des trois noeuds A, B et C. Le noeud C démarre l'exécution du mécanisme de détection des défaillances.

Tout d'abord, le noeud C sélectionne un noeud cible de manière aléatoire, ici B, et lui envoie un message *ping*. À la réception de ce message, le noeud B lui signifie qu'il est toujours opérationnel en lui répondant avec un message *ack*. À la réception de ce message par C, cette exécution du mécanisme de détection des défaillances prendrait fin. Mais dans l'exemple présenté ici, ce message est perdu par le réseau.

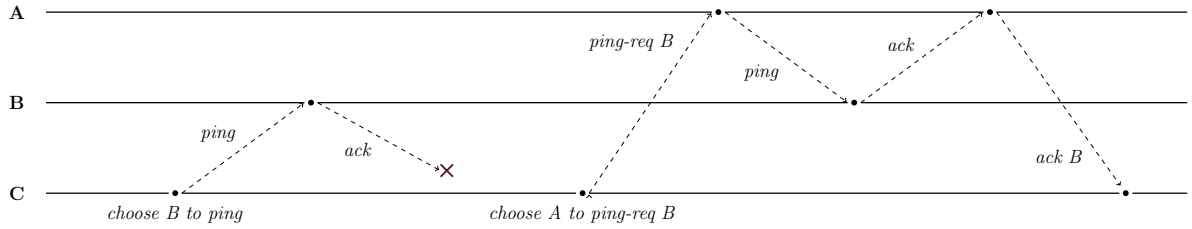


FIGURE 3.5 – Exécution du mécanisme de détection des défaillances par le noeud C pour tester le noeud B

En l'absence de réponse de la part de B au bout d'un temps spécifié au préalable, le noeud C passe à l'étape suivante du mécanisme. Le noeud C sélectionne un autre noeud, ici A, et lui demande de vérifier via le message *ping-req B* si B a eu une défaillance. À la réception de la requête de ping, le noeud A envoie un message *ping* à B. Comme précédemment, B répond au *ping* par le biais d'un *ack* à A. A informe alors C du bon fonctionnement du B via le message *ack B*. Le mécanisme prend alors fin, jusqu'à sa prochaine exécution.

Si C n'avait pas reçu de réponse suite à sa *ping-req B* envoyée à A, C aurait supposé que B a eu une défaillance. Afin de réduire le taux de faux positifs, SWIM ne considère pas directement les noeuds n'ayant pas répondu comme en panne : ils sont tout d'abord *suspectés* d'être en panne. Après un certain temps sans signe de vie d'un noeud suspecté d'être en panne, le noeud est *confirmé* comme défaillant.

L'information qu'un noeud est suspecté d'être en panne est propagé dans le réseau via le mécanisme de dissémination des mises à jour du groupe décrit ci-dessous. Si un noeud apprend qu'il est suspecté d'une panne, il dissémine à son tour l'information qu'il est toujours opérationnel pour éviter d'être confirmé comme défaillant.

Pour éviter qu'un message antérieur n'invalide une suspicion d'une défaillance et retarde ainsi sa détection, SWIM introduit un numéro d'*incarnation*. Chaque noeud maintient un numéro d'incarnation. Lorsqu'un noeud apprend qu'il est suspecté d'une panne, il incrémente son numéro d'incarnation avant de propager l'information contradictoire.

Ainsi, afin de représenter la liste des collaborateur-rices, le protocole SWIM utilise la structure de données présentée par la Définition 49 :

Définition 49 (Liste des collaborateur-rices)**.** La *liste des collaborateur-rices* est un ensemble de triplets $\langle nodeId, nodeStatus, nodeIncarn \rangle$ où

— *nodeId*, l'identifiant du noeud correspondant à ce tuple.

— *nodeStatus*, le statut courant du noeud correspondant à ce tuple, c.-à-d. *Alive* s'il est considéré comme opérationnel, *Suspect* s'il est suspecté d'une défaillance, *Confirm* s'il est considéré comme défaillant.

— *nodeIncarn*, le numéro d'incarnation maximal, c.-à-d. le plus récent, connu pour le noeud correspondant à ce tuple.

Chaque noeud réplique cette liste et la fait évoluer au cours de l'exécution du mécanisme présenté jusqu'ici. Lorsqu'une mise à jour est effectuée, celle-ci est diffusée de la manière présentée ci-dessous.

Mécanisme de dissémination des mises à jour du groupe

Quand l'exécution du mécanisme de détection des défaillances par un noeud met en lumière une évolution de la liste des collaborateur-rices, cette mise à jour doit être propagée au reste des noeuds.

Or, diffuser cette mise à jour à l'ensemble du réseau serait coûteux pour un seul noeud. Afin de propager cette information de manière efficace, SWIM propose d'utiliser un protocole de diffusion épidémique : le noeud transmet la mise à jour qu'à un nombre réduit λ [25] de pairs, qui se chargeront de la transmettre à leur tour. Le mécanisme de dissémination des mises à jour de SWIM fonctionne donc de la manière suivante.

Chaque mise à jour du groupe est stockée dans une liste et se voit attribuer un compteur, initialisé avec $\lambda \log n$ avec n le nombre de noeuds. À chaque génération d'un message pour le mécanisme de détection des défaillances, un nombre arbitraire de mises à jour sont sélectionnées dans la liste et attachées au message. Leur compteurs respectifs sont décrémentés. Une fois que le compteur d'une mise à jour atteint 0, celle-ci est retirée de la liste.

À la réception d'un message, le noeud le traite comme définit précédemment en section 3.3.2. De manière additionnelle, il intègre dans sa liste des collaborateur-rices les mises à jour attachées au message en utilisant la règle suivante :

$$\forall i, j, k \cdot i \leq j \cdot \langle Alive, i\rangle < \langle Suspect, j\rangle < \langle Confirm, k\rangle$$

Ainsi, le mécanisme de dissémination des mises à jour du groupe réutilise les messages du mécanisme de détection des défaillances pour diffuser les modifications. Cela permet de propager les évolutions de la liste des collaborateur-rices sans ajouter de message supplémentaire. De plus, les règles de précédence sur l'état d'un collaborateur permettent aux noeuds de converger même si les mises à jour sont reçues dans un ordre distinct.

Modifications apportées

Nous avons ensuite apporté plusieurs modifications à la version du protocole SWIM présentée dans [33]. Notre première modification porte sur l'ordre de priorité entre les états d'un pair.

Modification de l'ordre de précédence. Dans la version originale, un pair désigné comme défaillant l'est de manière irrévocable. Ce comportement est dû à la règle de précédence suivante :

$$\forall i, j \in \mathbb{N}, \forall s \in \{Alive, Suspect\} \cdot \langle s, i\rangle < \langle Confirm, j\rangle$$

25. [33] montre que choisir une valeur constante faible comme λ suffit néanmoins à garantir la dissémination des mises à jour à l'ensemble du réseau.

pour un noeud donné. Ainsi, un noeud déclaré comme défaillant par un autre noeud doit changer d'identité pour rejoindre de nouveau le groupe.

Ce choix n'est cependant pas anodin : il implique que la taille de la liste des collaborateur-rices croît de manière linéaire avec le nombre de connexions. S'agissant du paramètre avec le plus grand ordre de grandeur de l'application, nous avons cherché à le diminuer.

Nous avons donc modifié les règles de précédence de la manière suivante :

$$\forall i, j \in \mathbb{N}, i < j, \forall s, t \in \{Alive, Suspect, Confirm\} \cdot \langle i, s \rangle < \langle j, t \rangle$$

et

$$\forall i \in \mathbb{N} \cdot \langle i, Alive \rangle < \langle i, Suspect \rangle < \langle i, Confirm \rangle$$

Ces modifications permettent de donner la précédence au numéro d'incarnation, et d'utiliser le statut du collaborateur pour trancher seulement en cas d'égalité par rapport au numéro d'incarnation actuel. Ceci permet à un noeud auparavant déclaré comme défaillant de revenir dans le groupe en incrémentant son numéro d'incarnation. La taille de la liste des collaborateur-rices devient dès lors linéaire par rapport au nombre de noeuds.

Ces modifications n'ont pas d'impact sur la convergence des listes des collaborateur-rices des différents noeuds. Une étude approfondie reste néanmoins à effectuer pour déterminer si ces modifications ont un impact sur la vitesse à laquelle un noeud défaillant est déterminé comme tel par l'ensemble des noeuds.

Ajout d'un mécanisme de synchronisation. La seconde modification que nous avons effectué concerne l'ajout d'un mécanisme de synchronisation entre pairs. En effet, le papier ne précise pas de procédure particulière lorsqu'un nouveau pair rejoint le réseau. Pour obtenir la liste des collaborateur-rices, ce dernier doit donc la demander à un autre pair.

Nous avons donc implémenté pour la liste des collaborateur-rices un mécanisme d'anti-entropie : à sa connexion, puis de manière périodique, un noeud envoie une requête de synchronisation à un noeud cible choisi de manière aléatoire. Ce message sert aussi à transmettre l'état courant du noeud source au noeud cible. En réponse, le noeud cible lui envoie l'état courant de sa liste. À la réception de cette dernière, le noeud source fusionne la liste reçue avec sa propre liste. Cette fusion conserve l'entrée la plus récente pour chaque noeud.

Pour récapituler, les mises à jour du groupe sont diffusées de manière atomique de façon épidémique, en utilisant les messages du mécanisme de détection des défaillances des noeuds. De manière additionnelle, un mécanisme d'anti-entropie permet à deux noeuds de synchroniser leur état. Ce mécanisme nous permet de pallier aux défaillances éventuelles du réseau. Ainsi, nous avons dans les faits mis en place un CRDT synchronisé par différences d'états (cf. section 2.2.2, page 26) pour la liste des collaborateur-rices.

Synthèse

Pour générer et maintenir la liste des collaborateur-rices, nous avons implémenté le protocole distribué d'appartenance au réseau SWIM [33]. Par rapport à la version originale,

nous avons procédé à plusieurs modifications, notamment pour gérer plus efficacement les reconnexions successives d'un même noeud.

Ainsi, nous avons implémenté un mécanisme dont la complexité spatiale dépend linéairement du nombre de noeuds. Sa complexité en temps et sa complexité en communication, elles, sont indépendantes de ce paramètre. Elles dépendent en effet de paramètres dont nous choisissons les valeurs : la fréquence de déclenchement du mécanisme de détection de défaillance et le nombre de mises à jour du groupe propagées par message.

Des améliorations au protocole SWIM furent proposées dans [34]. Ces modifications visent notamment à réduire le délai de détection d'un noeud défaillant, ainsi que réduire le taux de faux positifs. Ainsi, une perspective est d'implémenter ces améliorations dans MUTE.

3.3.3 Curseurs

Toujours dans le but d'offrir des fonctionnalités de conscience de groupe aux utilisateurs pour leur permettre de se coordonner aisément, nous avons implémenté dans MUTE l'affichage des curseurs distants.

Pour représenter fidèlement la position des curseurs des collaborateur-rices distants, nous nous reposons sur les identifiants du CRDT choisi pour représenter la séquence. Le fonctionnement est similaire à la gestion des modifications du document : lorsque l'éditeur indique que l'utilisateur a déplacé son curseur, nous récupérons son nouvel index. Nous recherchons ensuite l'identifiant correspondant à cet index dans la séquence répliquée et le diffusons aux collaborateur-rices.

À la réception de la position d'un curseur distant, nous récupérons l'index correspondant à cet identifiant dans la séquence répliquée et représentons un curseur à cet index. Il est intéressant de noter que si l'identifiant a été supprimé en concurrence, nous pouvons à la place récupérer l'index de l'élément précédent et ainsi indiquer à l'utilisateur où son collaborateur est actuellement en train de travailler.

De façon similaire, nous gérons les sélections de texte à l'aide de deux curseurs : un curseur de début et un curseur de fin de sélection.

3.4 Couche livraison

Comme indiqué précédemment, la couche livraison est formée d'un unique composant, que nous nommons module de livraison. Ce module est associé aux CRDTs synchronisés par opérations représentant le document texte, c.-à-d. LogootSplit ou RenamableLogootSplit.

Le rôle de ce module est de garantir que le modèle de livraison des opérations requis par le CRDT pour assurer la convergence à terme (cf. **??**, page **??**) soit satisfait, c.-à-d. que l'ensemble des opérations soient livrées dans un ordre correct à l'ensemble des noeuds.

Pour cela, le module de livraison doit implémenter les contraintes imposées par ces CRDTs sur l'ordre de livraison des opérations (cf. Définition 45, page 59 et **??**, page 59).

Pour rappel, le modèle de livraison de RenamableLogootSplit est le suivant :

(i) Une opération doit être livrée à l'ensemble des noeuds à terme,

(ii) Une opération doit être livrée qu'une seule et unique fois aux noeuds,

(iii) Une opération *remove* doit être livrée à un noeud une fois que les opérations *insert* des éléments concernés par la suppression ont été livrées à ce dernier.

(iv) Une opération peut être délivrée à un noeud qu'à partir du moment où l'opération *rename* qui a introduit son époque de génération a été délivrée à ce même noeud.

Nous décrivons ci-dessous comment nous assurons chacune de ces contraintes.

3.4.1 Livraison des opérations en exactement un exemplaire

Afin de respecter la contrainte de livraison en exactement un exemplaire, il est nécessaire d'identifier de manière unique chaque opération. Pour cela, le module de livraison ajoute un *Dot* [77] à chaque opération :

Définition 50 (Dot)**.** Un *Dot* est une paire $\langle nodeId, nodeSyncSeq \rangle$ où

(i) *nodeId*, l'identifiant unique du noeud qui a généré l'opération.

(ii) *nodeSyncSeq*, le numéro de séquence courant du noeud à la génération de l'opération.

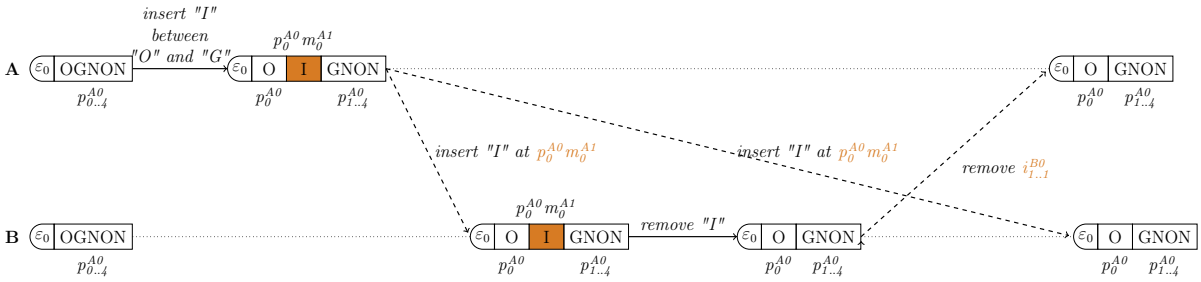
Il est à noter que *nodeSyncSeq* est différent du *nodeSeq* utilisé dans LogootSplit et RenamableLogootSplit (cf. section 2.4, page 52). En effet, *nodeSyncSeq* se doit d'augmenter à chaque opération tandis que *nodeSeq* n'augmente qu'à la création d'un nouveau bloc. Les contraintes étant différentes, il est nécessaire de distinguer ces deux données.

