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

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

Mis en page avec la classe thesul.

Remerciements

Les remerciements.

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

Sommaire

Table des figures

Introduction

1 Contexte

— Systèmes collaboratifs (wikis, plateformes de contenu, réseaux sociaux) et leurs bienfaits (qualité de l'info, vitesse de l'info (exemple de crise ?), diffusion de la parole). Démocratisation (sic) de ces systèmes au cours de la dernière décennie.

— En raison du volume de données et de requêtes, adoptent architecture décentralisée. Permet ainsi de garantir disponibilité, tolérance aux pannes et capacité de passage à l'échelle.

— Mais échoue à adresser problèmes non-techniques : confidentialité, souveraineté, protection contre censure, dépendance et nécessité de confiance envers autorité centrale.

— À l'heure où les entreprises derrière ces systèmes font preuve d'ingérence et d'intérêts contraires à ceux de leurs utilisateur-rices (Cambridge Analytica, Prism, non-modération/mise en avant de contenus racistes [1] [2] [3], invisibilisation de contenus féministes, dissolution du comité d'éthique de Google [4], inégalité d'accès à la méta-machine affectante [5] [6] [7]), parait fondamental de proposer les moyens technologiques accessibles pour concevoir et déployer des alternatives.

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

— Systèmes pair-à-pair sont une direction intéressante pour répondre à ces problématiques, de part leur absence d'autorité centrale, la distribution des tâches et leur conception mettant le pair au centre. Mais posent de nouvelles problématiques de recherche.

— Ces systèmes ne disposent d'aucun contrôle sur les noeuds qui les composent. Le nombre de noeuds peut donc croître de manière non-bornée et atteindre des centaines de milliers de noeuds. La complexité des algorithmes de ces systèmes ne doit donc pas dépendre de ce paramètre, ou alors de manière logarithmique.

1. *Algorithms of Oppression*, Safiya Umoja Noble
2. `https://www.researchgate.net/publication/342113147_The_YouTube_Algorithm_and_the_Alt-Right_Filter_Bubble`
3. `https://www.wsj.com/articles/the-facebook-files-11631713039`
4. `https://www.bbc.com/news/technology-56135817`
5. *Je suis une fille sans histoire*, Alice Zeniter, p. 75
6. Qui cite *Les affects de la politique*, Frédéric Lordon
7. `https://www.bbc.com/news/technology-59011271`

— De plus, ces noeuds n'offrent aucune garantie sur leur stabilité. Ils peuvent donc rejoindre et participer au système de manière éphèmère. S'agit du phénomène connu sous le nom de churn. Les algorithmes de ces systèmes ne peuvent donc pas reposer sur des mécanismes nécessitant une coordination synchrone d'une proportion des noeuds.

— Finalement, ces noeuds n'offrent aucune garanties sur leur fiabilité et intentions. Les noeuds peuvent se comporter de manière byzantine. Pour assurer la confidentialité, l'absence de confiance requise et le bon fonctionnement du système, ce dernier doit être conçu pour résister aux comportements byzantins de ses acteurs.

— Ainsi, il est nécessaire de faire progresser les technologies existantes pour les rendre compatible avec ce nouveau modèle de système. Dans le cadre de cette thèse, nous nous intéressons aux mécanismes de réplication de données dans les systèmes collaboratifs pair-à-pair temps réel.

2 Questions de recherche et contributions

2.1 Ré-identification sans coordination pour CRDTs pour Séquence

— Systèmes collaboratifs permettent aux utilisateur-rices de manipuler et éditer un contenu partagé. Pour des raisons de performance, ces systèmes autorisent généralement les utilisateur-rices à effectuer des modifications sans coordination. Leur copies divergent alors momentanément. Un mécanisme de synchronisation leur permet ensuite de récupérer l'ensemble des modifications et de les intégrer, de façon à converger. Cependant, des modifications peuvent être incompatibles entre elles, car de sémantiques contraires. Un mécanisme de résolution de conflits est alors nécessaire.

— Les CRDTs sont des types de données répliquées. Ils sont conçus pour être répliqués par les noeuds d'un système et pour permettre à ces derniers de modifier les données partagées sans aucune coordination. Dans ce but, ils incluent des mécanismes de résolution de conflits directement au sein leur spécification. Ces mécanismes leur permettent de résoudre le problème évoqué précédemment. Cependant, ces mécanimes induisent un surcoût, aussi bien d'un point de vue consommation mémoire et réseau que computationnel. Notamment, certains CRDTs comme ceux pour la Séquence souffrent d'une croissance monotone de leur surcoût. Ce surcoût s'avère handicapant dans le contexte des collaborations à large échelle.

— Pouvons-nous proposer un mécanisme sans coordination de réduction du surcoût des CRDTs pour Séquence, c.-à-d. compatible avec les systèmes pair-à-pair ?

— Dans le cadre des CRDTs pour Séquence, le surcoût du type de données répliquées provient de la croissance de leurs métadonnées. Métadonnées proviennent des identifiants associés aux éléments de la Séquence par les CRDTs. Ces identifiants sont nécessaires pour le bon fonctionnement de leur mécanisme de résolution de conflits.

— Plusieurs approches ont été proposées pour réduire le coût de ces identifiants. Notamment, [1, 2] proposent un mécanisme de ré-assignation d'identifiants de façon à réduire leur taille. Mécanisme non commutatif avec les modifications concurrentes de la Séquence, c.-à-d. l'insertion ou la suppression. Propose ainsi un mécanisme de transformation des modifications concurrentes pour gérer ces conflits. Mais mécanisme de ré-assignation n'est pas non plus commutatif avec lui-même. De fait, utilisent un algorithme de consensus pour empêcher l'exécution du mécanisme en concurrence.

— Proposons RenamableLogootSplit, un nouveau CRDT pour Séquence. Intègre un mécanisme de renommage directement au sein de sa spécification. Intègre un mécanisme de résolution de conflits pour les renommages concurrents. Permet ainsi l'utilisation du mécanisme de renommage par les noeuds sans coordination.

2.2 Éditeur de texte collaboratif pair-à-pair

— Systèmes collaboratifs adoptent généralement architecture décentralisée. Disposent d'autorités centrales qui facilitent la collaboration, l'authentification des utilisateur-rices, la communication et le stockage des données.

— Mais ces systèmes introduisent une dépendance des utilisateur-rices envers ces mêmes autorités centrales, une perte de confidentialité et de souveraineté.

— Pouvons-nous concevoir un éditeur de texte collaboratif sans autorités centrales, c.-à-d. un éditeur de texte collaboratif à large échelle pair-à-pair ?

— Ce changement de modèle, d'une architecture décentralisée à une architecture pair-à-pair, introduit un ensemble de problématiques de domaines variés, e.g.

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

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

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

— Présentons Multi User Text Editor (MUTE) [3]. S'agit, à notre connaissance, du seul prototype complet d'éditeur de texte collaboratif temps réel pair-à-pair chiffré de bout en bout. Allie ainsi les résultats issus des travaux de l'équipe sur les CRDTs pour Séquence [4, 5] et l'authentification des pairs dans systèmes distribués [6, 7] aux résultats de la littérature sur mécanismes de conscience de groupe *Matthieu: TODO : Trouver et ajouter références*, les protocoles d'appartenance aux groupe [8, 9], les réseaux pair-à-pair [10] et les protocoles d'établissement de clés de groupe [11].

3 Plan du manuscrit

4 Publications

Chapitre 1

État de l'art

Sommaire

1.1 Systèmes distribués

— Contexte des systèmes distribués à large échelle

— Réplique les données afin de pouvoir supporter les pannes

— Adopte le paradigme de la réplication optimiste [12]

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

— Autorise alors les noeuds à diverger temporairement

— Permet d'être toujours disponible, de toujours répondre aux requêtes même en cas de partition réseau
— Permet aussi, en temps normal, de réduire le temps de réponse (privilégie la latence) [13]
— Comme ce modèle autorise les noeuds à modifier la donnée sans se coordonner, possible d'effectuer des modifications concurrentes
— Généralement, un mécanisme de résolution de conflits est nécessaire afin d'assurer la convergence des noeuds dans une telle situation
— Plusieurs approches ont été proposées pour implémenter un tel mécanisme

1.2 Types de données répliquées sans conflits

Pour limiter la coordination entre les noeuds, les systèmes distribués adoptent le paradigme de la réplication optimiste. Ce paradigme consiste à ce que chaque noeud possède une copie de la donnée. Chaque noeud possède le droit de la modifier sans se coordonner 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 de nouveau converger. Ce paradigme offre ainsi aux noeuds une haute disponibilité [14] ainsi qu'une faible latence.

Afin d'ordonner les modifications effectués dans un système, la littérature repose généralement sur la relation *happens-before* [15]. Nous l'adaptons ci-dessous à notre contexte :

Définition 1 (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 respectée :*

(i) m_1 *a eu lieu avant* m_2 *sur le même noeud.*

(ii) m_1 *a été délivrée au noeud auteur de* m_2 *avant la génération de* 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, on note que la relation *happens-before* ne permet pas d'établir un ordre total entre les modifications [8]. 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 2 (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. Ainsi, une

8. 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.

exécution dans un système distribué suivant le paradigme de réplication optimiste peut mener à la génération de modifications concurrentes non commutatives. Nous parlons alors de conflits. La figure Figure 1.1 présente un scénario où des modifications de sémantiques opposées sont générées en concurrence.

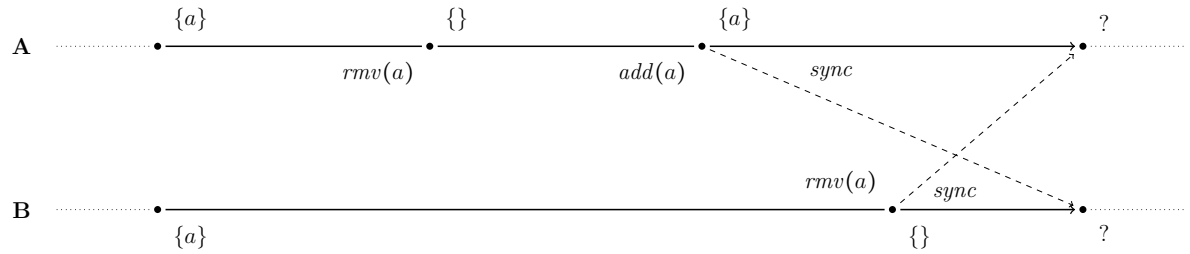


FIGURE 1.1 – Modifications concurrentes d'un Ensemble répliqué provoquant un conflit suite à l'ajout et la suppression d'un même élément

Dans cet exemple, deux noeuds A et B répliquent et partagent un même Ensemble. Les deux noeuds possèdent le même état initial : $\{a\}$. Le noeud A retire l'élément a de l'ensemble, $rmv(a)$. Puis, le noeud A ré-ajoute l'élément a dans l'ensemble via l'opération de 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) [16, 17, 18] répondent à ce besoin.

Définition 3 (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 *[16].*

Définition 4 (Convergence forte) *La convergence forte est une propriété de sûreté indiquant que l'ensemble des noeuds d'un système ayant observés le même ensemble de modifications obtiendront des états équivalents*[9]*.*

Pour offrir la propriété de *convergence forte*, la spécification des CRDTs reposent sur la théorie des treillis [19] :

9. 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.