Chaque noeud maintient une structure de données représentant l'ensemble des opérations reçues par le pair. Elle permet de vérifier à la réception d'une opération si le dot de cette dernière est déjà connu. S'il s'agit d'un nouveau dot, le module de livraison peut livrer l'opération au CRDT et ajouter son dot à la structure. Le cas échéant, cela indique que l'opération a déjà été livrée précédemment et doit être ignorée cette fois-ci.

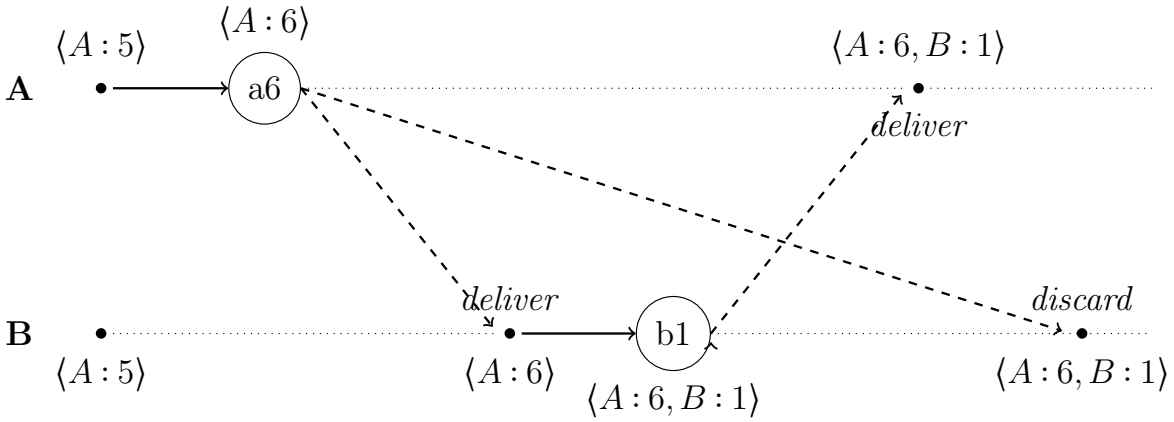
Plusieurs structures de données sont adaptées pour maintenir l'ensemble des opérations reçues. Dans le cadre de MUTE, nous avons choisi d'utiliser un vecteur de version. Cette structure nous permet de réduire à un dot par noeud le surcoût en métadonnées du module de livraison, puisqu'il ne nécessite que de stocker le dot le plus récent par noeud. Cette structure permet aussi de vérifier en temps constant si une opération est déjà connue. La Figure 3.6 illustre son fonctionnement.

Dans cet exemple, qui reprend celui de la Figure 2.24, deux noeuds A et B répliquent une séquence. Initialement, celle-ci contient les éléments "OGNON". Ces éléments ont été insérés un par un par le noeud A, donc par le biais des opérations *a1* à *a5*. Le module de livraison de chaque noeud maintient donc initialement le vecteur de version $\langle A : 5 \rangle$.

Le noeud A insère l'élément "I" entre les éléments "O" et "G". Cette modification est alors labellisée *a6* par son module de livraison et est envoyée au noeud B. À la réception de cette opération, le module de B compare son dot avec son vecteur de version local.



(a) Exécution avec livraison multiple d'une opération *insert*



(b) État et comportement du module de livraison au cours de l'exécution décrite en Figure 3.6a

FIGURE 3.6 – Gestion de la livraison en exactement un exemplaire des opérations

L'opération *a6* étant la prochaine opération attendue de A, celle-ci est acceptée : elle est alors livrée au CRDT et le vecteur de version est mis à jour.

Le noeud B supprime ensuite l'élément nouvellement inséré. S'agissant de la première modification de B, cette modification *b1* ajoute l'entrée correspondante dans le vecteur de version $\langle A:6, B:1\rangle$. L'opération est envoyée au noeud A. Cette opération étant la prochaine opération attendue de B, elle est acceptée et livrée.

Finalement, le noeud B reçoit de nouveau l'opération *a6*. Son module de livraison détermine alors qu'il s'agit d'un doublon : l'opération apparaît déjà dans le vecteur de version $\langle A:6, B:1\rangle$. L'opération est donc ignorée, et la résurgence de l'élément "I" illustrée dans la Figure 2.24 est évitée.

Il est à noter que dans le cas où un noeud reçoit une opération avec un dot plus élevé que celui attendu (e.g. le noeud A reçoit une opération *b3* à la fin de l'exemple), cette opération est mise en attente. En effet, livrer cette opération nécessiterait de mettre à jour le vecteur de version à $\langle A:6, B:3\rangle$ et masquerait le fait que l'opération *b2* n'a jamais été reçue. L'opération *b3* serait donc mise en attente jusqu'à la livraison de l'opération *b2*.

Ainsi, l'implémentation de livraison en exactement un exemplaire d'une opération avec un vecteur de version comme structure de données force une livraison First In, First Out (FIFO) des opérations par noeuds. Il s'agit d'une contrainte non-nécessaire et qui

peut introduire des délais dans la collaboration, notamment si une opération d'un noeud est perdue par le réseau. Nous jugeons cependant acceptable ce compromis entre le surcoût du mécanisme de livraison en exactement un exemplaire et son impact sur l'expérience utilisateur.

Pour retirer cette contrainte superflue, il est possible de remplacer cette structure de données par un *Interval Version Vector* [106]. Au lieu d'enregistrer seulement le dernier dot intégré par noeud, cette structure de données enregistre les intervalles de dots intégrés. Ceci permet une livraison *out of order* des opérations tout en garantissant une livraison en exactement un exemplaire et en compactant efficacement les données stockées par le module de livraison à terme.

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

La seconde contrainte que le modèle de livraison doit respecter spécifie qu'une opération *remove* doit être livrée après les opérations *insert* insérant les éléments concernés.

Pour cela, le module de livraison ajoute un ensemble *Deps* à chaque opération *remove* avant de la diffuser :

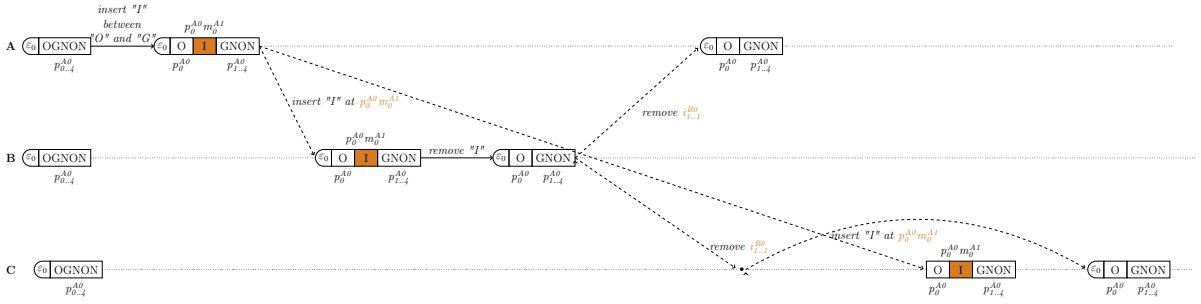
Définition 51 (Deps)**.** *Deps* est un ensemble d'opérations. Il représente l'ensemble des opérations dont dépend l'opération *remove* et qui doivent donc être livrées au préalable.

Plusieurs structures de données sont adaptées pour représenter les dépendances de l'opération *remove*. Dans le cadre de MUTE, nous avons choisi d'utiliser un ensemble de dots : pour chaque élément supprimé par l'opération *remove*, nous identifions le noeud l'ayant inséré et nous ajoutons le dot correspondant à l'opération la plus récente de ce noeud à l'ensemble des dépendances. Cette approche nous permet de limiter à un dot par élément supprimé le surcoût en métadonnées des dépendances et de les calculer en un temps linéaire par rapport au nombre d'éléments supprimés. Nous illustrons le calcul et l'utilisation des dépendances de l'opération *remove* à l'aide de la Figure 3.7.

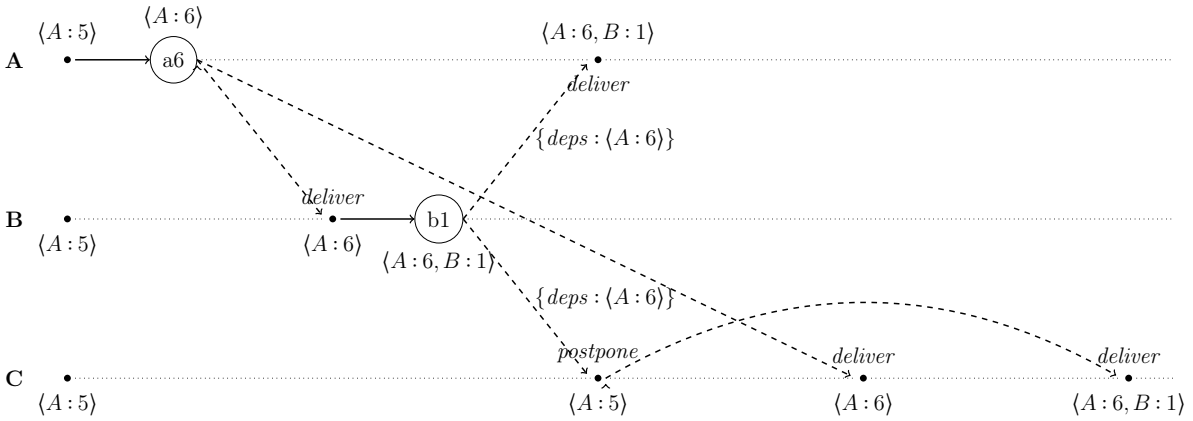
Cet exemple reprend et complète celui de la Figure 3.7. Trois noeuds A, B et C répliquent et éditent collaborativement une séquence. Les trois noeuds partagent le même état initial : une séquence contenant les éléments "OGNON" et un vecteur de version $\langle A:5\rangle$.

Le noeud A insère l'élément "I" entre les éléments "O" et "G". Cet élément se voit attribué l'identifiant $p_0^{A0}m_0^{A1}$. L'opération correspondante *a6* est diffusée aux autres noeuds.

À la réception de cette dernière, le noeud B supprime l'élément "I" nouvellement inséré et génère l'opération *b1* correspondante. Comme indiqué précédemment, l'opération *b1* étant une opération *remove*, le module de livraison calcule ses dépendances avant de la diffuser. Pour chaque élément supprimé ("I"), le module de livraison récupère l'identifiant de l'élément ($p_0^{A0}m_0^{A1}$) et en extrait l'identifiant du noeud qui l'a inséré (A). Le module ajoute alors le dot de l'opération la plus récente reçue de ce noeud ($\langle A:6\rangle$) à l'ensemble des dépendances de l'opération. L'opération est ensuite diffusée.



(a) Exécution avec livraison dans le désordre d'une insertion et de sa suppression



(b) État et comportement du module de livraison au cours de l'exécution décrite en Figure 3.7a

FIGURE 3.7 – Gestion de la livraison des opérations *remove* après les opérations *insert* correspondantes

À la réception de l'opération *b1*, le noeud A vérifie s'il possède l'ensemble des dépendances de l'opération. Le noeud A ayant déjà intégré l'opération *a6*, le module de livraison livre l'opération *b1* au CRDT.

À l'inverse, lorsque le noeud C reçoit l'opération *b1*, il n'a pas encore reçu l'opération *a6*. L'opération *b1* est alors mise en attente. À la réception de l'opération *a6*, celle-ci est livrée. Le module de livraison ré-évalue alors le cas de l'opération *b1* et détermine qu'elle peut à présent être livrée.

Il est à noter que notre approche pour générer l'ensemble des dépendances est une approximation. En effet, nous ajoutons les dots des opérations les plus récentes des auteurs des éléments supprimés. Nous n'ajoutons pas les dots des opérations qui ont spécifiquement insérés les éléments supprimés. Pour cela, il serait nécessaire de parcourir le log des opérations à la recherche des opérations *insert* correspondante. Cette méthode serait plus coûteuse, sa complexité dépendant du nombre d'opérations dans le log d'opérations, et incompatible avec un mécanisme tronquant le log des opérations en utilisant la stabilité causale. Notre approche introduit un potentiel délai dans la livraison d'une opération *remove* par rapport à une livraison utilisant ses dépendances exactes, puisqu'elle va reposer sur des opérations plus récentes et potentiellement encore inconnues par le noeud. Mais il s'agit là aussi d'un compromis que nous jugeons acceptable entre le surcoût du mécanisme

de livraison et l'expérience utilisateur.

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

La troisième contrainte spécifiée par le modèle de livraison est qu'une opération doit être livrée après l'opération *rename* qui a introduite son époque de génération.

Pour cela, le module de livraison doit donc récupérer l'époque courante de la séquence répliquée, récupérer le dot de l'opération *rename* l'ayant introduite et l'ajouter en tant que dépendance de chaque opération. Cependant, dans notre implémentation, le module de livraison et le module représentant la séquence répliquée sont découplés et ne peuvent interagir directement l'un avec l'autre.

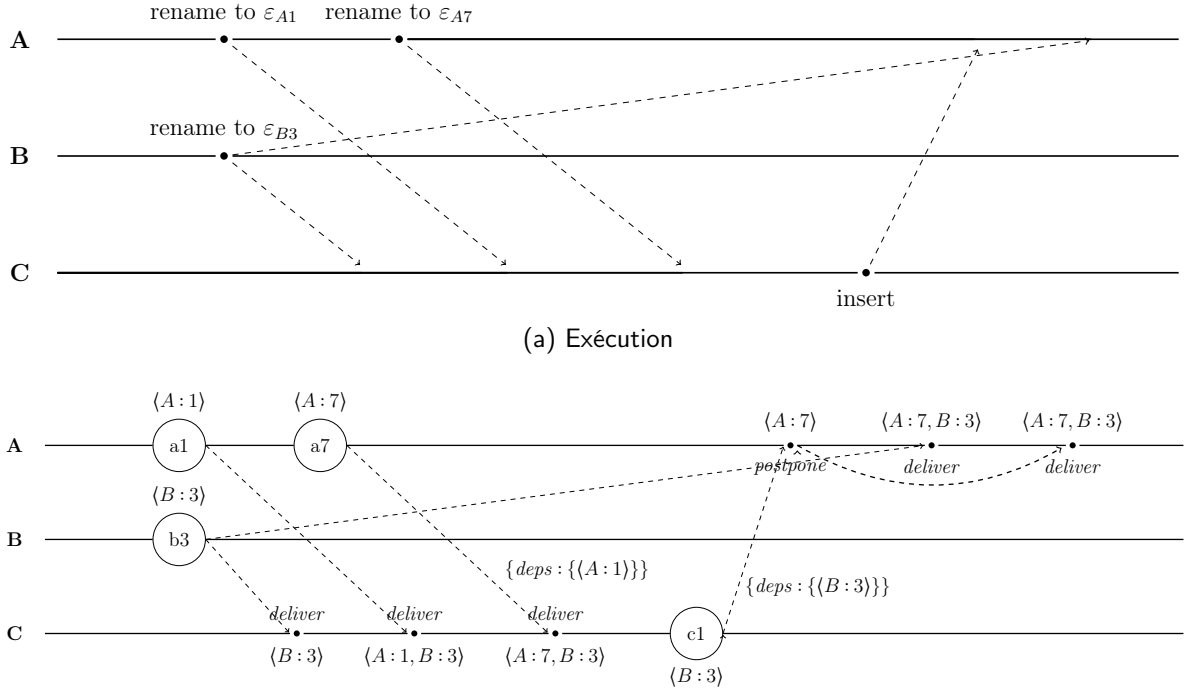
Pour remédier à ce problème, le module de livraison maintient une structure supplémentaire : un vecteur des dots des opérations *rename* connues. À la réception d'une opération *rename* distante, l'entrée correspondante de son auteur est mise à jour avec le dot de la nouvelle époque introduite. À la génération d'une opération locale, l'opération est examinée pour récupérer son époque de génération. Le module conserve alors seulement l'entrée correspondante dans le vecteur des dots des opérations *rename*. À ce stade, le contenu du vecteur est ajouté en tant que dépendance de l'opération. Ensuite, si l'opération locale s'avère être une opération *rename*, le vecteur est modifié pour ne conserver que le dot de l'époque introduite par l'opération. La Figure 3.8 illustre ce fonctionnement.

Dans la Figure 3.8a, nous décrivons une exécution suivante en ne faisant apparaître que les opérations importantes : les opérations *rename* et une opération *insert* finale. Dans cette exécution, trois noeuds A, B et C répliquent et éditent collaborativement une séquence. Initialement, aucune opération *rename* n'a encore eu lieu. Le noeud A effectue une première opération *rename* (*a1*) puis une seconde opération *rename* (*a7*), et les diffuse. En concurrence, le noeud B génère et propage sa propre opération *rename* (*b3*). De son côté, le noeud C reçoit les opérations *b3*, puis *a1* et *a7*. Il émet ensuite une opération *insert* (*c1*). Le noeud A reçoit cette opération avant de finalement recevoir l'opération *b3*.

Dans la Figure 3.8b, nous faisons apparaître l'état du module de livraison et les décisions prises par ce dernier au cours de l'exécution. Initialement, le vecteur des dots des opérations *rename* connues est vide. Ainsi, lorsque A génère l'opération *a1*, celle-ci ne se voit ajouter aucune dépendance (nous ne représentons pas les dépendances des opérations qui correspondent à l'ensemble vide). A met ensuite à jour son vecteur des dots des opérations *rename* avec le dot $\langle A : 1 \rangle$. B procède de manière similaire avec l'opération *b3*.

Quand A génère l'opération *a7*, le dot $\langle A : 1 \rangle$ est ajouté en tant que dépendance. Le dot $\langle A : 7 \rangle$ remplace ensuite ce dernier dans le vecteur des dots des opérations *rename*.

À la réception de l'opération *b3*, le module de livraison de C peut la livrer au CRDT, l'ensemble de ses dépendances étant vérifié. Le noeud C ajoute alors à son vecteur des dots des opérations *rename* le dot $\langle B : 3 \rangle$. Il procède de même pour l'opération *a1* : il la livre et ajoute le dot $\langle A : 1 \rangle$. Le module de livraison ne connaissant pas l'époque courante



(a) Exécution

(b) État et comportement du module de livraison au cours de l'exécution décrite en Figure 3.8a

FIGURE 3.8 – Gestion de la livraison des opérations après l'opération *rename* qui introduit leur époque

de la séquence répliquée, il maintient les deux dots localement.

Lorsque le noeud C reçoit l'opération *a7*, l'ensemble de ses contraintes est vérifié : l'opération *a1* a été livrée précédemment. L'opération est donc livrée et le vecteur de dots des opérations *rename* mis à jour avec $\langle A:7\rangle$.

Quand le noeud C effectue l'opération locale *c1*, le module de livraison obtient l'information de l'époque courante de la séquence : ε_{b3}. C met à jour son vecteur de dots des opérations *rename* pour ne conserver que l'entrée du noeud B : $\langle B:3\rangle$. Ce dot est ajouté en tant que dépendance de l'opération *c1* avant sa diffusion.

À la réception de l'opération *c1* par le noeud A, cette opération est mise en attente par le module de livraison, l'opération *b3* n'ayant pas encore été livrée. Le noeud reçoit ensuite l'opération *b3*. Son vecteur des dots des opérations *rename* est mis à jour et l'opération livrée. Les conditions pour l'opération *c1* étant désormais remplies, l'opération est alors livrée.

Cette implémentation de la contrainte de la livraison *epoch-based* dispose de plusieurs avantages : sa complexité spatiale dépend linéairement du nombre de noeuds et les opérations de mise à jour du vecteur des dots des opérations *rename* s'effectuent en temps constant. De plus, seul un dot est ajouté en tant que dépendance des opérations, la taille du vecteur des dots étant ramené à 1 au préalable. Finalement, cette implémentation ne contraint pas une livraison causale des opérations *rename* et permet donc de les appliquer dès que possible.

3.4.4 Livraison des opérations à terme

La contrainte restante du modèle de livraison précise que toutes les opérations doivent être livrées à l'ensemble des noeuds à terme. Cependant, le réseau étant non-fiable, des messages peuvent être perdus au cours de l'exécution. Il est donc nécessaire que les noeuds rediffusent les messages perdus pour assurer leur livraison à terme.

Pour cela, nous implémentons un mécanisme d'anti-entropie basé sur [57]. Ce mécanisme permet à un noeud source de se synchroniser avec un autre noeud cible. Il est executé par l'ensemble des noeuds de manière indépendante. Nous décrivons ci-dessous son fonctionnement.

De manière périodique, le noeud choisit un autre noeud cible de manière aléatoire. Le noeud source lui envoie alors une représentation de son état courant, c.-à-d. son vecteur de version.

À la réception de ce message, le noeud cible compare le vecteur de version reçu par rapport à son propre vecteur de version. À partir de ces données, il identifie les dots des opérations de sa connaissance qui sont inconnues au noeud source. Grâce à leur dot, le noeud cible retrouve ces opérations depuis son log des opérations. Il envoie alors une réponse composée de ces opérations au noeud source.

À la réception de la réponse, le noeud source intègre normalement les opérations reçues. La Figure 3.9 illustre ce mécanisme.

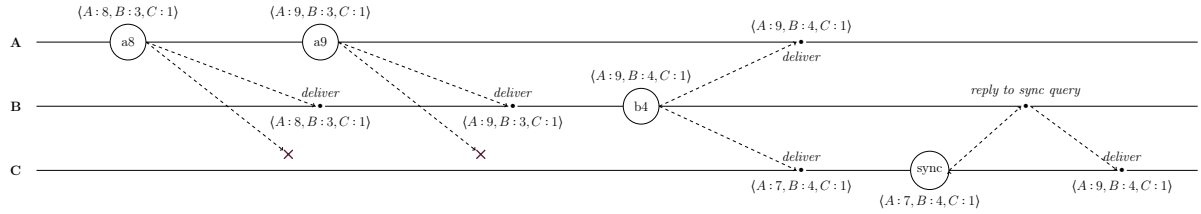


FIGURE 3.9 – Utilisation du mécanisme d'anti-entropie par le noeud C pour se synchroniser avec le noeud B

Dans cette figure, nous représentons une exécution à laquelle participent trois noeuds : A, B et C. Initialement, les trois noeuds sont synchronisés. Leur vecteurs de version sont identiques et ont pour valeur $\langle A : 7, B : 3, C : 1\rangle$.

Le noeud A effectue les opérations *a8* puis *a9* et les diffusent sur le réseau. Le noeud B reçoit ces opérations et les livre à son CRDT. Il effectue ensuite et propage l'opération *b4*, qui est reçue et livrée par A. Ils atteignent tous deux la version représenté par le vecteur $\langle A : 9, B : 4, C : 1\rangle$

De son côté, le noeud C ne reçoit pas les opérations *a8* et *a9* à cause d'une défaillance réseau. Néanmoins, cela ne l'empêche pas de livrer l'opération *b4* à sa réception et d'obtenir la version $\langle A : 7, B : 4, C : 1\rangle$.

Le noeud C déclenche ensuite son mécanisme d'anti-entropie. Il choisit aléatoirement le noeud B comme noeud cible. Il lui envoie un message de synchronisation avec pour contenu le vecteur de version $\langle A : 7, B : 8, C : 1\rangle$.

À la réception de ce message, le noeud B compare ce vecteur avec le sien. Il détermine que le noeud C n'a pas reçu les opérations *a8* et *a9*. B les récupère depuis son log des opérations et les envoie à C par le biais d'un nouveau message.

À la réception de la réponse de B, le noeud C livre les opérations *a8* et *a9*. Il atteint alors le même état que A et B, représenté par le vecteur de version $\langle A:9, B:4, C:1\rangle$.

Ce mécanisme d'anti-entropie nous permet ainsi de garantir la livraison à terme de toutes les opérations et de compenser les défaillances du réseau. Il nous sert aussi de mécanisme de synchronisation : à la connexion d'un pair, celui-ci utilise ce mécanisme pour récupérer les opérations effectuées depuis sa dernière connexion. Dans le cas où il s'agit de la première connexion du pair, il lui suffit d'envoyer un vecteur de version vide pour récupérer l'intégralité des opérations.

Ce mécanisme propose plusieurs avantages. Son exécution n'implique que le noeud source et le noeud cible, ce qui limite les coûts de coordination. De plus, si une défaillance a lieu lors de l'exécution du mécanisme (perte d'un des messages, panne du noeud cible...), cette défaillance n'est pas critique : le noeud source se synchronisera à la prochaine exécution du mécanisme. Ensuite, ce mécanisme réutilise le vecteur de version déjà nécessaire pour la livraison en exactement un exemplaire, comme présenté en sous-section 3.4.1. Il ne nécessite donc pas de stocker une nouvelle structure de données pour détecter les différences entre noeuds.

En contrepartie, la principale limite de ce mécanisme d'anti-entropie est qu'il nécessite de maintenir et de parcourir périodiquement le log des opérations pour répondre aux requêtes de synchronisation. La complexité spatiale et en temps du mécanisme dépend donc linéairement du nombre d'opérations. Qui plus est, nous sommes dans l'incapacité de tronquer le log des opérations en se basant sur la stabilité causale des opérations puisque nous utilisons ce mécanisme pour mettre à niveau les nouveaux pairs. À moins de mettre en place un mécanisme de compression du log comme évoqué en **??**, ce log des opérations croit de manière monotone. Néanmoins, une alternative possible est de mettre en place un système de chargement différé des opérations pour ne pas surcharger la mémoire.

3.5 Couche réseau

Pour permettre aux différents noeuds de communiquer, MUTE repose sur la librairie Netflux [26]. Développée au sein de l'équipe Coast, cette librairie permet de construire un réseau P2P entre des navigateurs, mais aussi des agents logiciels.

3.5.1 Établissement d'un réseau P2P entre navigateurs

Pour créer un réseau P2P entre navigateurs, Netflux utilise la technologie Web Real-Time Communication (WebRTC). WebRTC est une API [27] de navigateur spécifiée en 2011, et en cours d'implémentation dans les différents navigateurs depuis 2013. Elle permet

26. `https://github.com/coast-team/netflux`
27. Application Programming Interface (API) : Interface de Programmation

de créer une connexion directe entre deux navigateurs pour échanger des médias audio et/ou vidéo, ou simplement des données.

Cette API utilise pour cela un ensemble de protocoles. Ces protocoles réintroduisent des serveurs dans l'architecture système de MUTE. Dans la Figure 3.10, nous représentons un réseau P2P créé avec WebRTC et les différents serveurs impliqués.

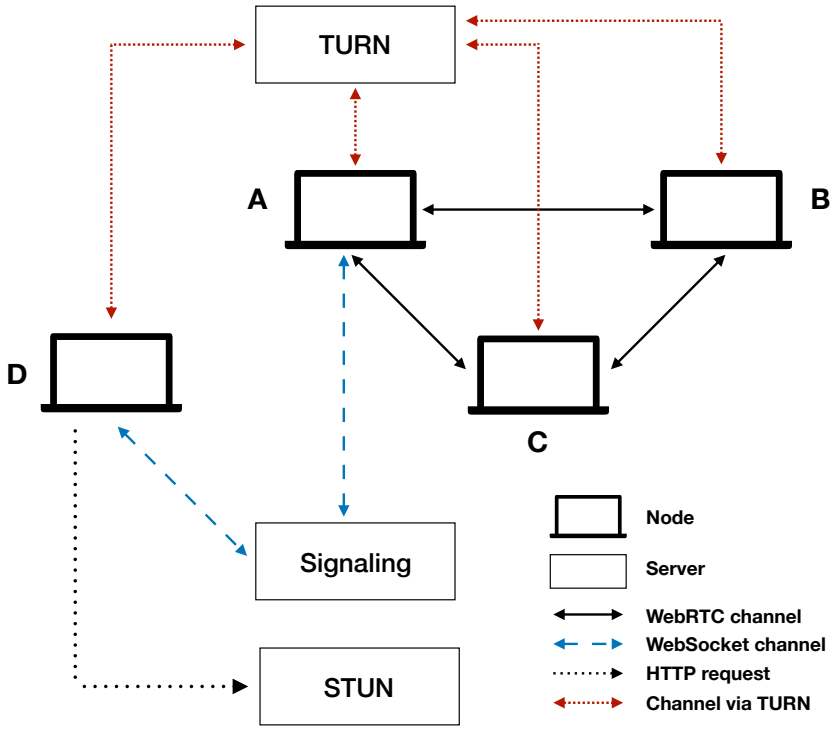


FIGURE 3.10 – Architecture système pour la couche réseau de MUTE

Nous décrivons ci-dessous leur rôle respectif dans la collaboration.