(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. On parle aussi 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 [10] correspondants du point de vue des utilisateurs. Ainsi, les CRDTs partagent le comportement des types séquentiels dans le cadre d'exécutions séquentielles. Cependant, ils définissent aussi une nouvelle sémantique pour chaque nouveau scénario ne pouvant se produire que dans le cadre d'une exécution distribuée.

Plusieurs sémantiques valides peuvent être proposées pour ces nouveaux scénarios. 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 sémantique en cas de concurrence et son modèle de synchronisation. Dans les prochaines sections, nous présenterons les différentes sémantiques possibles pour un type donné, l'Ensemble, en guise d'exemple. Puis nous présenterons les différents modèles de synchronisation proposés dans la littérature, et détaillerons 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 [20] et les MRDTs [21]

1.2.1 Sémantiques en cas de concurrence

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 1.1.

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

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 [22], 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 1.2 illustre son application à l'Ensemble pour résoudre le conflit de la Figure 1.1.

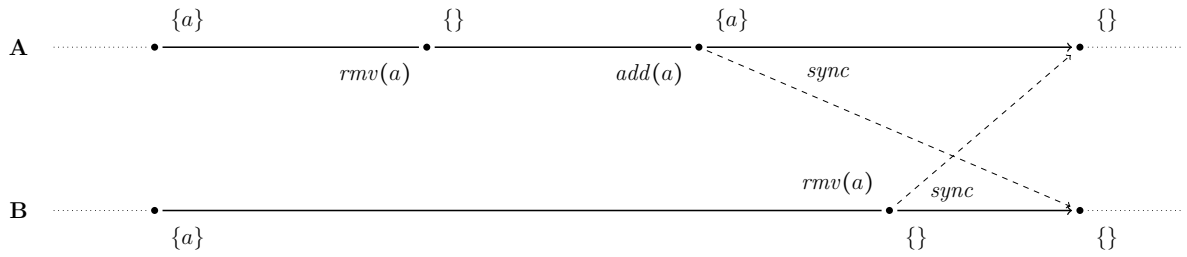


FIGURE 1.2 – Résolution du conflit en utilisant la sémantique LWW

Comme indiqué précédemment, le scénario illustré dans la Figure 1.2 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 [15]. *Matthieu: TODO : Ajouter refs des horloges logiques plus intelligentes (Interval Tree Clock, Hybrid Clock...)*

Sémantique *Multi-Value*

Une seconde sémantique générique[11] 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 1.3.

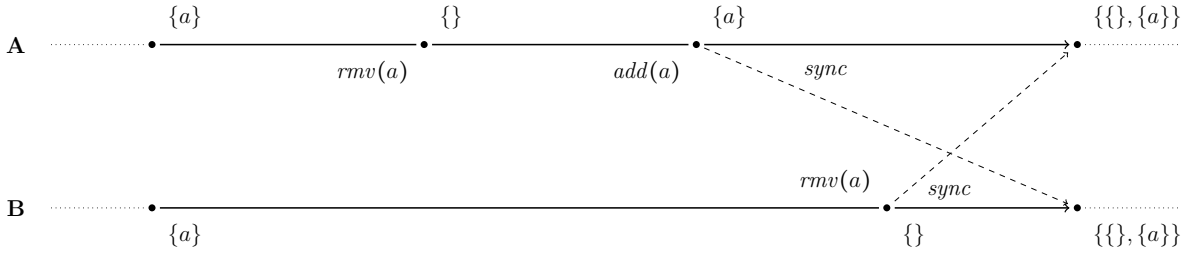


FIGURE 1.3 – Résolution du conflit en utilisant la sémantique MV

La Figure 1.3 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 1.3, 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 type correspondant au CRDT dans la Figure 1.3 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.

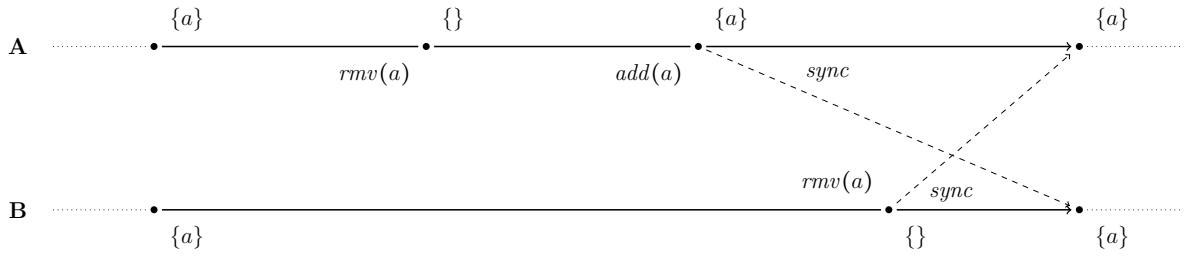
Dans le cadre de l'Ensemble, un conflit est provoqué lorsque des modifications *add* et *rmv* d'un même élément sont effectuées en concurrence. Ainsi, deux approches peuvent être proposées pour résoudre le conflit :

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

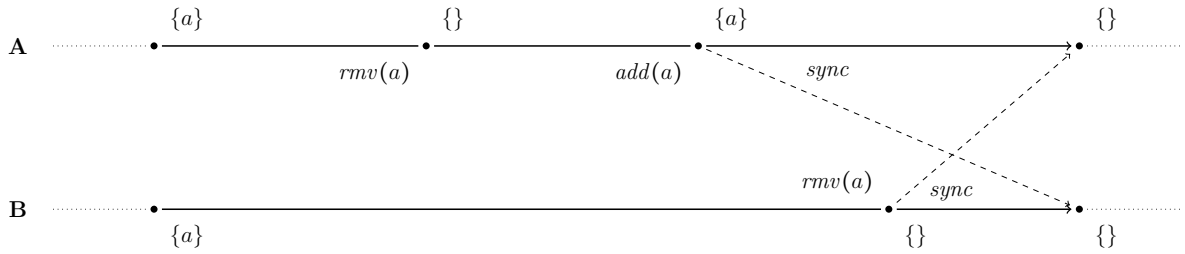
(i) Une sémantique où la modification *add* d'une élément prend la précédence sur les modifications concurrentes *rmv* du même élément, nommée *Add-Wins* (AW). L'élément est alors présent dans l'état obtenu à l'issue de la résolution du conflit.

(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 1.4 illustre l'application de chacune de ces sémantiques sur notre exemple.



(a) Application de la sémantique AW



(b) Application de la sémantique RW

FIGURE 1.4 – Résolution du conflit en utilisant soit la sémantique AW, soit la sémantique RW

Sémantique *Causal-Length*

Une nouvelle sémantique pour l'Ensemble fut proposée [23] 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 ROSTAD [23] 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 1.5 illustre son fonctionnement.

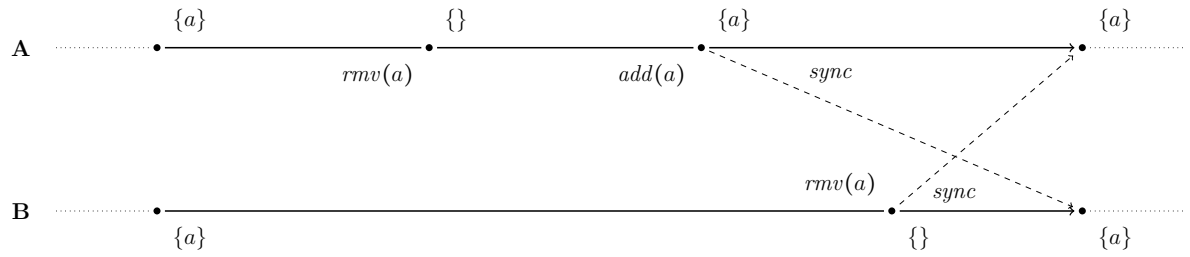


FIGURE 1.5 – Résolution du conflit en utilisant la sémantique CL

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.

1.2.2 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 1.6

illustre ce point.

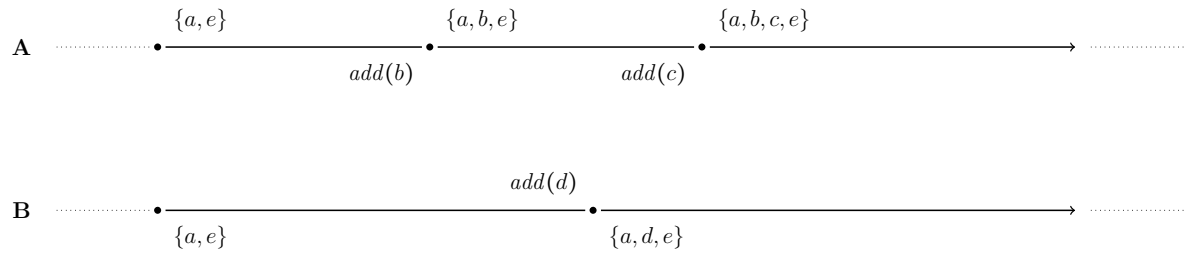


FIGURE 1.6 – Modifications en concurrence d'un Ensemble répliqué par les noeuds A et B

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 [12], 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 [16, 24] et une méthode de synchronisation par opérations [16, 24, 25, 26]. Une troisième approche, nommée synchronisation par différence d'états [27, 28], 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.

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 en (iii) : elle prend en paramètres une paire d'états et génère en retour l'état correspondant à leur LUB. 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

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

réseau sans lui imposer de contraintes fortes : les messages peuvent être perdus, réordonnés ou même dupliqués. Les noeuds convergeront tant que la couche réseau 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.

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 temps réel et repose généralement sur une synchronisation périodique.

Nous illustrons le fonctionnement de cette approche avec la Figure 1.7. 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).

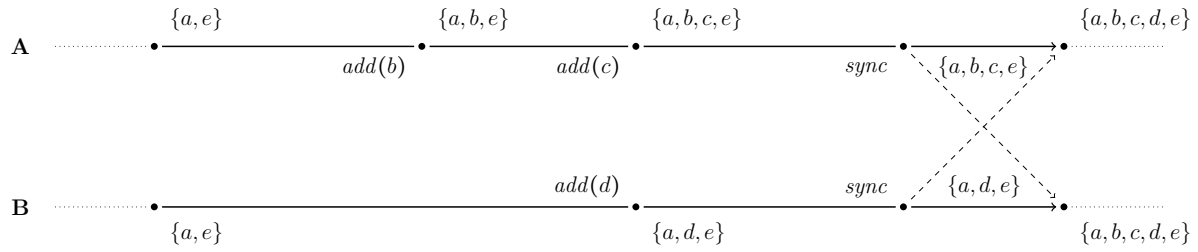


FIGURE 1.7 – Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par états

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 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 ce modèle de synchronisation propose par nature une diffusion des modifications suivant le modèle de cohérence causale [29]. En effet, ce modèle de synchronisation assure une livraison soit de toutes les modifications connues d'un noeud, soit d'aucune. Par exemple, dans la Figure 1.7, 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`.

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 5 (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 [26], les modifications labellisées 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 utilisables 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. 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 délivrées 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.

Généralement, les CRDTs synchronisés par opérations sont présentés dans la littérature comme nécessitant une livraison causale des opérations. Ce modèle de livraison permet de respecter le modèle de cohérence causale et ainsi de simplifier raisonnement sur exécutions.

Ce modèle introduit néanmoins plusieurs effets négatifs. Tout d'abord, ce modèle peut provoquer un délai dans la diffusion 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 sont délivrées qu'une fois l'opération perdue re-diffusée et délivrée.

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 *Matthieu: TODO : Insérer ref*. 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 [13]. *Matthieu: TODO : Ajouter mention que OT a été abandonné à cause de cette contrainte même.* 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 de la livraison dans l'ordre causal pour accélérer la vitesse d'intégration des modifications et pour réduire les métadonnées envoyées.

Une autre contrainte généralement associée aux CRDTs synchronisés par opérations est la nécessité d'une livraison en exactement un exemplaire de chaque opération. Cette contrainte dérive de :

(i) La potentielle non-idempotence des opérations.

(ii) La nécessité de la livraison de chaque opération pour la livraison causale.

Toutefois, nous observons que des opérations peuvent être sémantiquement rendues obsolètes par d'autres opérations, e.g. une opération d'ajout d'un élément dans un Ensemble est rendue obsolète par une opération de suppression ultérieure du même élément. Ainsi, l'intergiciel de livraison peut se contenter d'assurer une livraison en un exemplaire au plus des opérations non-obsolètes. Ce choix permet de réduire la consommation réseau en évitant la diffusion d'opérations désormais non-pertinentes.

Pour compenser la perte d'opérations par le réseau et ainsi garantir la livraison à terme des opérations pertinentes, l'intergiciel de livraison des opérations doit mettre en place un mécanisme d'anti-entropie. Plusieurs mécanismes de ce type ont été proposés dans la littérature [31, 32, 33, 34] *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 1.8. 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

13. La relation de causalité étant transitive, les opérations et leurs relations de causalité forment un DAG. [30] 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.

par opérations, c.-à-d. que les modifications et opérations sont confondues, et qu'il autorise une livraison dans le désordre des opérations *add*.

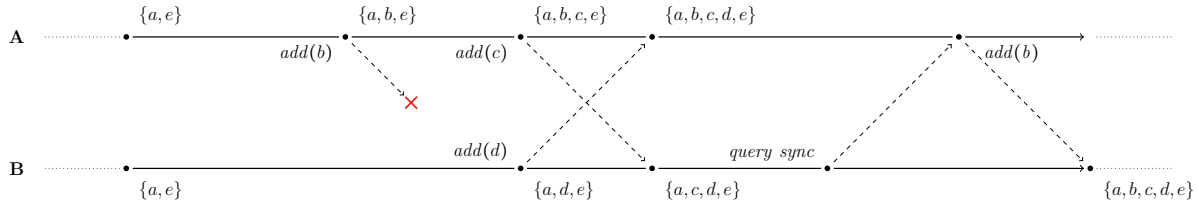


FIGURE 1.8 – Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par opérations

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.

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 [26, 35] 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 pour déterminer les opérations supprimables de manière sûre.

Définition 6 (Stabilité causale) *Une opération est stable causalement lorsqu'elle a été observée 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 de manière honnête .*

Un mécanisme d'instantané *Matthieu: TODO : Ajouter refs* doit néanmoins être associé au mécanisme de troncage du journal pour générer un état équivalent à celui résultant des opérations supprimées. 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 simplement un système fonctionnel à l'aide d'un CRDT synchronisé par opérations et d'un intergiciel de diffusion et de

livraison RCB. Mais comme illustré ci-dessus, chaque CRDT synchronisé par opérations établit les propriétés de ses différentes opérations et délègue potentiellement des responsabilités à l'intergiciel de diffusion et livraison. La complexité de cette approche réside ainsi dans l'ajustement du couple $\langle CRDT, intergiciel \rangle$ pour régler finement et optimiser leur fonctionnement en tandem. Des approches [26, 35] ont été proposées ces dernières années pour concevoir et structurer plus proprement ces composants et leurs relations, mais reposent encore sur une livraison causale des opérations. *Matthieu: TODO : Vérifier que c'est bien le cas dans [35]*

Synchronisation par différences d'états

Almeida, Shoker et Baquero [27] 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 vulnérables aux défaillances du réseau et nécessitent généralement pour pallier à cette vulnérabilité une livraison des opérations en exactement un exemplaire et respectant l'ordre causal.

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

Pour pallier aux faiblesses de chaque approche et allier le meilleur des deux mondes, les auteurs proposent les CRDTs synchronisés par différences d'états [27, 28, 36]. Il s'agit en fait d'une sous-famille des CRDTs synchronisés par états. Ainsi, comme ces derniers, ils disposent d'une fonction `merge` permettant de produire la LUB entre deux états, et qui est associative, commutative et idempotente.

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 [27, 28], fut ensuite précisée dans [36]. Dans cet article, Enes et al. précisent qu'utiliser des éléments irréductibles (cf. Définition 5) 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 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. Plusieurs de ces approches sont évaluées et comparées de manière expérimentale dans [36].

Nous illustrons cette approche avec la Figure 1.9. 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.

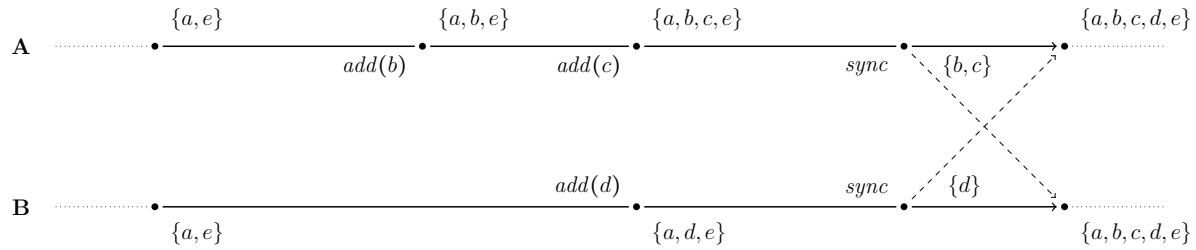


FIGURE 1.9 – Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par différences d'états

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 diffusion causale des modifications, les CRDTs synchronisés par différences d'états doivent donc définir et intégrer dans leur spécification un mécanisme similaire à l'intergiciel de livraison des CRDTs synchronisés par opérations. En revanche et à l'instar des CRDTs synchronisés par opérations, les CRDTs synchronisés par différences d'états peuvent aussi faire le choix inverse : s'affranchir du modèle de cohérence causale pour accélerer la diffusion des modifications et minimiser le surcoût du type répliqué.

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, qui peut s'avérer source d'erreurs et de coûts supplémentaires.

Synthèse

Nous récapitulons les principales propriétés et différences des modèles de synchronisations pour CRDTs dans Tableau 1.1.

TABLE 1.1 – Récapitulatif comparatif des différents modèles de synchronisation pour CRDTs

	Synchro. par états	Synchro. par opérations	Synchro. par diff. d'états
États forment sup-demi-treillis	✓	✓	✓
Intègre modifications par fusion d'états	✓	✗	✓
Intègre modifications de manière atomique	✗	✓	✓
Résistant aux défaillances réseau	✓	✗	✓
Peut s'affranchir de la cohérence causale	✗	✓	✓
Adapté pour systèmes temps réel	✗	✓	✓

1.2.3 Adoption dans la littérature et l'industrie

— Proposition et conception de CRDTs pour une variété de types de données : Registre, Compteur *Matthieu: TODO : Ajouter IPA*, Ensemble, Liste/Sequence, Graphe, JSON, Filesystem, Access Control. Propose généralement plusieurs sémantiques de résolution de conflits par type de données.

— Conception et développement de librairies mettant à disposition des développeurs d'applications des types de données composés [37, 38, 39, 40, 41] *Matthieu: TODO : Revoir et ajouter Melda (PaPoC'22) si fitting*

— Conception de langages de programmation intégrant des CRDTs comme types primitifs, destinés au développement d'applications distribuées [42, 43]

— Conception et implémentation de bases de données distribuées, relationnelles ou non, privilégiant la disponibilité et la minimisation de la latence à l'aide des CRDTs [44, 45, 46, 47, 48] *Matthieu: TODO : Ajouter Redis et Akka*

— Conception d'un nouveau paradigme d'applications, Local-First Software, dont une des fondations est les CRDTs [49, 50] *Matthieu: TODO : Vérifier et ajouter l'article avec Digital Garden (PaPoC'22 ?) si fitting*

— Éditeurs collaboratifs temps réel à large échelle et offrant de nouveaux scénarios de collaboration grâce aux CRDTs [51, 3]

1.3 Séquences répliquées sans conflits

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 1.10 notre représentation des séquences.

H	E	L	L	O
0	1	2	3	4

FIGURE 1.10 – Représentation de la séquence "HELLO"

Dans la Figure 1.11, nous présentons la spécification algébrique du type Séquence que nous utilisons.

$$
\begin{array}{lcll}
S \in Seq\langle V \rangle & & & \\
 & & & \\
emp & : & & \longrightarrow S \\
 & & & \\
ins & : & S \times N \times V & \longrightarrow S \\
rmv & : & S \times N & \longrightarrow S \\
 & & & \\
len & : & S & \longrightarrow N \\
rd & : & S & \longrightarrow Arr\langle V \rangle
\end{array}
$$

FIGURE 1.11 – Spécification algébrique du type abstrait usuel Séquence

Celle-ci définit deux modifications :

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

$$\forall s \in S, v \in V, i \in [0, m] \mid m = len(s) - 1, s = \langle v_0, ..., v_{i-1}, v_i, ..., v_m \rangle \cdot ins(s, i, v) = \langle v_0, ..., v_{i-1}, v, v_i, ..., v_m \rangle$$

(ii) *rmv*, 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 :

$$\forall s \in S, v \in V, i \in [0, m] \mid m = len(s) - 1, s = \langle v_0, ..., v_{i-1}, v_i, vi + 1, ..., v_m \rangle \cdot rmv(s, i) = \langle v_0, ..., v_{i-1}, v_{i+1}, ..., v_m \rangle$$

Les modifications définies dans Figure 1.11, *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.

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 1.12.

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

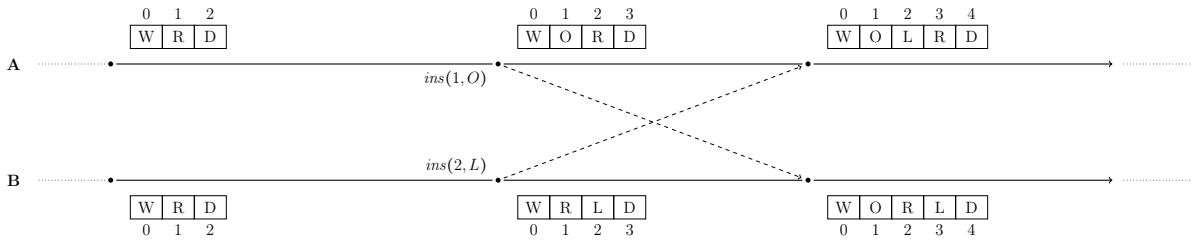


FIGURE 1.12 – Modifications concurrentes d'une séquence

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) [52, 53] 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 [54, 55, 56], 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 *Matthieu: TODO : Ajouter refs*, 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) [57] a été éprouvée à notre connaissance. *Matthieu: TODO : Vérifier que c'est pas plutôt les seules fonctions de transformations qui sont correctes et compatibles avec un système P2P.* 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 Pair-à-Pair (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) [58].