Serveur de signalisation

Pour rejoindre un réseau P2P déjà établi, un nouveau noeud a besoin de découvrir les noeuds déjà connectés et de pouvoir communiquer avec eux. Le serveur de signalisation offre ces fonctionnalités.

Au moins un noeud du réseau P2P doit maintenir une connexion avec le serveur de signalisation. À sa connexion, un nouveau noeud contacte le serveur de signalisation. Il est mis en relation avec un noeud du réseau P2P par son intermédiaire et échange les différents messages de WebRTC nécessaires à l'établissement d'une connexion P2P entre eux.

Une fois cette première connexion P2P établie, le nouveau noeud contacte et communique avec les autres noeuds par l'intermédiaire du premier noeud. Il peut alors terminer sa connexion avec le serveur de signalisation.

Serveur STUN

Pour se connecter, les noeuds doivent s'échanger plusieurs informations logicielles et matérielles, notamment leur adresse IP publique respective. Cependant, un noeud n'a pas accès à cette donnée lorsque son routeur utilise le protocole NAT. Le noeud doit alors la récupérer.

Pour permettre aux noeuds de découvrir leur adresse IP publique, WebRTC repose sur le protocole STUN. Ce protocole consiste simplement à contacter un serveur tiers dédié à cet effet. Ce serveur retourne en réponse au noeud qui le contacte son adresse IP publique.

Serveur TURN

Il est possible que des noeuds provenant de réseaux différents ne puissent établir une connexion P2P directe entre eux, par exemple à cause de restrictions imposées par leur pare-feux respectifs. Pour contourner ce cas de figure, WebRTC utilise le protocole TURN.

Ce protocole consiste à utiliser un serveur tiers comme relais entre les noeuds. Ainsi, les noeuds peuvent communiquer par son intermédiaire tout au long de la collaboration. Les échanges sont chiffrés, afin que le serveur TURN ne représente pas une faille de sécurité.

Rôles des serveurs

Ainsi, WebRTC implique l'utilisation de plusieurs serveurs.

Les serveurs de signalisation et STUN sont nécessaires pour permettre à de nouveaux noeuds de rejoindre la collaboration. Autrement dit, leur rôle est ponctuel : une fois le réseau P2P établit, les noeuds n'ont plus besoin d'eux. Ces serveurs peuvent alors être coupés sans impacter la collaboration.

À l'inverse, les serveurs TURN jouent un rôle plus prédominant dans la collaboration. Ils sont nécessaires dès lors que des noeuds proviennent de réseaux différents et sont alors requis tout au long de la collaboration. Une panne de ces derniers entraverait la collaboration puisqu'elle résulterait en une partition des noeuds. Il est donc primordial de s'assurer de la disponibilité et fiabilité de ces serveurs.

3.5.2 Topologie réseau

Netflux établit un réseau P2P par document. Chaque réseau P2P est un réseau entièrement maillé : chaque noeud se connecte à l'ensemble des autres noeuds.

Cette topologie simple est adaptée à des groupes de petite taille, mais ne passe pas à l'échelle. D'autres topologies limitant le nombre de connexions par noeuds, telle que celle décrite par [35], pourraient être implémentées pour adresser cette limite.

3.6 Couche sécurité

La couche sécurité a pour but de garantir l'authenticité et la confidentialité des messages échangés par les noeuds. Pour cela, elle implémente un mécanisme de chiffrement

de bout en bout.

Pour chiffrer les messages, MUTE utilise un mécanisme de chiffrement à base de clé de groupe. Le protocole choisi est le protocole Burmester-Desmedt [36]. Il nécessite que chaque noeud possède une paire de clés de chiffrement et enregistre sa clé publique auprès d'un PKI[28].

Afin d'éviter qu'un PKI malicieux n'effectue une attaque de l'homme au milieu sur la collaboration, les noeuds doivent vérifier le bon comportement des PKI de manière non-coordonnée. À cet effet, MUTE implémente le mécanisme d'audit de PKI Trusternity [31, 32]. Son fonctionnement nécessite l'utilisation d'un registre publique sécurisé *append-only*, c.-à-d. une blockchain.

L'architecture système nécessaire pour la couche sécurité est présentée dans la Figure 3.11.

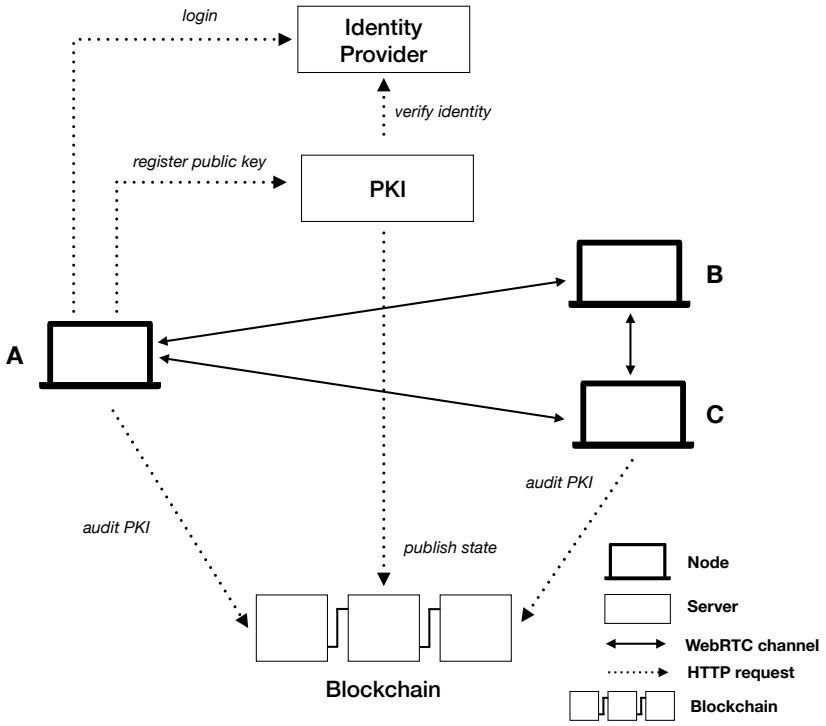


FIGURE 3.11 – Architecture système pour la couche sécurité de MUTE

Cette couche sécurité s'ajoute au mécanisme de chiffrement des messages inhérent à WebRTC. Cela nous offre de nouvelles possibilités : plutôt que de créer un réseau P2P par document, nous pouvons désormais mettre en place un réseau P2P global. Les messages étant chiffrés de bout en bout, les noeuds peuvent communiquer en toute sécurité et confidentialité par l'intermédiaire de noeuds tiers, c.-à-d. des noeuds extérieurs à la collaboration.

Une limite de l'approche actuelle est que la clé de groupe change avec l'évolution des noeuds connectés : à chaque connexion ou déconnexion d'un noeud, une nouvelle clé est recalculée avec les collaborateur-rices présents. Cette évolution fréquente de la clé de

28. Public Key Infrastructure (PKI) : Infrastructure de gestion de clés

chiffrement, nécessaire pour garantir la *backward secrecy* et *forward secrecy*, nous empêche par exemple de stocker les opérations de manière chiffrée chez des noeuds tiers. Cette fonctionnalité serait cependant bien pratique pour permettre à un noeud de récupérer la dernière version de ses documents, même en l'absence de ses collaborateur-rices. Une autre clé de chiffrement, dédiée au stockage, devrait être mise en place, ainsi qu'un mécanisme de découverte des noeuds tiers stockant les données de la collaboration.

3.7 Conclusion

Dans ce chapitre, nous avons présenté Multi User Text Editor (MUTE), notre éditeur collaboratif temps réel P2P chiffré de bout en bout.

MUTE permet d'éditer de manière collaborative des documents texte. Pour représenter les documents, MUTE implémente les structures de données répliquées décrites dans la section 2.4 et le **??**. Ces CRDTs offrent de nouvelles méthodes de collaborer, notamment en permettant de collaborer de manière synchrone ou asynchrone de manière transparente.

Pour permettre aux noeuds de communiquer, MUTE utilise WebRTC. Cette technologie permet de construire un réseau P2P entre navigateurs. Plusieurs serveurs sont néanmoins requis, notamment pour la découverte des pairs et pour la communication entre des noeuds dont les pare-feux respectifs empêche l'établissement d'une connexion directe.

Finalement, MUTE implémente un mécanisme de chiffrement de bout en bout garantissant l'authenticité et la confidentialité des échanges entre les noeuds. Ce mécanisme reposant sur d'autres serveurs, les PKIs, MUTE intègre un mécanisme d'audit permettant de détecter leurs éventuels comportements malicieux.

Chapitre 4

Conclusions et perspectives

Sommaire

4.1 Résumés des contributions 95
4.1.1 Réflexions sur l'état de l'art des CRDTs 95
4.1.2 Ré-identification sans coordination pour les CRDTs pour Séquence 97
4.1.3 Éditeur de texte collaboratif P2P chiffré de bout en bout 99
4.2 Perspectives 101
4.2.1 Définition de relations de priorité pour minimiser les traitements 101
4.2.2 Détection et fusion manuelle de versions distantes 102
4.2.3 Étude comparative des différents modèles de synchronisation pour CRDTs 106
4.2.4 Approfondissement du patron de conception de Pure Operation-Based CRDTs 108

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

4.1 Résumés des contributions

4.1.1 Réflexions sur l'état de l'art des CRDTs

Les Conflict-free Replicated Data Types (CRDTs) [22] sont de nouvelles spécifications des types de données. Ils sont conçus pour permettre à un ensemble de noeuds d'un système de répliquer une même donnée et pour leur permettre de la consulter et de la modifier sans aucune coordination préalable. Les noeuds se synchronisent

L'absence de coordination entre les noeuds avant modifications implique que des noeuds peuvent modifier la donnée en concurrence. De telles modifications peuvent donner lieu à des conflits, e.g. l'ajout et la suppression en concurrence d'un même élément dans un ensemble. Pour pallier ce problème, les CRDTs incorporent un mécanisme de résolution de conflits automatiques directement au sein de leur spécification.

Il convient de noter qu'il existe plusieurs solutions possibles pour résoudre un conflit. Pour reprendre l'exemple de l'élément ajouté et supprimé en concurrence d'un ensemble, nous pouvons par exemple soit le conserver l'élément, soit le supprimer. Nous parlons alors de sémantique du mécanisme de résolution de conflits automatique.

De la même manière, il existe plusieurs approches possibles pour synchroniser les noeuds, e.g. diffuser chaque modification de manière atomique ou diffuser l'entièreté de l'état périodiquement. Ainsi, lors de la définition d'un CRDT, il convient de préciser les sémantiques de résolution de conflits qu'il adopte et le modèle de synchronisation qu'il utilise [42].

Depuis leur formalisation, les travaux sur les CRDTs ont abouti à la conception de nouveaux, soit en spécifiant de nouvelles sémantiques de résolution de conflits pour un type de données [47], soit en spécifiant de nouveaux modèles de synchronisation [52] ou en enrichissant les spécifications des modèles existants [50, 63].

Dans notre présentation des CRDTs (cf. section 2.2, page 13), nous présentons chacun de ces aspects. Cependant, nous nous ne limitons pas à retranscrire l'état de l'art de la littérature. Notamment au sujet du modèle de synchronisation par opérations, nous précisons que le modèle de livraison causal n'est pas nécessaire pour l'ensemble des CRDTs synchronisés par opérations, c.-à-d. que certains peuvent adopter des modèles de livraison moins contraints et donc moins coûteux. Cette précision nous permet de proposer une étude comparative des différents modèles de synchronisation qui est, à notre connaissance, l'une des plus précises de la littérature (cf. section 2.2.2, page 28).

Nous présentons ensuite les différents CRDTs pour le type Séquence de la littérature (cf. section 2.3, page 30). Nous mettons alors en exergue les deux approches proposées pour concevoir le mécanisme de résolution de conflits automatiques pour le type Séquence, c.-à-d. l'approche à pierres tombales et l'approche à identifiants densément ordonnés. De nouveau, cette rétrospective nous permet d'expliciter des angles morts des articles d'origine, notamment vis-à-vis des modèle de livraison des opérations des CRDTs proposés. Puis, nous mettons en lumière les limites des évaluations comparant les deux approches, c.-à-d. le couplage entretenu entre approche du mécanisme de résolution de conflits et choix d'implémentations. Cette limite empêche d'établir la supériorité d'une des approches par rapport à l'autre. Finalement, nous conjecturons que le surcoût de ces deux approches est le même, c.-à-d. le coût nécessaire à la représentation d'un espace dense. Nous précisons dès lors par le biais de notre propre étude comparative comment ce surcoût s'exprime dans chacune des approches , c.-à-d. le compromis entre surcoût en métadonnées, calculs et bande-passante proposé par les deux approches (cf. sous-section 2.3.3, page 49).

Ces réflexions que nous présentons sur l'état des CRDTs définissent plusieurs pistes de recherches. Une première d'entre elles concerne notre étude comparative des modèles de

synchronisation. D'après les critères que nous utilisons, une conclusion possible de cette comparaison est que le modèle de synchronisation par différences d'états rend obsolètes les modèles de synchronisation par états et par opérations. En effet, le modèle de synchronisation par différences d'états apparaît comme adapté à l'ensemble des contextes d'utilisation qui étaient jusqu'à lors exclusifs à ces derniers, de par les multiples stratégies qu'il permet, e.g. synchronisation par états complets, synchronisation par états irréductibles...

Cette conclusion nous paraît cependant hâtive. Il convient d'étendre notre étude comparative pour prendre en compte des critères de comparaison additionnels pour confirmer cette conjecture, ou l'invalider et définir plus précisément les spécificités de chacun des modèles de synchronisation. Nous détaillons cette piste de recherche dans la sous-section 4.2.3.

Une seconde piste de recherche possible concerne les deux approches utilisées pour concevoir le mécanisme de résolution de conflits des CRDTs pour le type Séquence. Comme dit précédemment, nous conjecturons que ces deux approches ne sont finalement que deux manières différentes de représenter une même information : la position d'un élément dans un espace dense. La différence entre ces approches résiderait uniquement dans la manière que chaque représentation influe sur les performances du CRDT. Une piste de travail serait donc de confirmer cette conjecture, en proposant une formalisation unique des CRDTs pour le type Séquence.

4.1.2 Ré-identification sans coordination pour les CRDTs pour Séquence

Pour privilégier leur disponibilité, latence et tolérance aux pannes, les systèmes distribués peuvent adopter le paradigme de la réplication optimiste [16]. Ce paradigme consiste à relaxer la cohérence de données entre les noeuds du système pour leur permettre de consulter et modifier leur copie locale sans se coordonner. Leur copies peuvent alors temporairement diverger avant de converger de nouveau une fois les modifications de chacun propagées. Cependant, cette approche nécessite l'emploi d'un mécanisme de résolution de conflits pour assurer la convergence même en cas de modifications concurrentes. Pour cela, l'approche des CRDTs [21, 22] propose d'utiliser des types de données dont les modifications sont nativement commutatives.

Depuis la spécification des CRDTs, la littérature a proposé plusieurs de ces mécanismes résolution de conflits automatiques pour le type de données Séquence [74, 75, 76, 72]. Cependant, ces approches souffrent toutes d'un surcoût croissant de manière monotone. Ce problème a été identifié par la communauté, et celle-ci a proposé pour y répondre des mécanismes permettant soit de réduire la croissance du surcoût [94, 95], soit d'effectuer une GC du surcoût [75, 23, 24]. Nous avons cependant déterminé que ces mécanismes ne sont pas adaptés aux systèmes P2P à large échelle souffrant de churn et utilisant des CRDTs pour Séquence à granularité variable.

Dans le cadre de cette thèse, nous avons donc souhaité proposer un nouveau mécanisme adapté à ce type de systèmes. Pour cela, nous avons suivi l'approche proposée par [23, 24] : l'utilisation d'un mécanisme pour ré-assigner de nouveaux identifiants aux élements stockées dans la séquence. Nous avons donc proposé un nouveau mécanisme appartenant

à cette approche pour le CRDT LogootSplit [28].

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

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

Pour valider notre approche, nous proposons une évaluation expérimentale de cette dernière. Cette évaluation se base sur des traces de sessions d'édition collaborative que nous avons généré par simulations. Chacune de ces simulations représente la rédaction collaborative d'un document texte par 10 noeuds.

Notre évaluation nous permet de valider de manière empirique les résultats attendus. Le premier d'entre eux concerne la convergence des noeuds. En effet, nos simulations nous ont permis de valider que l'ensemble des noeuds obtenaient des états finaux équivalents, même en cas d'opérations *ren* concurrentes.

Notre évaluation nous a aussi permis de valider que le mécanisme de renommage réduit à une taille minimale le surcoût du mécanisme de résolution de conflits incorporé dans le CRDT pour Séquence.

L'évaluation expérimentale nous a aussi permis de prendre conscience d'effets additionnels du mécanisme de renommage que nous n'avions pas anticipé. Notamment, elle montre que le surcoût éventuel du mécanisme de renommage, notamment en termes de calculs, est toutefois contrebalancé par l'amélioration précisée précédemment, c.-à-d. la réduction de la taille de la séquence.

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

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

(i) La définition de stratégies de déclenchement du renommage efficaces. Le but de ces stratégies serait de déclencher le mécanisme de renommage de manière fréquente, de façon à garder son temps d'exécution acceptable, mais tout visant à minimiser la probabilité que les noeuds produisent des opérations *ren* concurrentes, de façon à minimiser le surcoût en métadonnées.

(ii) La définition de relations *priority* efficaces. Nous développons ce point dans la sous-section 4.2.1.

(iii) La proposition d'algorithmes de renommage efficaces. Cette amélioration peut prendre la forme de nouveaux algorithmes pour `renameId` et `revertRenameId` offrant une meilleure complexité en temps. Il peut aussi s'agir de la conception d'une nouvelle approche pour renommer l'état et gérer les modifications concurrentes, e.g. un mécanisme de renommage basé sur le journal des opérations (cf. **??**, page **??**).

Une seconde limite de RenamableLogootSplit que nous identifions concerne son mécanisme de GC des métadonnées introduites par le mécanisme de renommage. En effet, pour fonctionner, ce dernier repose sur la stabilité causale des opérations *ren*. Pour rappel, la stabilité causale représente le contexte causal commun à l'ensemble des noeuds du système. Pour le déterminer, chaque noeud doit récupérer le contexte causal de l'ensemble des noeuds du système. Ainsi, l'utilisation de la stabilité causale comme pré-requis pour la GC de métadonnées constitue une contrainte forte, voire prohibitive, dans les systèmes P2P à large échelle sujet au churn. En effet, un noeud du système déconnecté de manière définitive suffit pour empêcher la stabilité causale de progresser, son contexte causal étant alors indéterminé du point de vue des autres noeuds. Il s'agit toutefois d'une limite récurrente des mécanismes de GC distribués et asynchrones [75, 50, 107]. Nous présentons une piste de travail possible pour pallier ce problème dans la sous-section 4.2.2.

4.1.3 Éditeur de texte collaboratif P2P chiffré de bout en bout

Les applications collaboratives permettent à des utilisateur-rices de réaliser collaborativement une tâche. Elles permettent à plusieurs utilisateur-rices de consulter la version actuelle du document, de la modifier et de partager leurs modifications avec les autres. Ceci permet de mettre en place une réflexion de groupe, ce qui améliore la qualité du résultat produit [96, 7].

Cependant, les applications collaboratives sont historiquement des applications centralisées, e.g. Google Docs [25]. Ce type d'architecture induit des défauts d'un point de vue technique, e.g. faible capacité de passage à l'échelle et faible tolérance aux pannes,

mais aussi d'un point de vue utilisateur, e.g. perte de la souveraineté des données et absence de garantie de pérennité.

Les travaux de l'équipe Coast s'inscrivent dans une mouvance souhaitant résoudre ces problèmes et qui a conduit à la définition d'un nouveau paradigme d'applications : les LFS [14]. Le but de ce paradigme est la conception d'applications collaboratives, P2P, pérennes et rendant la souveraineté de leurs données aux utilisateur-rices.

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

Dans le cadre de cette thèse, nous avons implémenté dans MUTE nos travaux de recherche portant sur le nouvel CRDT pour le type Séquence : RenamableLogootSplit. MUTE a aussi servi à l'équipe pour présenter ses travaux concernant l'authentification des utilisateur-rices dans un système P2P [32]. Finalement, MUTE nous a permis de nous d'étudier et/ou de présenter les travaux de recherche existants concernant :

(i) Les protocoles distribués d'appartenance au groupe [33].

(ii) Les mécanismes d'anti-entropie [57].

(iii) Les protocoles d'établissement de clés de chiffrement de groupe [36].

(iv) Les protocoles d'établissement de topologies réseaux efficientes [35].

(v) Les mécanismes de conscience de groupe.

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

Tout d'abord, l'environnement web implique un certain nombre de contraintes, notamment au niveau des technologies et protocoles disponibles. Notamment, le protocole WebRTC repose sur l'utilisation de serveurs de signalisation, c.-à-d. de points de rendez-vous des pairs, et de serveurs de relai, c.-à-d. d'intermédiaires pour communiquer entre pairs lorsque les configurations de leur réseaux respectifs interdisent l'établissement d'une connexion directe. Ainsi, les applications P2P web doivent soit déployer et maintenir leur propre infrastructure de serveurs, soit reposer sur une infrastructure existante, e.g. celle proposée par OpenRelay [108]. Afin de minimiser l'effort requis aux applications P2P et la confiance exigée à leurs utilisateur-rices, nous devons supporter la mise en place d'une telle infrastructure transparente et pérenne.

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

effectuer une synchronisation. Notre collaborateur était donc dans l'incapacité de récupérer l'état et d'effectuer sa tâche. Afin de pallier ce problème, une solution possible est de faire reposer MUTE sur un réseau P2P global, e.g. le réseau de InterPlanetary File System (IPFS) [109], et d'utiliser les pairs de ce dernier, potentiellement des pairs étrangers à l'application, comme pairs de stockage pour permettre une synchronisation future. Cette solution limite ainsi le risque qu'un pair ne puisse récupérer l'état du document faute de pairs disponibles. Cependant, elle nécessite de mettre en place un mécanisme de réplication de données additionnel. Ce mécanisme de réplication sur des pairs additionnels doit cependant garantir qu'il n'introduit pas de vulnérabilités, e.g. la possibilité pour les pairs de stockage selectionnés de reconstruire et consulter le document.

4.2 Perspectives

4.2.1 Définition de relations de priorité pour minimiser les traitements

Dans la **??**, nous avons spécifié la relation *priority* (cf. **??**, page **??**). Pour rappel, cette relation doit établir un ordre strict total sur les époques de notre mécanisme de renommage.

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

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

(i) Réduire la distance entre les époques courantes des noeuds, et ainsi minimiser le surcoût en calculs par opération du mécanisme de renommage. En effet, il n'est pas nécessaire de transformer une opérations livrée avant de l'intégrer si celle-ci provient de la même époque que le noeud courant.

(ii) Définir un nouveau Plus Petit Ancêtre Commun (PPAC) entre les époques courantes des noeuds. Cela permet aux noeuds d'appliquer le mécanisme de GC pour supprimer les époques devenues obsolètes et leur anciens états associés, pour ainsi minimiser le surcoût en métadonnées du mécanisme de renommage.

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

(i) Sa simplicité.

(ii) Son surcoût limité, c.-à-d. cette approche n'introduit pas de métadonnées supplémentaires à stocker et diffuser, et l'algorithme de comparaison utilisé est simple.

(iii) Sa propriété arrangeante sur les déplacements des noeuds dans l'arbre des époques. De manière plus précise, cette définition de *priority* impose aux noeuds de se déplacer que vers l'enfant le plus à droite de l'arbre des époques. Ceci empêche les noeuds de faire un aller-retour entre deux époques données. Cette propriété permet de passer outre une contrainte concernant le couple de fonctions `renameId` et `revertRenameId` : leur reciprocité.

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

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

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

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

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

4.2.2 Détection et fusion manuelle de versions distantes

À l'issue de cette thèse, nous identifions deux limites des mécanismes de résolution de conflits automatiques dans les systèmes P2P à large échelle sujets au churn :

(i) La croissance continue de leur surcoût dûe à l'utilisation du contexte causal.

(ii) Leur compréhension limitée des intentions des utilisateur-rices, ce qui peut conduire à la génération d'anomalies lors de l'intégration de modifications.

Nous conjecturons que la probabilité que l'intégration de modifications résulte en des anomalies et que le travail nécessaire par les utilisateur-rices pour corriger ces anomalies

augmentent avec la distance entre la version de génération des modifications et la version d'intégration de ces dernières. Si cette conjecture se vérifie, nous proposons d'identifier la distance seuil entre versions à partir de laquelle l'utilisation d'un mécanisme de résolution de conflit automatique donné s'avère inefficace. Dès lors, nous proposons de recourir à un mécanisme d'intégration manuelle des modifications lorsque des modifications reçues sont originaires d'une version du document se trouvant au-delà de ce seuil.

Le recours à un mécanisme d'intégration manuelle des modifications permettrait alors :

(i) Prévenir la génération d'anomalies entravant la collaboration.

(ii) Supprimer les métadonnées nécessaires pour intégrer automatiquement des modifications originaires d'une version au-delà de la distance seuil, celles-ci n'étant dès lors plus requises puisque ces modifications seront intégrées manuellement. Par exemple, dans le cadre de RenamableLogootSplit, ce mécanisme nous permettrait de supprimer les époques se trouvant au-delà de cette distance, et ainsi leurs métadonnées associées.

Ainsi, cette approche nous permettrait de répondre aux deux problèmes identifiés précédemment.

Pour expliciter cette approche, revenons sur l'utilisation du contexte causal par les mécanismes de résolution de conflits automatiques. Le contexte causal est utilisé par les mécanismes de résolution de conflits automatiques pour :

(i) Satisfaire le modèle de cohérence causale, c.-à-d. assurer que si nous avons deux modifications m_1 et m_2 telles que $m1 \rightarrow m_2$, alors l'effet de m_2 supplantera celui de m_1. Ceci permet d'éviter des anomalies de comportement de la part de la structure de données du point de vue des utilisateur-rices, par exemple la résurgence d'un élément supprimé au préalable.

(ii) Permettre de préserver l'intention d'une modification malgré l'intégration préalable de modifications concurrentes.

Les mécanismes de résolution de conflits automatiques maintiennent donc le contexte causal de l'état de la donnée. Le contexte causal peut être représenté de différentes manières. Par exemple, le contexte causal peut prendre la forme du journal des opérations auquel on associe soit le vecteur de version correspondant à l'état de la donnée [54, 55], soit l'ensemble des extremités du Directed Acyclic Graph (DAG) formé par les opérations [56]. Cependant, de manière intrinsèque, le contexte causal ne fait que de croître au fur et à mesure que des modifications sont effectuées ou que des noeuds rejoignent le système, incrémentant son surcoût en métadonnées, calculs et bande-passante.

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

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

(i) Considérer tout noeud déconnecté comme déconnecté de manière définitive, et donc les ignorer dans le calcul de la stabilité causale. Cette première approche permet à la stabilité causale de progresser, et ainsi aux noeuds connectés de travailler dans des conditions optimales. Mais elle implique cependant que les modifications potentielles des noeuds déconnectés soient perdues, c.-à-d. de ne plus pouvoir les intégrer en l'absence d'un lien entre leur contexte causal de génération et le contexte causal actuel de chaque autre noeud. Il s'agit là de la stratégie la plus aggressive en terme de GC du contexte causal.

(ii) Assurer en toutes circonstances la capacité d'intégration des modifications des noeuds, même ceux déconnectés. Cette seconde approche permet de garantir que les modifications potentielles des noeuds déconnectés pourront être intégrées automatiquement, dans l'éventualité où ces derniers se reconnectent à terme. Mais elle implique de bloquer potentiellement de manière définitive la stabilité causale et donc le mécanisme de GC du contexte causal. Il s'agit là de la stratégie la plus timide en terme de GC du contexte causal.

La seconde limite que nous constatons est la limite des mécanismes actuels de résolution de conflits automatiques pour préserver l'intention des utilisateur-rices. Par exemple, les mécanismes de résolution de conflits automatiques pour le type Séquence présentés dans ce manuscrit (cf. section 2.3, page 30) définissent l'intention de la manière suivante : *l'intégration de la modification par les noeuds distants doit reproduire l'effet de la modification sur la copie d'origine*. Cette définition assure que chaque modification est porteuse d'une intention, mais limite voire ignore toute la dimension sémantique de la dite intention. Nous conjecturons que l'absence de dimension sémantique réduit les cas d'utilisation de ces mécanismes.