Définition 7 (Modèle CCI) *Le modèle de cohérence CCI définit qu'un système d'édition collaboratif doit respecter les critères suivants :*

Définition 7.1 (Convergence) *Le critère de* Convergence *indique que des noeuds ayant intégrés le même ensemble de modifications convergent à un état équivalent.*

Définition 7.2 (Causality preservation) *Le critère de* Causality preservation *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 7.3 (Intention preservation) *Le critère de* Intention preservation *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 [59], nous ajoutons le critère de *Scalability* à ces critères :

Définition 8 (Scalability) *Le critère de* Scalability *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 7.2 et 8 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 [60]. 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 [61], un modèle de séquence répliquée qui pose les fondations des CRDTs. Depuis, plusieurs CRDTs pour Séquence furent définies [62, 63, 59]. Ces CRDTs peuvent être répartis en deux approches : l'approche à pierres tombales [61, 62] et l'approche à identifiants densément ordonnés [63, 59]. 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 [5] 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.

1.3.1 Approche à pierres tombales

WOOT

WOOT [61] fait suite aux travaux présentés dans [57]. Il est considéré a posteriori comme le premier CRDT synchronisé par opérations pour Séquence [14]. 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.

14. [64] n'ayant formalisé les CRDTs qu'en 2007.

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.

Définition 9 (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 [65] 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 10 (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 1.13.

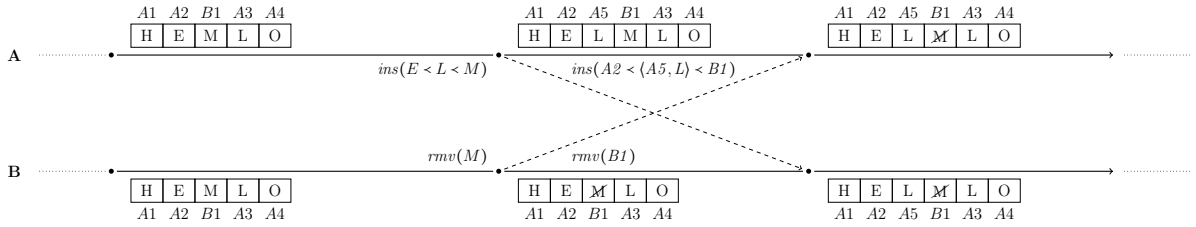


FIGURE 1.13 – Modifications concurrentes d'une séquence répliquée WOOT

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 11 (Modèle de livraison WOOT) *Le modèle de livraison WOOT définit que :*

(i) Une opération doit être livrée exactement une fois à chaque noeud[15].

15. 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*.

(ii) Une opération $ins(predId \prec \langle id, elt \rangle \prec succId)$ *ne peut être délivrée 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 délivrée 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. 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. [61] 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 [61].* Cette piste de recherche est développée dans [66].

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 [67]. Plusieurs évolutions de WOOT sont proposées pour mitiger cette limite : WOOTO [68] et WOOTH [67].

Weiss, Urso et Molli [68] 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. [67] 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 [67] 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 [69, 70]. 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) [62] 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'Operation Commutativity (OC).

Définition 12 (Operation Commutativity) *L'Operation Commutativity (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 Precedence Transitivity (PT).

Définition 13 (Precedence Transitivity) *La Precedence Transitivity (PT) se base sur une relation de précédence, notée* $\dashrightarrow$.

Définition 13.1 (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 on a* $o_1 \dashrightarrow o_3$.

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 14 (Identifiants S4Vector) *Les identifiants S4Vector sont de la forme* $\langle ssid, sum, ssn,$ *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 [16, 71], les auteurs utilisent des horloges de Lamport [15] 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 15 (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 1.14.

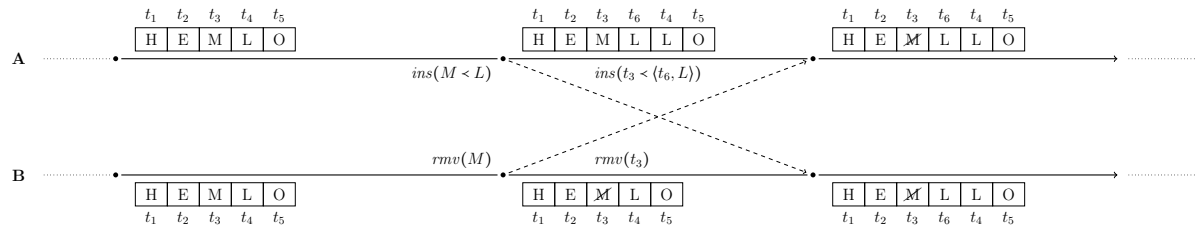


FIGURE 1.14 – Modifications concurrentes d'une séquence répliquée RGA

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. [62] jugent le coût des pierres tombales trop élévé. Ils proposent alors un mécanisme de Garbage Collection (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 observé 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, [62] 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 [5] indique que RGA pourrait adopter un modèle de livraison similaire à celui de WOOT. Ce modèle consisterait :

Définition 16 (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 délivrée 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 délivrée à un noeud qu'après la livraison de l'opération d'insertion de l'élément associé à id.

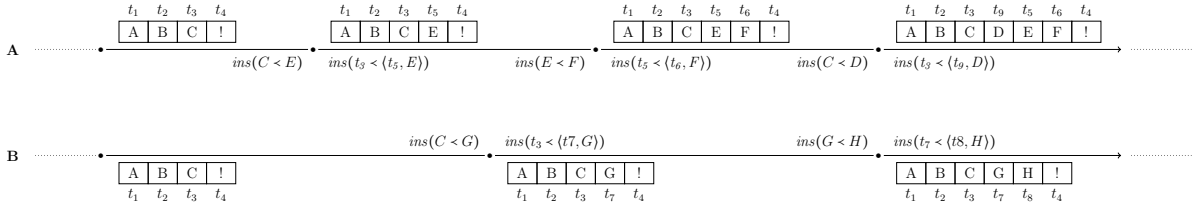
Nous secondons cette observation.

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 [67]. De plus, [71, 72] 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 [41].

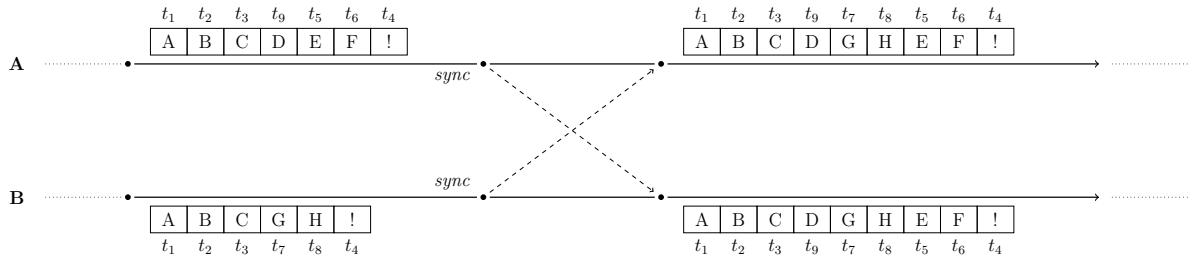
Plusieurs extensions de RGA ont par la suite été proposées. Briot, Urso et Shapiro [73] indiquent que les pauvres performances des modifications locales [16] des CRDTs pour Séquence constituent une de leurs limites. Il s'agit en effet des performances impactant le plus l'expérience utilisateur, celleux-ci 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 [17] tels que proposés par [74, 4], qui permet de réduire la quantité de métadonnées stockées par la séquence répliquée. Cette combinaison aboutit à la définition d'un nouveau CRDT pour Séquence, *RGATreeSplit*, qui offre une meilleure complexité en temps et en espace.

Dans [75], 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 1.15 présente un tel cas de figure :



(a) Modifications concurrentes d'une séquence répliquée RGA



(b) Synchronisation résultant en un entrelacement des modifications concurrentes

Figure 1.15 – Entrelacement d'éléments insérés de manière concurrente

Dans la Figure 1.15a, 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

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

"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 1.15b. 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 [71, 72], cette nouvelle spécification précise que les éléments insérés en concurrence ne doivent pas s'entrelacer dans l'état final. Kleppmann et al. [75] 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 [75] que cette extension empêche tout entrelacement [18].

1.3.2 Approche à identifiants densément ordonnés

Treedoc

[64, 63] 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 17 (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 $\mathbb{I}$.

18. Un travail en cours [76] indique en effet qu'une séquence répliquée empêchant tout entrelacement est impossible.

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

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

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 $rmv(id)$.

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

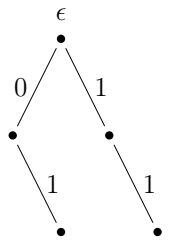


Figure 1.16 – Identifiants de positions

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 1.16, 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 9). 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 1.17 présente un exemple de cette situation. Dans cet exemple, deux identifiants

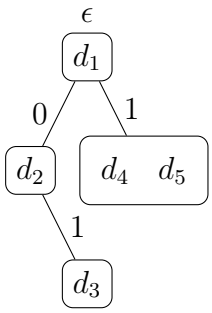


FIGURE 1.17 – Identifiants de position avec désambiguateurs

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.

(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 1.18. 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.

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

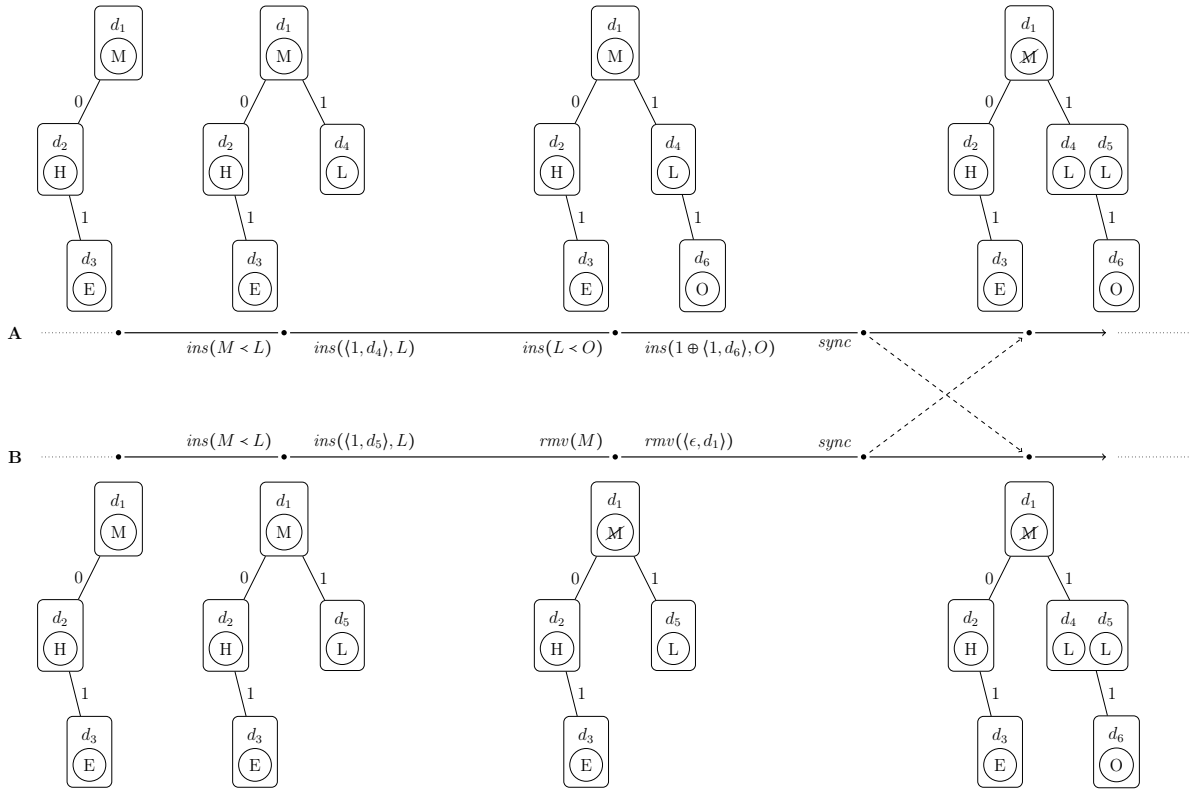


FIGURE 1.18 – Modifications concurrentes d'une séquence répliquée Treedoc

élément avec son désambiguateur dans noeud de chemin 1, après (resp. avant) l'élément existant (on considère que $d_4 \prec d_5$).