Considérons par exemple une édition collaborative d'un même texte par un ensemble de noeuds. Lors de la présence d'une faute de frappe dans le texte, e.g. le mot "HLLO", plusieurs utilisateur-rices peuvent la corriger en concurrence, c.-à-d. insérer l'élément "E" entre "H" et "L". Les mécanismes de résolution de conflits automatiques permettent aux noeuds d'obtenir des résultats qui convergent mais à notre sens insatisfaisant, e.g. "HEEEEEELLO". Nous considérons ce type de résultats comme des anomalies, au même titre que l'entrelacement [88]. Dans le cadre de collaborations temps réel à échelle limitée, nous conjecturons cependant qu'une granularité fine des modifications permet de pallier ce problème. En effet, les utilisateur-rices peuvent observer une anomalie produite par le mécanisme de résolution de conflits automatique, déduire l'intention initiale des modifications concernées et la restaurer par le biais d'actions supplémentaires de compensation.

Cependant, dans le cadre de collaborations asynchrones ou à large échelle, nous conjecturons que ces anomalies de résolution de conflits s'accumulent. Cette accumulation peut atteindre un seuil rendant laborieuse la déduction et le rétablissement de l'intention ini-

tiale des modifications. Le travail imposé aux utilisateur-rices pour résoudre ces anomalies par le biais d'actions de compensation peut alors entraver la collaboration. Pour reprendre l'exemple de l'édition collaborative de texte, nous pouvons constater de tels cas suite à de la duplication de contenu et/ou l'entrelacement de mots, phrases voire paragraphes nuisant à la clarté et correction du texte. Il convient alors de s'interroger sur le bien-fondé de l'utilisation de mécanismes de résolutions de conflits automatiques pour intégrer un ensemble de modifications dans l'ensemble des situations.

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

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

(ii) Si la distance calculée dépasse la distance seuil, c.-à-d. que nous ne disposons plus à ce stade de mécanismes de résolution de conflits automatiques satisfaisants, faire intervenir les utilisateur-rices par le biais d'un mécanisme de résolution de conflits manuel. L'utilisation d'un mécanisme manuel n'exclut cependant pas tout pré-travail de notre part pour réduire la charge de travail des utilisateur-rices dans le processus de fusion.

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

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

Pour mener à bien ce travail, il conviendra tout d'abord de définir la notion de distance entre versions de la donnée répliquée. Nous envisageons de baser cette dernière sur les deux

aspects temporel et spatial, c.-à-d. en utilisant la distance entre contextes causaux et la distance entre contenus. Dans le cadre de l'édition collaborative, nous pourrons pour cela nous baser sur les travaux existants pour évaluer la distance entre deux textes. *Matthieu: TODO : Insérer refs distance de Hamming, Levenstein, String-to-string correction problem (Tichy et al)*

Il conviendra ensuite de déterminer comment établir la valeur seuil à partir de laquelle la distance entre versions est jugée trop importante. Les approches d'évaluation de la qualité du résultat [110] pourront être utilisées pour déterminer un couple ⟨ méthode de calcul de la distance, valeur de distance ⟩ spécifiant les cas pour lesquels les méthodes de résolution de conflits automatiques ne produisent plus un résultat satisfaisant. *Matthieu: TODO : Insérer refs travaux Claudia et Vinh* Le couple obtenu pourra ensuite être confirmé par le biais d'expériences utilisateurs inspirées de [92, 93].

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

4.2.3 Étude comparative des différents modèles de synchronisation pour CRDTs

Comme évoqué dans l'état de l'art (cf. section 2.2.2, page 26), un nouveau modèle de synchronisation pour CRDT fut proposé récemment [51]. Ce dernier propose une synchronisation des noeuds par le biais de différences d'états. Dans notre étude comparative des différents modèles de synchronisation (cf. section 2.2.2, page 28), nous avons justifié que ce modèle de synchronisation est adapté à l'ensemble des contextes d'utilisation qui étaient jusqu'à lors exclusifs soit au modèle de synchronisation par états, soit par opérations. Dans cette piste de recherche, nous souhaitons approfondir notre étude comparative pour déterminer si le modèle de synchronisation par différences d'états rend obsolètes les modèles de synchronisation précédents.

Pour rappel, ce nouveau modèle de synchronisation se base sur le modèle de synchronisation par états. Il partage les mêmes pré-requis, à savoir la nécessité d'une fonction `merge` associative, commutative et idempotente. Cette dernière doit permettre de la fusion toute paire d'états possible en calculant leur borne supérieure, c.-à-d. leur LUB [41].

La spécificité de ce nouveau modèle de synchronisation est de calculer pour chaque modification la différence d'état correspondante. Cette différence correspond à un élément irréductible du sup-demi-treillis du CRDT [63], c.-à-d. un état particulier de ce dernier. Cet élément irréductible peut donc être diffusé et intégré par les autres noeuds, toujours à l'aide de la fonction `merge`.

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

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

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

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

(i) La composition de CRDTs, c.-à-d. la capacité de combiner et de mettre en relation plusieurs CRDTs au sein d'un même système, afin d'offrir des fonctionnalités plus complexes. Par exemple, une composition de CRDTs peut se traduire par l'ajout de dépendances entre les modifications des différents CRDTs composés. Le modèle de synchronisation par opérations nous apparaît plus adapté pour cette utilisation, de par le découplage qu'il induit entre les CRDTs et la couche de livraison de messages.

(ii) L'utilisation de CRDTs au sein de systèmes non-sûrs, c.-à-d. pouvant compter un ou plusieurs adversaires byzantins [111]. Dans de tels systèmes, les adversaires byzantins peuvent générer des modifications différentes mais qui sont perçues comme identiques par les mécanismes de résolution de conflits. Cette attaque, nommée *équivoque*, peut provoquer la divergence définitive des copies. [107] propose une solution adaptée aux systèmes P2P à large échelle. Celle-ci se base notamment sur l'utilisation de journaux infalsifiables. *Matthieu: TODO : Ajouter refs* Il convient alors d'étudier si l'utilisation de journaux infalsifiables ne limite pas le potentiel du modèle de synchronisation par différences d'états, e.g. en interdisant la diffusion des modifications par états complets.

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

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

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

Plusieurs approches ont été proposées dans la littérature pour guider la conception de CRDTs :

(i) L'utilisation de la théorie des treillis pour la conception de CRDTs synchronisés par états et par différences d'états [22, 63].

(ii) L'utilisation d'un journal partiellement ordonné des opérations, nommé PO-Log, pour la conception de CRDTs synchronisés par opérations [50].

Cependant, ce framework proposé par [50] souffre de plusieurs limitations. Nous souhaitons donc proposer un nouveau framework pour la conception de CRDTs synchronisés par opérations, en nous basant sur ce dernier.

Le framework proposé dans [50] possède plusieurs objectifs :

(i) Proposer une approche partiellement générique pour définir un CRDT synchronisé par opérations.

(ii) Factoriser les métadonnées utilisées par le CRDT pour le mécanisme de résolution de conflits, notamment pour identifier les éléments, et celles utilisées par la couche livraison, notamment pour identifier les opérations.

(iii) Inclure des mécanismes de GC de ces métadonnées pour réduire la taille de l'état.

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

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

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

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

En plus du coût induit en termes de métadonnées et de bande-passante, le modèle de livraison causale peut aussi introduire un délai superflu dans la livraison des opérations. En effet, ce modèle impose que tous les messages précédant un nouveau message d'après la

relation *happens-before* soient eux-mêmes livrés avant de livrer ce dernier. Il en résulte que des opérations peuvent être mises en attente par la couche livraison, e.g. suite à la perte d'une de leurs dépendances d'après la relation *happens-before*, alors que leurs dépendances réelles ont déjà été livrées et que les opérations sont de fait intégrables en l'état. Plusieurs travaux [112, 61] ont noté ce problème. Pour y répondre et ainsi améliorer la réactivité du framework Pure Operation-Based, ils proposent d'exposer les opérations mises en attente par la couche livraison au CRDT. Bien que fonctionnelle, cette approche induit toujours le coût d'une couche de livraison respectant le modèle de livraison causale et nous fait considérer la raison de ce coût, le modèle de livraison n'étant dès lors plus respecté.

Ensuite, ce framework impose que la modification **prepare** ne puisse pas inspecter l'état courant du noeud. Cette contrainte est compatible avec les CRDTs pour les types de données simples qui sont considérés dans [50], e.g. le Compteur ou l'Ensemble. Elle empêche cependant l'expression de CRDTs pour des types de données plus complexes, e.g. la Séquence ou le Graphe. *Matthieu: TODO : À confirmer pour le graphe* Nous jugeons dommageable qu'un framework pour la conception de CRDTs limite de la sorte son champ d'application.

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

Pour répondre aux différents problèmes soulevés, nous souhaitons proposer un nouveau framework en nous basant sur [50]. Nos objectifs sont les suivants :

(i) Proposer un framework mettant en lumière la présence et le rôle de deux modèles de livraison :
  (i) Le modèle de livraison minimal requis par le CRDT pour assurer la convergence forte à terme [22].
  (ii) Le modèle de livraison employé par le système qui utilise le CRDT. Ce second modèle de livraison est une stratégie permettant au système de respecter un modèle de cohérence donné et régissant les règles de compaction de l'état. Il doit être égal ou plus contraint que modèle de livraison minimal du CRDT et peut être amené à évoluer en fonction de l'état du système et de ses besoins. Par exemple, un système pourrait par défaut utiliser le modèle de livraison causale pour assurer le modèle de cohérence causal. Puis, lorsque le nombre de noeuds atteint un seuil donné et que le coût de la livraison causale devient trop élevé, le système pourrait passer au modèle de livraison FIFO pour assurer le modèle de cohérence PRAM afin de réduire les coûts en bande-passante.

(ii) Étendre la notion de redondance causale pour prendre en compte la redondance

partielle des opérations. De plus, nous souhaitons rendre cette notion accessible à la couche de livraison, pour détecter au plus tôt les opérations désormais obsolètes et prévenir leur diffusion.

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

Annexe A

Entrelacement d'insertions concurrentes dans Treedoc

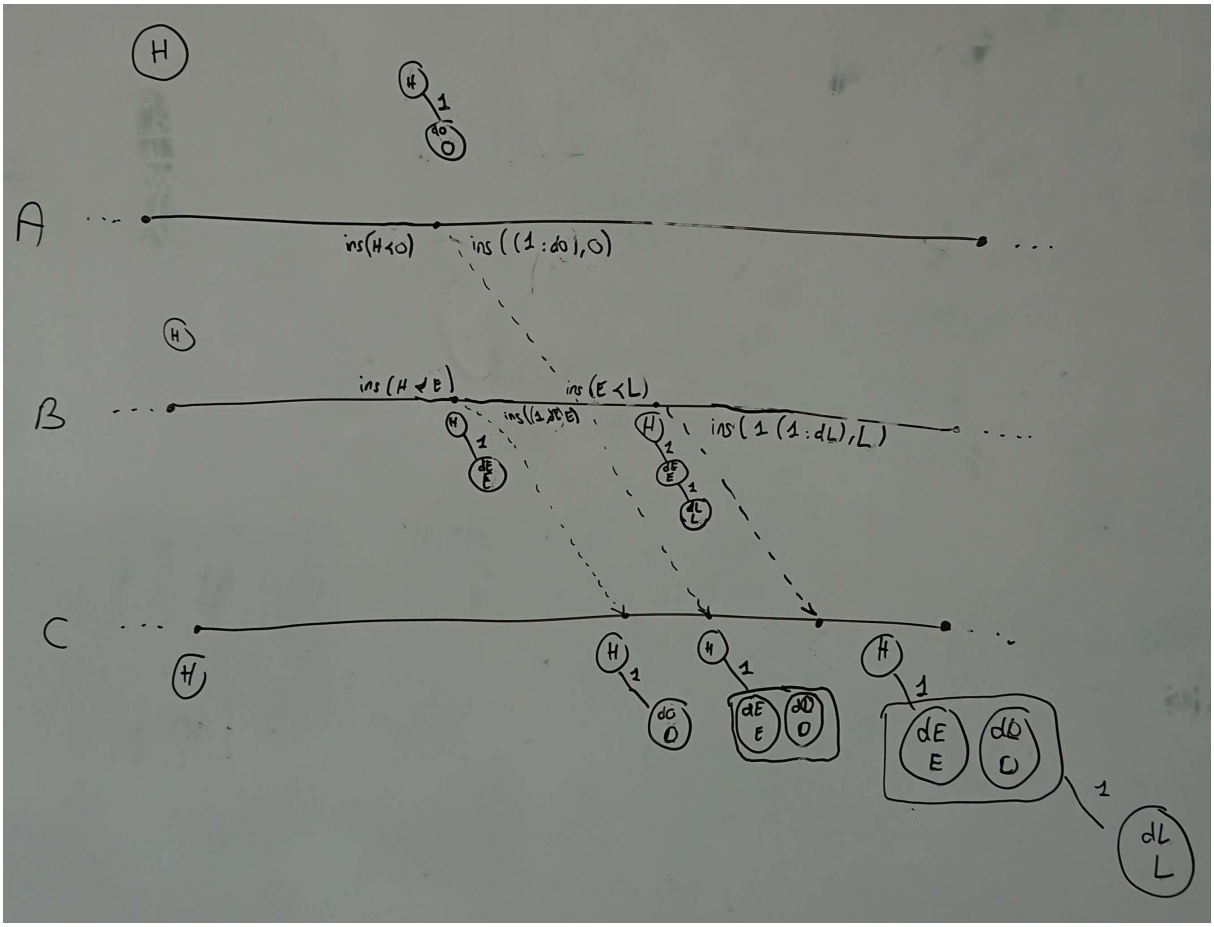


FIGURE A.1 – Modifications concurrentes d'une séquence Treedoc résultant en un entrelacement

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

Annexe B

Algorithmes RENAMEID

Algorithme 2 Remaining functions to rename an identifier

```
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
```

Annexe C

Algorithmes revertRenameId

Algorithme 3 Remaining functions to revert an identifier renaming

```
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
```

Index

Voici un index

FiXme :
Notes :
10 : Matthieu : TODO : Autres Sequence CRDTs à considérer : String-wise CRDT [87], Chronofold [91], 49
11 : Matthieu : TODO : Revoir refs utilisées ici, 49
12 : Matthieu : TODO : Ajouter MàJ de generateId ici. Pour profiter de cette fonctionnalité, LogootSplit propose une nouvelle fonction generateId. Le principal ajout est un cas supplémentaire favorisant la génération d'un id contigu dans le cas où predId est le dernier élément d'un intervalle d'identifiants de l'auteur. Le booléen isAppendable est nécessaire pour éviter de re-générer un identifiant avec un triplet nodeId,seq,offset déjà utilisé. , 54
13 : Matthieu : TODO : Serait plus intéressant de proposer des stats sur la taille des ids, le nombre de blocs composant la séquence, le nombre d'éléments par blocs et la proportion de la taille du contenu sur la taille de la structure de données en fonction du nombre d'opérations jouées. Pose la question de quand introduire le protocole suivi pour générer les traces. , 61
14 : Matthieu : TODO : Serait intéressant d'ajouter une catégorisation des éditeurs collaboratifs en fonction de leurs caractéristiques (décentralisé vs. p2p, pas de chiffrement vs. chiffrement serveur vs. chiffrement de bout en bout, OT vs CRDT vs mécanisme de résolution de conflits custom...) pour mettre en avant le caractère unique de MUTE, 68
15 : Matthieu : TODO : Serait intéressant d'ajouter une catégorisation des éditeurs collaboratifs en fonction de leurs caractéristiques (décentralisé vs. p2p, pas de chiffrement vs. chiffrement serveur vs. chiffrement de bout en bout, OT vs CRDT vs mécanisme de résolution de conflits custom...) pour mettre en avant le caractère unique de MUTE, 69
16 : Matthieu : TODO : Insérer refs distance de Hamming, Levenstein, String-to-string correction problem (Tichy et al), 106
17 : Matthieu : TODO : Insérer refs travaux Claudia et Vinh, 106
18 : Matthieu : TODO : Ajouter refs, 107
19 : Matthieu : TODO : À confirmer pour le graphe, 109
1 : Matthieu : TODO : Introduire notion d'agents artificiels/logiciels, 4
20 : Matthieu : TODO : Réaliser au propre contre-exemple. Nécessite que $d_E < d_O$, inverser A et B histoire d'éviter toute confusion. En soi, C pas nécessaire, à voir si le conserve. , 111
2 : Matthieu : TODO : Voir si angle

écologique/réduction consommation d'énergie peut être pertinent., 4
3 : Matthieu : TODO : Vérifier du côté des applis de IPFS, 7
4 : Matthieu : TODO : Faire le lien avec les travaux de Burckhardt [44] et les MRDTs [45], 16
5 : Matthieu : TODO : Ajouter refs des horloges logiques plus intelligentes (Interval Tree Clock, Hybrid Clock...), 18
6 : Matthieu : TODO : Ajouter refs Scuttlebutt si applicable à Op-based, 25
7 : Matthieu : TODO : Ajouter refs, 26
8 : Matthieu : TODO : Vérifier que c'est bien le cas dans [61], 26
9 : Matthieu : TODO : Ajouter refs, celles utilisées dans [74]., 36
FiXme (Matthieu) :
Notes :
10 : TODO : Autres Sequence CRDTs à considérer : String-wise CRDT [87], Chronofold [91], 49
11 : TODO : Revoir refs utilisées ici, 49
12 : TODO : Ajouter MàJ de generateId ici. Pour profiter de cette fonctionnalité, LogootSplit propose une nouvelle fonction generateId. Le principal ajout est un cas supplémentaire favorisant la génération d'un id contigu dans le cas où predId est le dernier élément d'un intervalle d'identifiants de l'auteur. Le booléen isAppendable est nécessaire pour éviter de re-générer un identifiant avec un triplet nodeId,seq,offset déjà utilisé. , 54
13 : TODO : Serait plus intéressant de proposer des stats sur la taille des ids, le nombre de blocs composant la séquence, le nombre d'éléments par blocs et la proportion de la taille du contenu sur la taille de la structure de données en fonction du nombre d'opérations jouées. Pose la question de quand introduire le protocole suivi pour générer les traces. , 61
14 : TODO : Serait intéressant d'ajouter une catégorisation des éditeurs collaboratifs en fonction de leurs caractéristiques (décentralisé vs. p2p, pas de chiffrement vs. chiffrement serveur vs. chiffrement de bout en bout, OT vs CRDT vs mécanisme de résolution de conflits custom...) pour mettre en avant le caractère unique de MUTE, 68
15 : TODO : Serait intéressant d'ajouter une catégorisation des éditeurs collaboratifs en fonction de leurs caractéristiques (décentralisé vs. p2p, pas de chiffrement vs. chiffrement serveur vs. chiffrement de bout en bout, OT vs CRDT vs mécanisme de résolution de conflits custom...) pour mettre en avant le caractère unique de MUTE, 69
16 : TODO : Insérer refs distance de Hamming, Levenstein, String-to-string correction problem (Tichy et al), 106
17 : TODO : Insérer refs travaux Claudia et Vinh, 106
18 : TODO : Ajouter refs, 107
19 : TODO : À confirmer pour le graphe, 109
1 : TODO : Introduire notion d'agents artificiels/logiciels, 4
20 : TODO : Réaliser au propre contre-exemple. Nécessite que $d_E < d_O$, inverser A et B histoire d'éviter toute confusion. En soi, C pas nécessaire, à voir si le conserve. , 111
2 : TODO : Voir si angle écologique/réduction consommation d'énergie peut être pertinent., 4
3 : TODO : Vérifier du côté des applis

de IPFS, 7
4 : TODO : Faire le lien avec les travaux de Burckhardt [44] et les MRDTs [45], 16
5 : TODO : Ajouter refs des horloges logiques plus intelligentes (Interval Tree Clock, Hybrid Clock...), 18
6 : TODO : Ajouter refs Scuttlebutt si applicable à Op-based, 25
7 : TODO : Ajouter refs, 26
8 : TODO : Vérifier que c'est bien le cas dans [61], 26
9 : TODO : Ajouter refs, celles utilisées dans [74]., 36

Bibliographie

[1] Jonathan A Obar et Steven S Wildman. « Social Media Definition and the Governance Challenge - An Introduction to the Special Issue ». In : *Obar, JA and Wildman, S.(2015). Social media definition and the governance challenge : An introduction to the special issue. Telecommunications policy* 39.9 (2015), p. 745–750.

[2] Statista. *Biggest social media platforms 2022*. Last Accessed : 2022-10-06. URL : https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/.

[3] Wikimedia. *Wikimedia Statistics - English Wikipedia*. Last Accessed : 2022-10-06. URL : https://stats.wikimedia.org/#/en.wikipedia.org.

[4] David Meek. « YouTube and Social Movements : A Phenomenological Analysis of Participation, Events and Cyberplace ». In : *Antipode* 44.4 (2012), p. 1429–1448. DOI : https://doi.org/10.1111/j.1467-8330.2011.00942.x. eprint : https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8330.2011.00942.x. URL : https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8330.2011.00942.x.

[5] Yannis Theocharis. « The wealth of (occupation) networks ? Communication patterns and information distribution in a Twitter protest network ». In : *Journal of Information Technology & Politics* 10.1 (2013), p. 35–56.

[6] José Van Dijck et Thomas Poell. « Understanding social media logic ». In : *Media and communication* 1.1 (2013), p. 2–14.

[7] Jim Giles. « Special Report Internet encyclopaedias go head to head ». In : *nature* 438.15 (2005), p. 900–901.

[8] Lada A Adamic, Jun Zhang, Eytan Bakshy et Mark S Ackerman. « Knowledge sharing and yahoo answers : everyone knows something ». In : *Proceedings of the 17th international conference on World Wide Web*. 2008, p. 665–674.

[9] Paul Baran. « On distributed communications networks ». In : *IEEE transactions on Communications Systems* 12.1 (1964), p. 1–9.

[10] Safiya Umoja Noble. *Algorithms of Oppression : How Search Engines Reinforce Racism*. NYU Press, 2018. ISBN : 9781479849949.

[11] Amnesty International. *#Toxictwitter : Violence and abuse against women online*. Last Accessed : 2022-10-07. URL : https://www.amnesty.org/en/documents/act30/8070/2018/en/.

[12] Wall Street Journal. *Facebook Tried to Make Its Platform a Healthier Place. It Got Angrier Instead.* Last Accessed : 2022-10-07. URL : https://t.co/P6JohMdhQE.

[13] Wall Street Journal. *Facebook Knows Instagram Is Toxic for Teen Girls, Company Documents Show.* Last Accessed : 2022-10-07. URL : https://t.co/JAvzKFc61q.

[14] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg et Mark McGranaghan. « Local-First Software : You Own Your Data, in Spite of the Cloud ». In : *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward ! 2019. Athens, Greece : Association for Computing Machinery, 2019, p. 154–178. ISBN : 9781450369954. DOI : 10.1145/3359591.3359737. URL : https://doi.org/10.1145/3359591.3359737.

[15] Daniel Abadi. « Consistency Tradeoffs in Modern Distributed Database System Design : CAP is Only Part of the Story ». In : *Computer* 45.2 (2012), p. 37–42. DOI : 10.1109/MC.2012.33.

[16] 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.

[17] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer et Carl H Hauser. « Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System ». In : *SIGOPS Oper. Syst. Rev.* 29.5 (déc. 1995), p. 172–182. ISSN : 0163-5980. DOI : 10.1145/224057.224070. URL : https://doi.org/10.1145/224057.224070.

[18] Daniel Stutzbach et Reza Rejaie. « Understanding Churn in Peer-to-Peer Networks ». In : *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*. IMC '06. Rio de Janeriro, Brazil : Association for Computing Machinery, 2006, p. 189–202. ISBN : 1595935614. DOI : 10.1145/1177080.1177105. URL : https://doi.org/10.1145/1177080.1177105.

[19] Leslie Lamport. « The part-time parliament ». In : *Concurrency : the Works of Leslie Lamport*. 2019, p. 277–317.

[20] Diego Ongaro et John Ousterhout. « In search of an understandable consensus algorithm ». In : *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 2014, p. 305–319.

[21] Marc Shapiro et Nuno Preguiça. *Designing a commutative replicated data type*. Research Report RR-6320. INRIA, 2007. URL : https://hal.inria.fr/inria-00177693.

[22] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero et Marek Zawirski. « Conflict-Free Replicated Data Types ». In : *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. SSS 2011. 2011, p. 386–400. DOI : 10.1007/978-3-642-24550-3_29.

[23] Mihai Letia, Nuno Preguiça et Marc Shapiro. « Consistency without concurrency control in large, dynamic systems ». In : *LADIS 2009 - 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*. T. 44. Operating Systems Review 2. Big Sky, MT, United States : Assoc. for Computing Machinery, oct. 2009, p. 29–34. DOI : 10.1145/1773912.1773921. URL : https://hal.inria.fr/hal-01248270.

[24] 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.

[25] Google. *Google Docs*. Last Accessed : 2022-10-07. URL : https://docs.google.com/.

[26] Cody Odgen. *Google Graveyard*. Last Accessed : 2022-10-11. URL : https://killedbygoogle.com/.

[27] Matthieu Nicolas, Victorien Elvinger, Gérald Oster, Claudia-Lavinia Ignat et François Charoy. « MUTE : A Peer-to-Peer Web-based Real-time Collaborative Editor ». In : *ECSCW 2017 - 15th European Conference on Computer-Supported Cooperative Work*. T. 1. Proceedings of 15th European Conference on Computer-Supported Cooperative Work - Panels, Posters and Demos 3. Sheffield, United Kingdom : EUSSET, août 2017, p. 1–4. DOI : 10.18420/ecscw2017_p5. URL : https://hal.inria.fr/hal-01655438.

[28] Luc André, Stéphane Martin, Gérald Oster et Claudia-Lavinia Ignat. « Supporting Adaptable Granularity of Changes for Massive-Scale Collaborative Editing ». In : *International Conference on Collaborative Computing : Networking, Applications and Worksharing - CollaborateCom 2013*. Austin, TX, USA : IEEE Computer Society, oct. 2013, p. 50–59. DOI : 10.4108/icst.collaboratecom.2013.254123.

[29] 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.

[30] Matthieu Nicolas, Gerald Oster et Olivier Perrin. « Efficient Renaming in Sequence CRDTs ». In : *IEEE Transactions on Parallel and Distributed Systems* 33.12 (déc. 2022), p. 3870–3885. DOI : 10.1109/TPDS.2022.3172570. URL : https://hal.inria.fr/hal-03772633.

[31] 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.

[32] Hoang-Long Nguyen, Jean-Philippe Eisenbarth, Claudia-Lavinia Ignat et Olivier Perrin. « Blockchain-Based Auditing of Transparent Log Servers ». In : *32th IFIP Annual Conference on Data and Applications Security and Privacy (DBSec)*. Sous la dir. de Florian Kerschbaum et Stefano Paraboschi. T. LNCS-10980. Data and Applications Security and Privacy XXXII. Part 1 : Administration. Bergamo, Italy : Springer International Publishing, juil. 2018, p. 21–37. doi : 10.1007/978-3-319-95729-6_2. url : https://hal.archives-ouvertes.fr/hal-01917636.

[33] Abhinandan Das, Indranil Gupta et Ashish Motivala. « SWIM : scalable weakly-consistent infection-style process group membership protocol ». In : *Proceedings International Conference on Dependable Systems and Networks*. 2002, p. 303–312. doi : 10.1109/DSN.2002.1028914.

[34] 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.

[35] Brice Nédelec, Julian Tanke, Davide Frey, Pascal Molli et Achour Mostéfaoui. « An adaptive peer-sampling protocol for building networks of browsers ». In : *World Wide Web* 21.3 (2018), p. 629–661.

[36] 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.

[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] Rachid Guerraoui, Matej Pavlovic et Dragos-Adrian Seredinschi. « Trade-offs in replicated systems ». In : *IEEE Data Engineering Bulletin* 39.ARTICLE (2016), p. 14–26.

[40] Leslie Lamport. « Time, Clocks, and the Ordering of Events in a Distributed System ». In : *Commun. ACM* 21.7 (juil. 1978), p. 558–565. issn : 0001-0782. doi : 10.1145/359545.359563. url : https://doi.org/10.1145/359545.359563.

[41] Nuno M. Preguiça, Carlos Baquero et Marc Shapiro. « Conflict-free Replicated Data Types (CRDTs) ». In : *CoRR* abs/1805.06358 (2018). arXiv : 1805.06358. url : http://arxiv.org/abs/1805.06358.