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 [63], les auteurs indiquent reposer sur le modèle de livraison causal. En pratique, nous pouvons néanmoins relaxer le modèle de livraison comme expliqué dans [5] :

Définition 18 (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 délivrées dans un ordre quelconque.

(iii) L'opération $rmv(id)$ ne peut délivrée 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 1.18, 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 [63] 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 [75]. En effet, nous présentons un contre-exemple correspondant dans l'Annexe A.

Logoot

En parallèle à Treedoc [63], Weiss, Urso et Molli [59] 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 17.

Dans [59], les identifiants de positions sont décrits de la manière suivante :

Définition 19 (Identifiant Logoot v1) *Un identifiant Logoot est une paire* $\langle tuples, seq \rangle$ *avec*

- *tuples, une liste de tuples Logoot,*
- *seq, le numéro de séquence courant du noeud auteur de l'élément.*

avec les tuples Logoot définis de la manière suivante :

Définition 19.1 (Tuple Logoot v1) *Un tuple Logoot est une paire* $\langle pos, nodeId \rangle$ *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.

Par la suite, [77] re-spécifie les identifiants de positions de la manière suivante :

Définition 20 (Identifiant Logoot v2) *Un identifiant Logoot est une liste de tuples Logoot avec les tuples Logoot définis de la manière suivante :*

Définition 20.1 (Tuple Logoot v2) *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 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 21 (Relation $<_{id}$) *Étant donné deux identifiants* $id = t_1 \oplus t_2 \oplus ... \oplus t_n$ *et* $id' = t'_1 \oplus t'_2 \oplus ... \oplus t'_m$*, on a :*

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

avec $<_t$ *défini de la manière suivante :*

Définition 21.1 (Relation $<_t$) *Étant donné deux tuples* $t = \langle pos, nodeId, seq\rangle$ *et* $t' = \langle pos', nodeId', seq'\rangle$*, on a :*

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

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, [59] 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`.

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*

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
```

n'ayant pas de troisième tuple, `generateId` utilise la valeur minimale pour *pos*, $\bot_{\mathbb{N}}$. `generateId` choisit donc une valeur aléatoire dans $]\bot_{\mathbb{N}}, \top_{\mathbb{N}}]$ [19], 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 1.19.

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.

19. 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 $id' <_{id} id$.

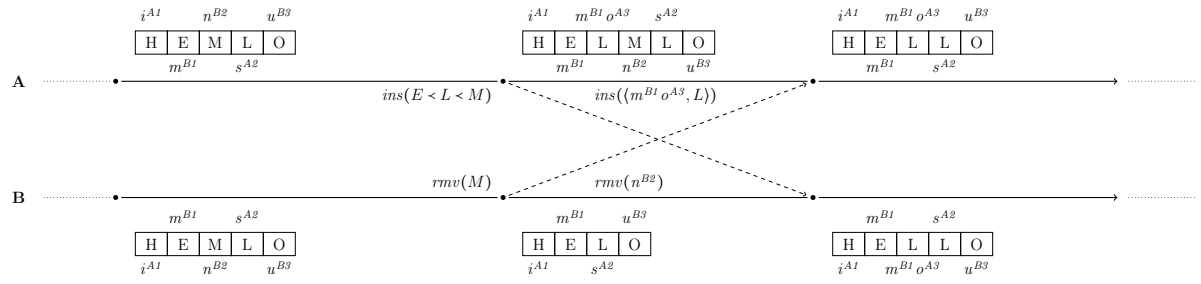


FIGURE 1.19 – Modifications concurrentes d'une séquence répliquée Logoot

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".

Concernant le modèle de livraison de Logoot, [59] indique se reposer sur le modèle de livraison causal. Nous constatons cependant que nous pouvons proposer un modèle de livraison moins contraint :

Définition 22 (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 délivrées dans un ordre quelconque.

(iii) L'opération rmv(id) ne peut délivrée 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 [5].

En contrepartie, Logoot souffre d'un problème de croissance de la taille des identifiants. Comme mis en lumière dans la Figure 1.19, 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, Urso et Molli [77] 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 [67]. 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 [77], 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 [20]. 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. [4] introduisent *LogootSplit*. Reprenant les idées introduites par [74], 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 1.4 à sa présentation en détails.

Matthieu: TODO : Autres Sequence CRDTs à considérer : String-wise CRDT [74], Chronofold [78]

1.3.3 Synthèse

— Deux approches différentes pour la résolution de conflits ont été proposées pour CRDTs pour Séquence. Chacune de ces approches visent à minimiser surcoût du type répliqué, que ce soit d'un point de vue mémoire, computations et réseau.

— Au fil des années, ces approches ont été raffinées avec de nouveaux CRDTs de plus en plus en efficaces.

20. 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).

— Néanmoins, malgré les évaluations et comparaisons, la littérature n'a pas établi une supériorité d'une approche sur l'autre. Les approches proposent seulement des compromis différents sur la nature du surcoût, que nous récapitulons dans Tableau 1.2. L'approche basée sur pierres tombales offre une consommation réseau constante grâce à ses identifiants de taille fixe, mais souffre d'une consommation mémoire ne pouvant qu'augmenter. L'approche basée sur identifiants densément ordonnés bénéficie d'un meilleur délai de diffusion des modifications, les modifications pouvant être livrées dans le désordre, mais souffre d'une empreinte réseau augmentant avec la taille de ses identifiants. L'approche basée sur pierres tombales et l'approche basée sur identifiants densément ordonnés souffrent toutes les deux d'une augmentation théorique de leur surcoût en mémoire et en computations, respectivement dûe au nombre forcément croissant d'éléments stockées dans la Séquence et à la taille croissante [21] des identifiants associés aux éléments de la Séquence.

TABLE 1.2 – Récapitulatif comparatif des différents approches pour CRDTs pour Séquence

	Pierres tombales	Identifiants densément ordonnés
Performances stables en fct. de la taille de la séq.	✗	✗
Identifiants de taille fixe	✓	✗
Eléments réellement supprimés	✗	✓
Taille des messages fixe	✓	✗
Peut s'affranchir de la cohérence causale	✓	✓

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

1.4 LogootSplit

LogootSplit [4] est l'état de l'art des séquences répliquées à identifiants densément ordonnés. Comme expliqué précédemment, LogootSplit utilise des identifiants provenant d'un ordre total dense pour positionner les éléments dans la séquence répliquée.

1.4.1 Identifiants

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

Définition 23 (Tuple) *Un* Tuple *est un quadruplet* ⟨*position, nodeId, nodeSeq, offset*⟩ *où*

— *position incarne la position souhaitée de l'élément.*

— *nodeId est l'identifiant unique du noeud qui a généré le tuple.*

21. [67] montre expérimentalement que les performances de l'approche basée sur identifiants densément ordonnés restent stables tout au long des tâches d'édition collaborative proposées.

— *nodeSeq est le numéro de séquence courant du noeud à la génération du tuple.*

— *offset indique la position de l'élément au sein d'un bloc. Nous reviendrons plus en détails sur ce composant dans la sous-section 1.4.2.*

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

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

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

Définition 24 (Identifiant) *Un* Identifiant *est une liste de* Tuples.

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

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

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

Définition 25 (Relation $<_{id}$) *La relation $<_{id}$ est un ordre strict total sur l'ensemble des identifiants. Elle permet aux noeuds de comparer n'importe quelle paire d'identifiants. Elle est définie en utilisant l'ordre lexicographique sur les composants des différents tuples des identifiants comparés.*

— En utilisant cette relation d'ordre, les noeuds peuvent ordonner les éléments grâce à leur identifiant.

— Par exemple, déterminent que $i_0^{A1} <_{id} i_0^{B0}$ car les positions sont identiques et que le *nodeId* (A) du premier est plus petit que le *nodeId* (B) du second

— et que $i_0^{B0} <_{id} i_0^{B0} f_0^{A0}$ car le premier est un préfixe du second

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

1.4.2 Aggrégation dynamique d'élements en blocs

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

Définition 26 (IdInterval) *Un* IdInterval *est un couple $\langle$idBegin, offsetEnd$\rangle$ où*

— *idBegin est l'identifiant du premier élément de l'interval.*

— *offsetEnd est l'offset du dernier identifiant de l'interval.*

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

LogootSplit regroupe les éléments avec des identifiants *contigus* dans un interval. Nous appelons *contigus* deux identifiants qui partagent une même base (c.-à-d. qui sont identiques à l'exception de leur dernier offset) et dont les *offsets* sont consécutifs. Nous représentons un 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.

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

Définition 27 (Bloc) *Un* Bloc *est un quadruplet* ⟨*idInterval, elts, isAppendable, isPrependable*⟩ *où*

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

La Figure 1.20 présente un exemple de séquence LogootSplit : dans la Figure 1.20a, les identifiants i_0^{B0}, i_1^{B0}, i_2^{B0} forment une chaîne d'identifiants contigus. LogootSplit est donc capable de regrouper ces éléments en un bloc représentant l'intervalle d'identifiants $i_{0..2}^{B0}$ pour minimiser les métadonnées stockées, comme montré dans la Figure 1.20b.

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

FIGURE 1.20 – Représentation d'une séquence LogootSplit contenant les éléments "HLO"

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

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

Notons que pour une séquence donnée, nous pouvons identifier chacun de ses identifiants par le triplet ⟨nodeId, nodeSeq, offset⟩ issue de leur dernier Tuple. Par exemple, le triplet ⟨B, 0, 2⟩ désigne de manière unique l'identifiant i_2^{B0} dans Figure 1.20.

1.4.3 Modèle de données

André et al. [4] définissent une séquence LogootSplit de la manière suivante :

Définition 28 (Séquence LogootSplit) *Une séquence* Séquence LogootSplit *est un triplet* ⟨*nodeId, nodeSeq, blocks*⟩ *où*

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

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

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

Définition 29 (insert) *Une opération* insert *est un couple* ⟨*id, elts*⟩ *où*

- *id est l'identifiant du premier élément inséré par cette opération.*
- *elts est la liste des éléments insérés par cette opération.*

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

Définition 30 (remove) *Une opération* remove *est une liste d'intervalles d'identifiants où chaque intervalle désigne un ensemble d'éléments à supprimer.*

Nous présentons dans la Figure 1.21 un exemple d'utilisation de cette séquence répliquée.

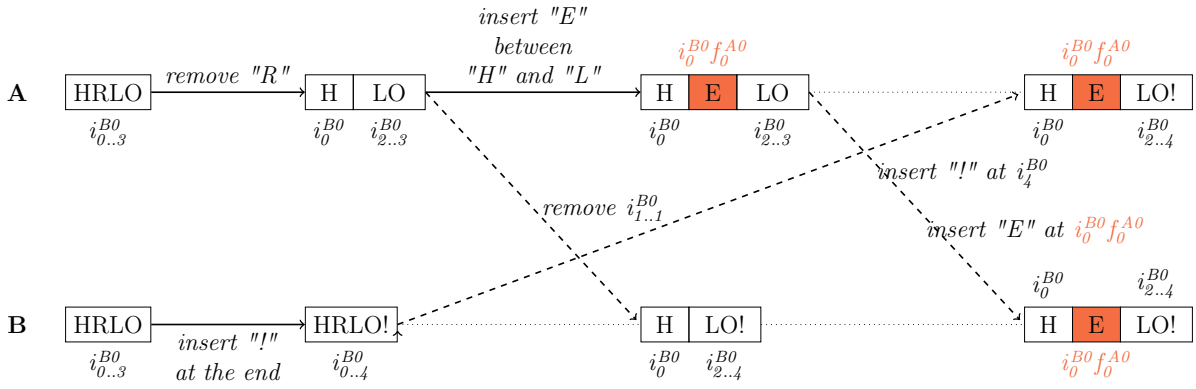


FIGURE 1.21 – Modifications concurrentes d'une séquence répliquée LogootSplit

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 $i^{B0}_{0..3}$ aux éléments "HRLO". Les noeuds se mettent ensuite à éditer le document.

Le noeud A commence par supprimer l'élément "R" de la séquence. LogootSplit génère l'opération *remove* correspondante en utilisant l'identifiant de l'élément supprimé (i^{B0}_1). Cette opération est envoyée au noeud B pour qu'il intègre cette modification.

Le noeud A insère ensuite un élément "E" dans la séquence, entre le "H" et le "L". LogootSplit doit alors générer un identifiant *id* à associer à ce nouvel élément. Ce nouvel identifiant *id* doit respecter la contrainte suivante : $i^{B0}_0 <_{id} id <_{id} i^{B0}_2$. Cependant, LogootSplit ne peut pas générer un identifiant composé d'un seul tuple respectant cet ordre. LogootSplit génère alors *id* en recopiant le premier tuple (i^{B0}_0) et en y ajoutant un nouveau tuple (f^{A0}_0). LogootSplit génère l'opération *insert* correspondante, indiquant l'élément à insérer et sa position grâce à son identifiant. Cette opération est ensuite diffusée sur le réseau.

En parallèle, le noeud B insère un élément " !" à la fin de la séquence. Comme le noeud B est l'auteur du bloc $i^{B0}_{0..3}$, il peut y ajouter de nouveaux éléments. LogootSplit associe donc l'identifiant i^{B0}_4 à l'élément " !" et l'ajoute au bloc existant.

Les noeuds se synchronisent ensuite. Le noeud A reçoit l'opération *insert* de l'élément " !" à la position i^{B0}_4. Le noeud A détermine que cet élément doit être inséré à la fin de la séquence (puisque $i^{B0}_3 <_{id} i^{B0}_4$) et qu'il peut être ajouté au bloc $i^{B0}_{2..3}$ (puisque i^{B0}_3 et i^{B0}_4 sont contigus).

De son côté, le noeud B reçoit tout d'abord l'opération *remove* des éléments identifiés par l'intervalle $i^{B0}_{1..1}$, c.-à-d. l'élément attaché à l'identifiant i^{B0}_1. Le noeud B supprime donc l'élément "R" de son état.

Il reçoit ensuite l'opération *insert* de l'élément "E" à la position $i^{B0}_0 f^{A0}_0$. Le noeud B insère cet élément entre les éléments "H" et "L" (puisque $i^{B0}_0 <_{id} i^{B0}_0 f^{A0}_0 <_{id} i^{B0}_2$), respectant ainsi l'intention du noeud A.

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

1.4.4 Modèle de livraison

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

Livraison des opérations en exactement un exemplaire

Tout d'abord, la couche de livraison de messages doit assurer que toutes les opérations soient délivrées aux noeuds, mais qu'une seule et unique fois. La Figure 1.22 représente un exemple illustrant la nécessité de cette contrainte.

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

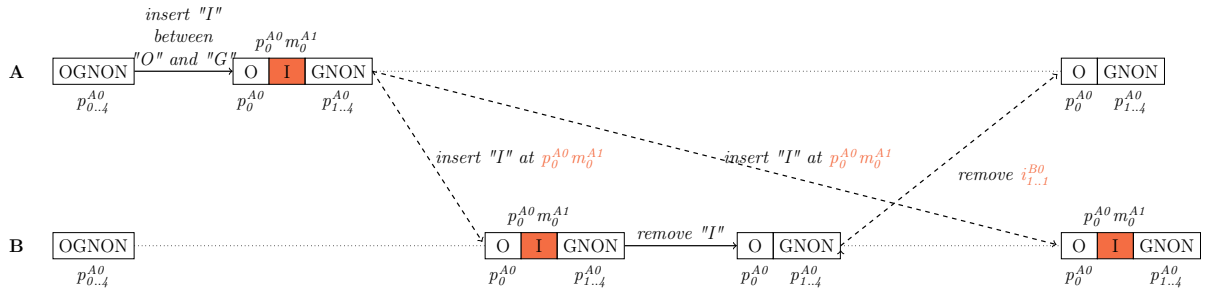


FIGURE 1.22 – Résurgence d'un élément supprimé suite à la relivraison de son opération *insert*

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

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

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

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

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

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

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

Une autre propriété que doit assurer la couche de livraison de messages est que les opérations *remove* doivent être livrées au CRDT après les opérations *insert* correspondantes. La Figure 1.23 présente un exemple justifiant cette contrainte.

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

À la réception de l'opération *insert*, le noeud B l'intègre à son état. Cependant, le noeud