[42] Nuno M. Preguiça. « Conflict-free Replicated Data Types : An Overview ». In : *CoRR* abs/1806.10254 (2018). arXiv : 1806.10254. url : http://arxiv.org/abs/1806.10254.

[43] B. A. Davey et H. A. Priestley. *Introduction to Lattices and Order*. 2e éd. Cambridge University Press, 2002. doi : 10.1017/CBO9780511809088.

[44] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang et Marek Zawirski. « Replicated Data Types : Specification, Verification, Optimality ». In : *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA : Association for Computing Machinery, 2014, p. 271–284. isbn : 9781450325448. doi : 10.1145/2535838.2535848. url : https://doi.org/10.1145/2535838.2535848.

[45] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan et Suresh Jagannathan. « Mergeable Replicated Data Types ». In : *Proc. ACM Program. Lang.* 3.OOPSLA (oct. 2019). doi : 10.1145/3360580. url : https://doi.org/10.1145/3360580.

[46] Paul R Johnson et Robert Thomas. *RFC0677 : Maintenance of duplicate databases*. RFC Editor, 1975.

[47] Weihai Yu et Sigbjørn Rostad. « A Low-Cost Set CRDT Based on Causal Lengths ». In : *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. New York, NY, USA : Association for Computing Machinery, 2020. isbn : 9781450375245. url : https://doi.org/10.1145/3380787.3393678.

[48] Marc Shapiro, Nuno Preguiça, Carlos Baquero et Marek Zawirski. *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.

[49] 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.

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

[51] 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.

[52] 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.

[53] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin et al. « Consistency, availability, and convergence ». In : *University of Texas at Austin Tech Report* 11 (2011), p. 158.

[54] Friedemann MATTERN et al. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[55] Colin FIDGE. « Logical Time in Distributed Computing Systems ». In : *Computer* 24.8 (août 1991), p. 28–33. ISSN : 0018-9162. DOI : 10.1109/2.84874. URL : https://doi.org/10.1109/2.84874.

[56] Ravi PRAKASH, Michel RAYNAL et Mukesh SINGHAL. « An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments ». In : *Journal of Parallel and Distributed Computing* 41.2 (1997), p. 190–204. ISSN : 0743-7315. DOI : https://doi.org/10.1006/jpdc.1996.1300. URL : https://www.sciencedirect.com/science/article/pii/S0743731596913003.

[57] D. S. PARKER, G. J. POPEK, G. RUDISIN, A. STOUGHTON, B. J. WALKER, E. WALTON, J. M. CHOW, D. EDWARDS, S. KISER et C. KLINE. « Detection of Mutual Inconsistency in Distributed Systems ». In : *IEEE Trans. Softw. Eng.* 9.3 (mai 1983), p. 240–247. ISSN : 0098-5589. DOI : 10.1109/TSE.1983.236733. URL : https://doi.org/10.1109/TSE.1983.236733.

[58] Giuseppe DECANDIA, Deniz HASTORUN, Madan JAMPANI, Gunavardhan KAKULAPATI, Avinash LAKSHMAN, Alex PILCHIN, Swaminathan SIVASUBRAMANIAN, Peter VOSSHALL et Werner VOGELS. « Dynamo : Amazon's highly available key-value store ». In : *ACM SIGOPS operating systems review* 41.6 (2007), p. 205–220.

[59] Nico KRUBER, Maik LANGE et Florian SCHINTKE. « Approximate Hash-Based Set Reconciliation for Distributed Replica Repair ». In : *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. 2015, p. 166–175. DOI : 10.1109/SRDS.2015.30.

[60] Ricardo Jorge Tomé GONÇALVES, Paulo Sérgio ALMEIDA, Carlos BAQUERO et Vitor FONTE. « DottedDB : Anti-Entropy without Merkle Trees, Deletes without Tombstones ». In : *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. 2017, p. 194–203. DOI : 10.1109/SRDS.2017.28.

[61] Jim BAUWENS et Elisa Gonzalez BOIX. « Improving the Reactivity of Pure Operation-Based CRDTs ». In : *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '21. Online, United Kingdom : Association for Computing Machinery, 2021. ISBN : 9781450383387. DOI : 10.1145/3447865.3457968. URL : https://doi.org/10.1145/3447865.3457968.

[62] 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.

[63] Vitor ENES, Paulo Sérgio ALMEIDA, Carlos BAQUERO et João LEITÃO. « Efficient Synchronization of State-Based CRDTs ». In : *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, p. 148–159. DOI : 10.1109/ICDE.2019.00022.

[64] Clarence A. Ellis et Simon J. Gibbs. « Concurrency Control in Groupware Systems ». In : *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. Portland, Oregon, USA : Association for Computing Machinery, 1989, p. 399–407. ISBN : 0897913175. DOI : 10.1145/67544.66963. URL : https://doi.org/10.1145/67544.66963.

[65] Chengzheng Sun et Clarence Ellis. « Operational transformation in real-time group editors : issues, algorithms, and achievements ». In : *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. 1998, p. 59–68.

[66] Matthias Ressel, Doris Nitsche-Ruhland et Rul Gunzenhäuser. « An integrating, transformation-oriented approach to concurrency control and undo in group editors ». In : *Proceedings of the 1996 ACM conference on Computer supported cooperative work*. 1996, p. 288–297.

[67] Chengzheng Sun, Yun Yang, Yanchun Zhang et David Chen. « A consistency model and supporting schemes for real-time cooperative editing systems ». In : *Australian Computer Science Communications* 18 (1996), p. 582–591.

[68] David Sun et Chengzheng Sun. « Context-Based Operational Transformation in Distributed Collaborative Editing Systems ». In : *Parallel and Distributed Systems, IEEE Transactions on* 20 (nov. 2009), p. 1454–1470. DOI : 10.1109/TPDS.2008.240.

[69] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang et David Chen. « Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems ». In : *ACM Transactions on Computer-Human Interaction (TOCHI)* 5.1 (1998), p. 63–108.

[70] Gérald Oster, Pascal Molli, Pascal Urso et Abdessamad Imine. « Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems ». In : *2006 International Conference on Collaborative Computing : Networking, Applications and Worksharing*. 2006, p. 1–10. DOI : 10.1109/COLCOM.2006.361867.

[71] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang et David Chen. « 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.

[72] 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.

[73] Bernadette Charron-Bost. « Concerning the size of logical clocks in distributed systems ». In : *Information Processing Letters* 39.1 (1991), p. 11–16.

[74] Gérald Oster, Pascal Urso, Pascal Molli et Abdessamad Imine. « Data Consistency for P2P Collaborative Editing ». In : *ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*. Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work. Banff, Alberta, Canada : ACM Press, nov. 2006, p. 259–268. URL : https://hal.inria.fr/inria-00108523.

[75] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim et Joonwon Lee. « Replicated abstract data types : Building blocks for collaborative applications ». In : *Journal of Parallel and Distributed Computing* 71.3 (2011), p. 354–368. ISSN : 0743-7315. DOI : https://doi.org/10.1016/j.jpdc.2010.12.006. URL : http://www.sciencedirect.com/science/article/pii/S0743731510002716.

[76] Nuno Preguica, Joan Manuel Marques, Marc Shapiro et Mihai Letia. « A Commutative Replicated Data Type for Cooperative Editing ». In : *2009 29th IEEE International Conference on Distributed Computing Systems*. Juin 2009, p. 395–403. DOI : 10.1109/ICDCS.2009.20.

[77] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça et Victor Fonte. « Scalable and Accurate Causality Tracking for Eventually Consistent Stores ». In : *Distributed Applications and Interoperable Systems*. Sous la dir. de Kostas Magoutis et Peter Pietzuch. Berlin, Heidelberg : Springer Berlin Heidelberg, 2014, p. 67–81. ISBN : 978-3-662-43352-2.

[78] Charbel Rahhal, Stéphane Weiss, Hala Skaf-Molli, Pascal Urso et Pascal Molli. *Undo in Peer-to-peer Semantic Wikis*. Research Report RR-6870. INRIA, 2009, p. 18. URL : https://hal.inria.fr/inria-00366317.

[79] Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh et Pascal Urso. « 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.

[80] 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.

[81] Ben Shneiderman. « Response Time and Display Rate in Human Performance with Computers ». In : *ACM Comput. Surv.* 16.3 (sept. 1984), p. 265–285. ISSN : 0360-0300. DOI : 10.1145/2514.2517. URL : https://doi.org/10.1145/2514.2517.

[82] Caroline Jay, Mashhuda Glencross et Roger Hubbold. « Modeling the Effects of Delayed Haptic and Visual Feedback in a Collaborative Virtual Environment ». In : *ACM Trans. Comput.-Hum. Interact.* 14.2 (août 2007), 8–es. ISSN : 1073-0516. DOI : 10.1145/1275511.1275514. URL : https://doi.org/10.1145/1275511.1275514.

[83] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang et Marek Zawirski. « Specification and Complexity of Collaborative Text Editing ». In : *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. PODC '16. Chicago, Illinois, USA : Association for Computing Machinery, 2016, p. 259–268. ISBN : 9781450339643. DOI : 10.1145/2933057.2933090. URL : https://doi.org/10.1145/2933057.2933090.

[84] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang et Marek Zawirski. « Specification and space complexity of collaborative text editing ». In : *Theoretical Computer Science* 855 (2021), p. 141–160. ISSN : 0304-3975. DOI : https://doi.org/10.1016/j.tcs.2020.11.046. URL : http://www.sciencedirect.com/science/article/pii/S0304397520306952.

[85] Automerge. *Automerge : data structures for building collaborative applications in Javascript*. Last Accessed : 2022-10-07. URL : https://github.com/automerge/automerge.

[86] 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.

[87] 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.

[88] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan et Alastair R. Beresford. « Interleaving Anomalies in Collaborative Text Editors ». In : *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '19. Dresden, Germany : Association for Computing Machinery, 2019. ISBN : 9781450362764. DOI : 10.1145/3301419.3323972. URL : https://doi.org/10.1145/3301419.3323972.

[89] Matthew Weidner. *There Are No Doubly Non-Interleaving List CRDTs*. Last Accessed : 2022-10-07. URL : https://mattweidner.com/assets/pdf/List_CRDT_Non_Interleaving.pdf.

[90] 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.

[91] 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.

[92] Claudia-Lavinia IGNAT, Gérald OSTER, Meagan NEWMAN, Valerie SHALIN et François CHAROY. « Studying the Effect of Delay on Group Performance in Collaborative Editing ». In : *Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014, Springer 2014 Lecture Notes in Computer Science*. Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014. Seattle, WA, United States, sept. 2014, p. 191–198. DOI : 10.1007/978-3-319-10831-5_29. URL : https://hal.archives-ouvertes.fr/hal-01088815.

[93] Claudia-Lavinia IGNAT, Gérald OSTER, Olivia FOX, François CHAROY et Valerie SHALIN. « How Do User Groups Cope with Delay in Real-Time Collaborative Note Taking ». In : *European Conference on Computer Supported Cooperative Work 2015*. Sous la dir. de Nina BOULUS-RODJE, Gunnar ELLINGSEN, Tone BRATTETEIG, Margunn AANESTAD et Pernille BJORN. Proceedings of the 14th European Conference on Computer Supported Cooperative Work. Oslo, Norway : Springer International Publishing, sept. 2015, p. 223–242. DOI : 10.1007/978-3-319-20499-4_12. URL : https://hal.inria.fr/hal-01238831.

[94] Brice NÉDELEC, Pascal MOLLI, Achour MOSTÉFAOUI et Emmanuel DESMONTILS. « LSEQ : an adaptive structure for sequences in distributed collaborative editing ». In : *Proceedings of the 2013 ACM Symposium on Document Engineering*. DocEng 2013. Sept. 2013, p. 37–46. DOI : 10.1145/2494266.2494278.

[95] 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.

[96] Sylvie NOËL et Jean-Marc ROBERT. « Empirical study on collaborative writing : What do co-authors do, use, and like ? » In : *Computer Supported Cooperative Work (CSCW)* 13.1 (2004), p. 63–89.

[97] ETHERPAD. *Etherpad*. Last Accessed : 2022-10-07. URL : https://etherpad.org/.

[98] Quang-Vinh DANG et Claudia-Lavinia IGNAT. « Performance of real-time collaborative editors at large scale : User perspective ». In : *Internet of People Workshop, 2016 IFIP Networking Conference*. Proceedings of 2016 IFIP Networking Conference, Networking 2016 and Workshops. Vienna, Austria, mai 2016, p. 548–553. DOI : 10.1109/IFIPNetworking.2016.7497258. URL : https://hal.inria.fr/hal-01351229.

[99] Barton GELLMAN et Laura POITRAS. *U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program*. Last Accessed : 2022-10-07. URL : https://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html.

[100] Glen Greenwald et Ewen MacAskill. *NSA Prism program taps in to user data of Apple, Google and others*. Last Accessed : 2022-10-07. URL : https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data.

[101] 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.

[102] Jim Pick. *PeerPad*. Last Accessed : 2022-10-07. URL : https://peerpad.net/.

[103] Jim Pick. *Graf, Nikolaus*. Last Accessed : 2022-10-07. URL : https://www.serenity.re/en/notes.

[104] 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.

[105] John Gruber. *Daring Fireball : Markdown*. Last Accessed : 2022-10-17. URL : https://daringfireball.net/projects/markdown/.

[106] 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.

[107] Victorien Elvinger, Gérald Oster et Francois Charoy. « Prunable Authenticated Log and Authenticable Snapshot in Distributed Collaborative Systems ». In : *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. 2018, p. 156–165. DOI : 10.1109/CIC.2018.00031.

[108] OpenRelay. *OpenRelay*. Last Accessed : 2022-10-07. URL : https://openrelay.xyz/.

[109] Protocol Labs. *IPFS*. Last Accessed : 2022-10-07. URL : https://ipfs.io/.

[110] Quang Vinh Dang et Claudia-Lavinia Ignat. « Quality Assessment of Wikipedia Articles : A Deep Learning Approach by Quang Vinh Dang and Claudia-Lavinia Ignat with Martin Vesely as Coordinator ». In : *SIGWEB Newsl.* Autumn (nov. 2016). ISSN : 1931-1745. DOI : 10.1145/2996442.2996447. URL : https://doi.org/10.1145/2996442.2996447.

[111] Leslie Lamport, Robert Shostak et Marshall Pease. « The Byzantine Generals Problem ». In : *Concurrency : The Works of Leslie Lamport*. New York, NY, USA : Association for Computing Machinery, 2019, p. 203–226. ISBN : 9781450372701. URL : https://doi.org/10.1145/3335772.3335936.

[112] 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.

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