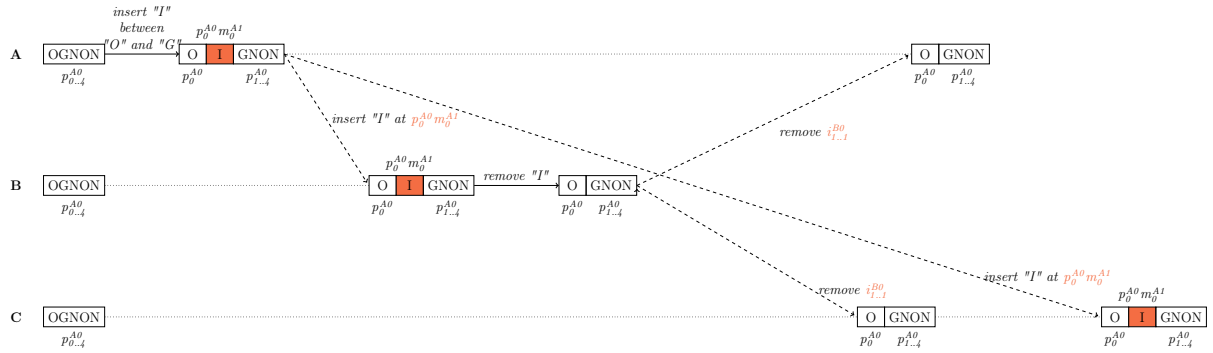


FIGURE 1.23 – Non-effet de l'opération *remove* car reçue avant l'opération *insert* correspondante

B supprime dans la foulée l'élément nouvellement ajouté. Il diffuse ensuite l'opération *remove* générée.

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

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

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

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

Définition du modèle de livraison

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

Définition 31 (Exactly-once + Causal remove) *Le modèle de livraison* Exactly-once + Causal remove *définit les 3 règles suivantes sur la livraison des opérations :*

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

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

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

Il est à noter que Elvinger [5] a récemment proposé dans ses travaux de thèse Dotted LogootSplit, un nouveau Sequence CRDT basée sur les différences. Inspiré de Logoot et LogootSplit, ce nouveau CRDT associe une séquence à identifiants densément ordonnés à un contexte causal. Le contexte causal est une structure de données permettant à Dotted LogootSplit de représenter et de maintenir efficacement les informations des modifications déjà intégrées à l'état courant. Cette association permet à Dotted LogootSplit de fonctionner de manière autonome, sans imposer de contraintes particulières à la couche livraison autres que la livraison à terme.

1.4.5 Limites

Comme indiqué précédemment, la taille des identifiants provenant d'un ordre total dense est variable. Quand les noeuds insèrent de nouveaux éléments entre deux autres ayant la même valeur de *position*, LogootSplit n'a pas d'autre choix que d'augmenter la taille de l'identifiant résultant. La Figure 1.24 illustre de tels cas. Dans cet exemple, puisque le noeud A insère un nouvel élément entre deux identifiants contigus i_0^{B0} et i_1^{B0}, LogootSplit ne peut pas générer un identifiant adapté de la même taille. Pour respecter l'ordre souhaité, LogootSplit génère un identifiant en ajoutant un nouveau tuple à l'identifiant du prédecesseur : $i_0^{B0} f_0^{A0}$.

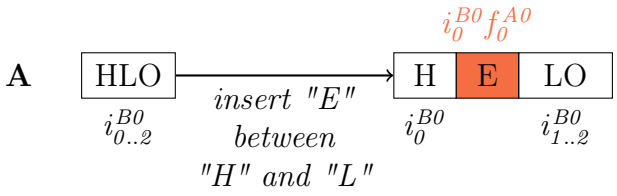


Figure 1.24 – Insertion menant à une augmentation de la taille des identifiants

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

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

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

Comme illustré plus loin, nous avons mesuré au cours de nos évaluations que le contenu représente à terme moins de 1% de taille de la structure de données. Les 99% restants

correspondent aux métadonnées utilisées par la séquence répliquée. Il est donc nécessaire de proposer des mécanismes et techniques afin de mitiger les problèmes soulignés précédemments.

1.5 Mitigation du surcoût des séquences répliquées sans conflits

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

— RGA (et RGASplit) propose un mécanisme de GC des pierres tombales. Nécessite cependant stabilité causale des opérations de suppression. S'agit d'une contrainte forte, peu adaptée aux systèmes dynamiques à large échelle. *Matthieu: TODO : Trouver référence sur la stabilité causale dans systèmes dynamiques*

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

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

1.6 Synthèse

— Systèmes distribués adoptent le modèle de la réplication optimiste pour offrir de meilleures performances, c.-à-d. disponibilité et latence, et assurer la résilience du système, c.-à-d. accroître la capacité de tolérance aux pannes.

— Ce modèle autorise les noeuds à modifier leur copie sans coordination, provoquant ainsi des divergences temporaires. Pour résoudre les potentiels conflits et assurer la convergence à terme des copies, systèmes utilisent les CRDTs en place et lieu des types de données séquentiels.

— CRDTs pour Séquence ont été proposés pour conception d'éditeurs collaboratifs pair-à-pair. Deux approches sont utilisées pour concevoir leur mécanismes de résolution de conflits : l'approche basée sur les pierres tombales et l'approche basée sur les identifiants densément ordonnés.

— Chacune de ces approches introduit un surcoût croissant, pénalisant leurs performances à terme. Plusieurs travaux ont été proposés pour limiter ce surcoût, notamment [1, 2] qui présentent un mécanisme de renommage des identifiants pour les CRDTs pour Séquence basés sur identifiants densément ordonnés.

— Mais cette approche requiert un mécanisme de consensus, des renommages concurrents provoquant un nouveau conflit. Cette contrainte empêche son utilisation dans les systèmes pair-à-pair ne disposant pas de noeuds suffisamment stables et bien connectés pour exécuter le mécanisme de consensus.

1.7 Proposition

— Dans ce manuscrit, nous proposons et présentons un nouveau mécanisme de renommage pour CRDTs pour Séquence, ne nécessitant pas de coordination synchrone entre les noeuds.

— Concevons ce mécanisme pour le CRDT pour Séquence LogootSplit, mais principe de notre approche est générique. Pourrait ainsi l'adapter et proposer un équivalent pour autres CRDTs pour Séquence, e.g. RGASplit.

— Présentons et détaillons notre contribution dans le chapitre suivant.

Chapitre 2

Conclusions et perspectives

Sommaire

2.1 Résumé des contributions

2.2 Perspectives

2.2.1 Définition de relations de priorité pour minimiser les traitements

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

2.2.3 Définition de types de données répliquées sans conflits plus complexes

2.2.4 Étude comparative des différentes familles de CRDTs

— La spécification récente des Delta-based CRDTs . Ce nouveau type de CRDTs se base sur celui des State-based CRDTs. Partage donc les mêmes pré-requis :

  — États du type de données répliqué forment un sup-demi-treillis

  — Modifications locales entraînent une inflation de l'état

  — Possède une fonction de `merge`, permettant de fusionner deux états S et S', et qui

    — Est associative, commutative et idempotente

    — Retourne S", la LUB de S et S' (c.-à-d. $\nexists S''' \cdot merge(S, S') < S''' < S''$)

  Et bénéficie de son principal avantage : synchronisation possible entre deux pairs en fusionnant leur états, peu importe le nombre de modifications les séparant.

— Spécificité des Delta-based CRDTs est de proposer une synchronisation par différence d'états. Plutôt que de diffuser l'entièreté de l'état pour permettre aux autres pairs de se mettre à jour, idée est de seulement transmettre la partie de l'état ayant été mise à jour. Correspond à un élément irréductible du sup-demi-treillis. Permet ainsi de mettre en place une synchronisation en temps réel de manière efficace. Et d'utiliser la synchronisation par fusion d'états complets pour compenser les défaillances du réseau

— Ainsi, ce nouveau type de CRDTs semble allier le meilleur des deux mondes :

  — Absence de contrainte sur le réseau autre que la livraison à terme

  — Propagation possible en temps réel des modifications

  Semble donc être une solution universelle :

  — Utilisable peu importe la fiabilité réseau à disposition

  — Empreinte réseau du même ordre de grandeur qu'un Op-based CRDT

  — Utilisable peu importe la fréquence de synchronisation désirée

  Pose la question de l'intérêt des autres types de CRDTs.

— Delta-based CRDT est un State-based CRDT dont on a identifié les éléments irréductibles et qui utilise ces derniers pour la propagation des modifications plutôt que l'état complet. Famille des State-based CRDTs semble donc rendue obsolète par celle des Delta-based CRDTs. À confirmer.

— Les Op-based CRDTs proposent une spécification différente du type répliqué de leur équivalent Delta-based, généralement plus simple. À première vue, famille des Op-based CRDTs semble donc avoir la simplicité comme avantage par rapport à celle des Delta-based CRDTs. S'agit d'un paramètre difficilement mesurable et auquel on peut objecter si on considère qu'un Op-based CRDT s'accompagne d'une couche livraison de messages, qui cache sa part de complexité. Intéressant d'étudier si la spécification différente des Op-based CRDTs présente d'autres avantages par rapport aux Delta-based CRDTs : performances (temps d'intégration des modifications, délai de convergence...), fonctionnalités spécifiques (composition, undo...)

— But serait de fournir des guidelines sur la famille de CRDT à adopter en fonction du cas d'utilisation.

2.2.5 Définition d'opérations supplémentaires pour fonctionnalités liées à l'édition collaborative

— Commentaires

— Suggestions

2.2.6 Conduction d'expériences utilisateurs d'édition collaborative

— Absence d'un dataset réel et réutilisable sur les sessions d'édition collaborative

— Généralement, expériences utilisent données d'articles de Wikipédia *Matthieu: TODO : Revoir références, mais me semble que c'est celui utilisé pour Logoot, LogootSplit et RGASplit entre autres*. Mais ces données correspondent à une exécution séquentielle, c.-à-d. aucune édition concurrente ne peut être réalisée avec le système de résolution de conflits de Wikipédia. *Matthieu: TODO : Me semble que Kleppmann a aussi utilisé et mis à disposition ses traces correspondant à la rédaction d'un de ses articles. Mais que cet article n'était rédigé que par lui. Peu de chances de présence d'éditions concurrentes. À retrouver et vérifier.*

— Inspiré par expériences de Claudia, pourrait mener des sessions d'édition collaborative sur des outils orchestrés pour produire ce dataset

— Devrait rendre ce dataset agnostique de l'approche choisie pour la résolution automatique de conflits

— Absence de retours sur les collaborations à grande échelle

— Comment on collabore lorsque plusieurs centaines d'utilisateur-rices ?

2.2.7 Comparaison des mécanismes de synchronisation

Serait intéressant de comparer à d'autres méthodes de synchronisation : mécanisme d'anti-entropie basé sur un Merkle Tree[32, 33, 34], synchronisation par états (state/delta-based CRDTs). Dans le cadre des Delta-based CRDTs, pourrait évaluer un protocole de diffusion épidémique des deltas comme celui proposé par SWIM[8].

2.2.8 Distance entre versions d'un document

- Est-ce que ça a vraiment du sens d'intégrer automatiquement des modifications ayant été généré sur une version du document distante de l'état actuel du document (voir distance de Hamming, Levenstein, String-to-string correction problem (Tichy et al))
- Jusqu'à quelle distance est-ce que la fusion automatique a encore du sens ? *Matthieu: NOTE : Peut connecter ça à la nécessité de conserver un chemin d'une époque à l'autre : si les opérations émises depuis cette époque ont probablement plus d'intérêt pour l'état actuel, couper l'arbre ?*

2.2.9 Contrôle d'accès

- Pour le moment, n'importe quel utilisateur ayant l'URL du document peut y accéder dans MUTE
- Pour des raisons de confidentialité, peut vouloir contrôler quels utilisateurs ont accès à un document
- Nécessite l'implémentation de liste de contrôle d'accès
- Mais s'agit d'une tâche complexe dans le cadre d'un système distribué
- Peut s'inspirer des travaux réalisés au sein de la communautée CRDTs [79, 80] pour cela

2.2.10 Détection et éviction de pairs malhonnêtes

- À l'heure actuelle, MUTE suppose qu'ensemble des collaborateurs honnêtes
- Vulnérable à plusieurs types d'attaques par des adversaires byzantins, tel que l'équivoque
- Ce type d'attaque peut provoquer des divergences durables et faire échouer des collaborations
- Elvinger [5] propose un mécanisme permettant de maintenir des logs authentifiés dans un système distribué
- Les logs authentifiés permettent de mettre en lumière les comportements malveillants des adversaires et de borner le nombre d'actions malveillantes qu'ils peuvent effectuer avant d'être évincé
- Implémenter ce mécanisme permettrait de rendre compatible MUTE avec des environnements avec adversaires byzantins

— Nécessiterait tout de même de faire évoluer le CRDT pour résoudre les équivoques détectés

2.2.11 Vecteur *epoch-based*

— Comme présenté précédemment, nous utilisons plusieurs vecteurs pour représenter des données dans l'application MUTE

— Notamment pour le vecteur de version, utilisé pour respecter le modèle de livraison requis par le CRDT

— Et pour la liste des collaborateurs, utilisé pour offrir des informations nécessaires à la conscience de groupe aux utilisateurs

— Ces vecteurs sont maintenus localement par chacun des noeuds et sont échangés de manière périodique

— Cependant, la taille de ces vecteurs croit de manière linéaire au nombre de noeuds impliqués dans la collaboration

— Les systèmes P2P à large échelle sont sujets au *churn*

— Dans le cadre d'un tel système, ces structures croissent de manière non-bornée

— Ceci pose un problème de performances, notamment d'un point de vue consommation en bande-passante

— Cependant, même si on observe un grand nombre de pairs différents dans le cadre d'une collaboration à large échelle

— Intuition est qu'une collaboration repose en fait sur un petit noyau de collaborateurs principaux

— Et que majorité des collaborateurs se connectent de manière éphèmère

— Serait intéressant de pouvoir réduire la taille des vecteurs en oubliant les collaborateurs éphèmères

— Dynamo[32] tronque le vecteur de version lorsqu'il dépasse une taille seuil

— Conduit alors à une perte d'informations

— Pour la liste des collaborateurs, approche peut être adoptée (pas forcément gênant de limiter à 100 la taille de la liste)

— Mais pour vecteur de version, conduirait à une relivraison d'opérations déjà observées

— Approche donc pas applicable pour cette partie

— Autre approche possible est de réutiliser le système d'époque

— Idée serait de ACK un vecteur avec un changement d'époque

— Et de ne diffuser à partir de là que les différences

— Un mécanisme de transformation (une simple soustraction) permettrait d'obtenir le dot dans la nouvelle époque d'une opération concurrente au renommage

— Peut facilement mettre en place un mécanisme d'inversion du renommage (une simple addition) pour revenir à une époque précédente

— Et ainsi pouvoir circuler librement dans l'arbre des époques et gérer les opérations *rename* concurrentes

— Serait intéressant d'étudier si on peut aller plus loin dans le cadre de cette structure de données et notamment rendre commutatives les opérations de renommage concurrentes

2.2.12 Fusion de versions distantes d'un document collaboratif

2.2.13 Rôles et places des bots dans systèmes collaboratifs

— Stockage du document pour améliorer sa disponibilité

— Overleaf en P2P ?

— Comment réinsérer des bots dans la collaboration sans en faire des éléments centraux, sans créer des failles de confidentialité, et tout en rendant ces fonctionnalités accessibles ?

Annexe A

Entrelacement d'insertions concurrentes dans Treedoc

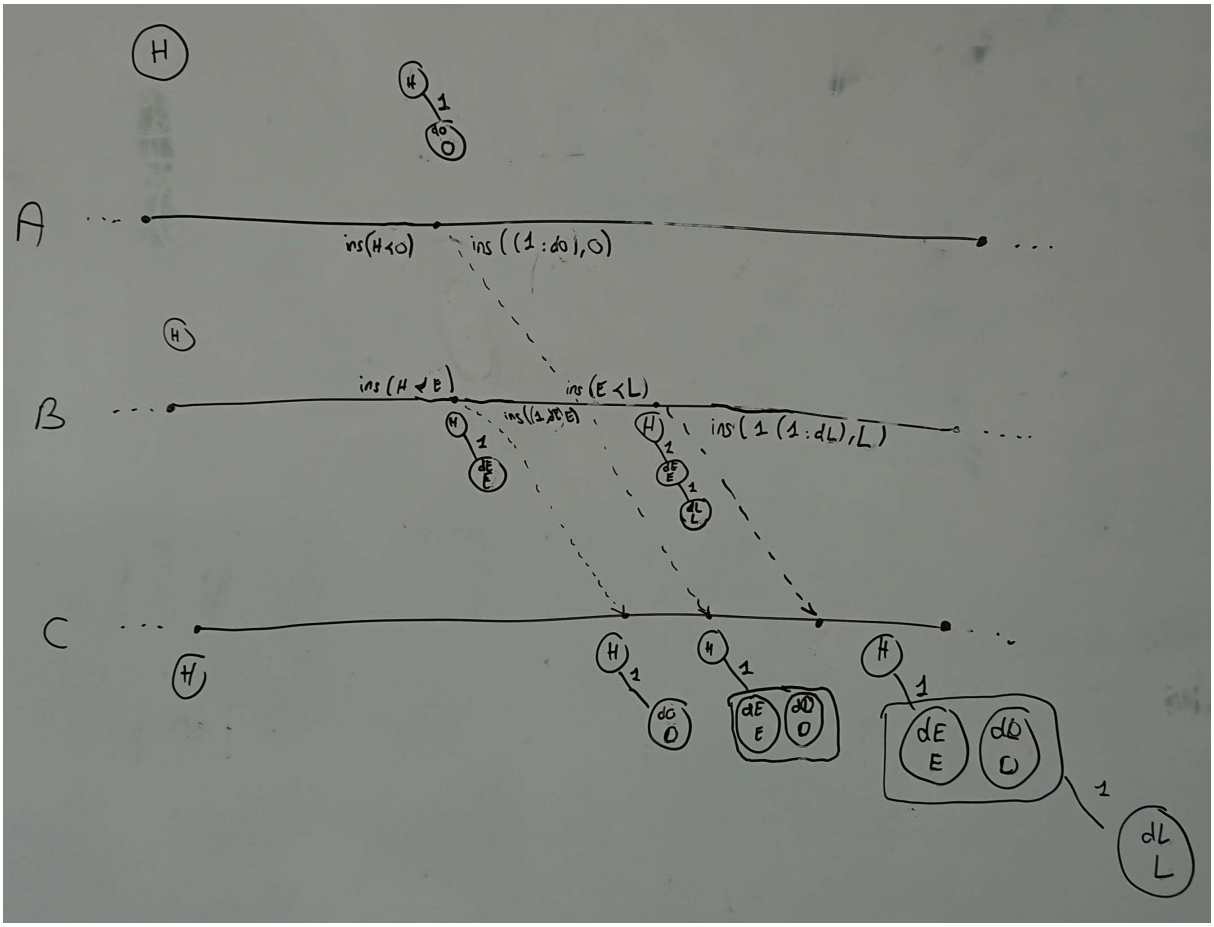


FIGURE A.1 – Modifications concurrentes d'une séquence Treedoc résultant en un entrelacement

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

Bibliographie

[1] 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.

[2] 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.

[3] Matthieu Nicolas et al. « MUTE : A Peer-to-Peer Web-based Real-time Collaborative Editor ». In : *ECSCW 2017 - 15th European Conference on Computer-Supported Cooperative Work*. T. 1. Proceedings of 15th European Conference on Computer-Supported Cooperative Work - Panels, Posters and Demos 3. Sheffield, United Kingdom : EUSSET, août 2017, p. 1–4. DOI : 10.18420/ecscw2017_p5. URL : https://hal.inria.fr/hal-01655438.

[4] Luc André et al. « Supporting Adaptable Granularity of Changes for Massive-Scale Collaborative Editing ». In : *International Conference on Collaborative Computing : Networking, Applications and Worksharing - CollaborateCom 2013*. Austin, TX, USA : IEEE Computer Society, oct. 2013, p. 50–59. DOI : 10.4108/icst.collaboratecom.2013.254123.

[5] 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.

[6] 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.

[7] Hoang-Long Nguyen et al. « Blockchain-Based Auditing of Transparent Log Servers ». In : *32th IFIP Annual Conference on Data and Applications Security and Privacy (DBSec)*. Sous la dir. de Florian Kerschbaum et Stefano Paraboschi. T. LNCS-10980. Data and Applications Security and Privacy XXXII. Part 1 : Administration. Bergamo, Italy : Springer International Publishing, juil. 2018, p. 21–37. DOI : 10.1007/978-3-319-95729-6_2. URL : https://hal.archives-ouvertes.fr/hal-01917636.

[8] A. Das, I. Gupta et A. Motivala. « SWIM : scalable weakly-consistent infection-style process group membership protocol ». In : *Proceedings International Conference on Dependable Systems and Networks*. 2002, p. 303–312. doi : 10.1109/DSN.2002.1028914.

[9] 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.

[10] Brice Nédelec et al. « An adaptive peer-sampling protocol for building networks of browsers ». In : *World Wide Web* 21.3 (2018), p. 629–661.

[11] 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.

[12] 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.

[13] D. Abadi. « Consistency Tradeoffs in Modern Distributed Database System Design : CAP is Only Part of the Story ». In : *Computer* 45.2 (2012), p. 37–42. doi : 10.1109/MC.2012.33.

[14] Rachid Guerraoui, Matej Pavlovic et Dragos-Adrian Seredinschi. « Trade-offs in replicated systems ». In : *IEEE Data Engineering Bulletin* 39.ARTICLE (2016), p. 14–26.

[15] 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.

[16] Marc Shapiro et al. « Conflict-Free Replicated Data Types ». In : *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. SSS 2011. 2011, p. 386–400. doi : 10.1007/978-3-642-24550-3_29.

[17] 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.

[18] 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.

[19] B. A. Davey et H. A. Priestley. *Introduction to Lattices and Order*. 2e éd. Cambridge University Press, 2002. doi : 10.1017/CBO9780511809088.

[20] Sebastian Burckhardt et al. « Replicated Data Types : Specification, Verification, Optimality ». In : *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA : Association for Computing Machinery, 2014, p. 271–284. isbn : 9781450325448. doi : 10.1145/2535838.2535848. url : https://doi.org/10.1145/2535838.2535848.

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

[22] Paul R Johnson et Robert Thomas. *RFC0677 : Maintenance of duplicate databases*. RFC Editor, 1975.

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin et al. « Consistency, availability, and convergence ». In : ().

[30] 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.

[31] D. S. Parker et al. « Detection of Mutual Inconsistency in Distributed Systems ». In : *IEEE Trans. Softw. Eng.* 9.3 (mai 1983), p. 240–247. ISSN : 0098-5589. DOI : 10.1109/TSE.1983.236733. URL : https://doi.org/10.1109/TSE.1983.236733.

[32] Giuseppe DeCandia et al. « Dynamo : Amazon's highly available key-value store ». In : *ACM SIGOPS operating systems review* 41.6 (2007), p. 205–220.

[33] 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.

[34] Ricardo Jorge Tomé Gonçalves et al. « 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.

[35] 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.

[36] V. Enes et al. « 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.

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

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

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

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

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

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

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

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

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

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

[47] Concordant. *Concordant*. URL : http://www.concordant.io/.

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

[49] Martin Kleppmann et al. « Local-First Software : You Own Your Data, in Spite of the Cloud ». In : *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward ! 2019. Athens, Greece : Association for Computing Machinery, 2019, p. 154–178. ISBN : 9781450369954. DOI : 10.1145/3359591.3359737. URL : https://doi.org/10.1145/3359591.3359737.

[50] 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.

[51] 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.

[52] C. A. Ellis et S. 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.

[53] 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.

[54] 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.

[55] Chengzheng Sun et al. « A consistency model and supporting schemes for real-time cooperative editing systems ». In : *Australian Computer Science Communications* 18 (1996), p. 582–591.

[56] 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.

[57] Gérald Oster et al. « 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.

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

[59] 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.

[60] Bernadette Charron-Bost. « Concerning the size of logical clocks in distributed systems ». In : *Information Processing Letters* 39.1 (1991), p. 11–16.

[61] Gérald Oster et al. « Data Consistency for P2P Collaborative Editing ». In : *ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*. Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work. Banff, Alberta, Canada : ACM Press, nov. 2006, p. 259–268. URL : https://hal.inria.fr/inria-00108523.

[62] Hyun-Gul Roh et al. « Replicated abstract data types : Building blocks for collaborative applications ». In : *Journal of Parallel and Distributed Computing* 71.3 (2011), p. 354–368. ISSN : 0743-7315. DOI : https://doi.org/10.1016/j.jpdc.2010.12.006. URL : http://www.sciencedirect.com/science/article/pii/S0743731510002716.

[63] Nuno Preguica et al. « A Commutative Replicated Data Type for Cooperative Editing ». In : *2009 29th IEEE International Conference on Distributed Computing Systems*. Juin 2009, p. 395–403. DOI : 10.1109/ICDCS.2009.20.

[64] Marc Shapiro et Nuno Preguiça. *Designing a commutative replicated data type*. Research Report RR-6320. INRIA, 2007. URL : https://hal.inria.fr/inria-00177693.

[65] Paulo Sérgio Almeida et al. « 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.

[66] Charbel Rahhal et al. *Undo in Peer-to-peer Semantic Wikis*. Research Report RR-6870. INRIA, 2009, p. 18. URL : https://hal.inria.fr/inria-00366317.

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

[68] 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.

[69] 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.

[70] 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.

[71] Hagit Attiya et al. « 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.

[72] Hagit Attiya et al. « 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.

[73] 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.

[74] 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.

[75] Martin Kleppmann et al. « 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.

[76] Matthew Weidner. *There Are No Doubly Non-Interleaving List CRDTs*. URL : https://mattweidner.com/assets/pdf/List_CRDT_Non_Interleaving.pdf.

[77] 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.

[78] 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.

[79] Elena Yanakieva et al. « Access Control Conflict Resolution in Distributed File Systems Using CRDTs ». In : *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '21. Online, United Kingdom : Association for Computing Machinery, 2021. isbn : 9781450383387. doi : 10.1145/3447865.3457970. url : https://doi.org/10.1145/3447865.3457970.

[80] Pierre-Antoine Rault, Claudia-Lavinia Ignat et Olivier Perrin. « Distributed Access Control for Collaborative Applications Using CRDTs ». In : *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '22. Rennes, France : Association for Computing Machinery, 2022, p. 33–38. isbn : 9781450392563. doi : 10.1145/3517209.3524826. url : https://doi.org/10.1145/3517209.3524826.

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