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 16 Décembre 2022

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Matthieu Nicolas

Composition du jury

Président :	À déterminer	
Rapporteurs :	Hanifa Boucheneb	Professeure, Polytechnique Montréal
	Davide Frey	Chargé de recherche, HdR, Inria Rennes Bretagne-Atlantique
Examinateurs :	Hala Skaf-Molli	Maîtresse de conférences, HdR, Nantes Université, LS2N
	Stephan Merz	Directeur de Recherche, Inria Nancy - Grand Est
Encadrants :	Olivier Perrin	Professeur des Universités, Université de Lorraine, LORIA
	Gérald Oster	Maître de conférences, Université de Lorraine, LORIA

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

Mis en page avec la classe thesul.

Remerciements

WIP

WIP

Sommaire

Table des figures

Chapitre 1

État de l'art

Sommaire

Dans ce chapitre, nous définissons le modèle du système que nous considérons (section 1.1). Puis nous présentons le fonctionnement de LogootSplit, le Conflict-free Replicated Data Type (CRDT) pour le type Séquence qui sert de base pour nos travaux (section 1.4). Ensuite, nous présentons les approches proposées pour réduire le surcoût des CRDTs pour le type Séquence et identifions leurs limites (sous-section 1.5.2 et sous-section 1.5.3). Finalement, nous introduisons l'approche que nous proposons (section 1.7) pour répondre à notre première problématique de recherche (cf. **??**, page **??**), que nous présentons en détails par la suite dans le chapitre 2.

Néanmoins, afin d'offrir une vision plus globale de notre domaine de recherche, nous complétons notre état de l'art de plusieurs points. Dans la section 1.2, nous rappelons la notion de CRDTs, c.-à-d. de types de données répliquées sans conflits. Ce rappel se compose d'une section présentant la notion de sémantique pour un mécanisme de résolution de conflits automatiques (sous-section 1.2.1) et d'une section présentant les différents modèles de synchronisation pour CRDTs définis dans la littérature, c.-à-d. la synchronisation par états, la synchronisation par opérations et la synchronisation par différences d'états (sous-section 1.2.2). À notre connaissance, nous présentons une des études les plus complètes comparant ces modèles de synchronisation en guise de synthèse de cette même section.

De manière similaire, nous rappelons les différents CRDTs pour le type Séquence définis dans la littérature dans la section 1.3. Ce rappel prend la forme d'un historique des CRDTs pour le type Séquence, catégorisés en fonction de l'approche sur laquelle se base leur mécanisme de résolution de conflits, c.-à-d. l'approche à pierres tombales ou l'approche à identifiants densément ordonnés. De nouveau, ce rappel aboutit à notre connaissance à l'une des études les plus précises comparant ces deux approches (sous-section 1.3.3).

1.1 Modèle du système

Le système que nous considérons est un système pair-à-pair (P2P) à large échelle. Il est composé d'un ensemble de noeuds dynamique. En d'autres termes, un noeud peut rejoindre ou quitter le système à tout moment. Certains noeuds peuvent participer au système que de manière éphémère, e.g. le temps d'une session.

Du point de vue d'un noeud du système, les autres noeuds sont soit connectés, c.-à-d. joignables par le biais des connexions P2P disponibles, soit déconnectés, c.-à-d. injoignable. Lorsqu'un noeud se déconnecte, nous considérons possible qu'il se déconnecte de manière définitive sans indication au préalable. Du point de vue des autres noeuds du système, il est donc impossible de déterminer le statut d'un noeud déconnecté. Ce dernier peut être déconnecté de manière temporaire ou définitive. Toutefois, nous assimilons les noeuds déconnectés de manière définitive à des noeuds ayant quittés le système, ceux-ci ne participant plus au système.

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

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

Définition 1 (Convergence à terme)**.** La convergence à terme est une propriété de sûreté indiquant que l'ensemble des noeuds du système ayant intégrés le même ensemble de modifications obtiendront des états équivalents [1].

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

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

Définition 2 (Livraison à terme)**.** La livraison à terme est un modèle de livraison garantissant que l'ensemble des messages du système seront livrés à l'ensemble des noeuds du système à terme.

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

1.2 Types de données répliquées sans conflits

Afin d'offrir une haute disponibilité à leurs clients et afin d'accroître leur tolérance aux pannes [1], les systèmes distribués peuvent adopter le paradigme de la réplication optimiste [2]. Ce paradigme consiste à ce que chaque noeud composant le système possède une copie de la donnée répliquée. Chaque noeud possède le droit de la consulter et de la modifier, sans coordination préalable avec les autres noeuds. Les noeuds peuvent alors temporairement diverger, c.-à-d. posséder des états différents. Un mécanisme de synchronisation leur permet ensuite de partager leurs modifications respectives et d'obtenir de nouveau des états équivalent, c.-à-d. de converger à terme [3].

Pour permettre aux noeuds de converger, les protocoles de réplication optimiste ordonnent généralement les évènements se produisant dans le système distribué. Pour les

1. 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.

ordonner, la littérature repose généralement sur la relation de causalité entre les évènements, qui est définie par la relation *happens-before* [4]. Nous l'adaptons ci-dessous à notre contexte, en ne considérant que les modifications [2] effectuées et celles intégrées :

Définition 3 (Relation *happens-before*)**.** La relation *happens-before* indique qu'une modification m_1 a eu lieu avant une modification m_2, notée $m_1 \rightarrow m_2$, si et seulement si une des conditions suivantes est satisfaite :

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

(ii) m_1 a été intégrée par le noeud auteur [3] de m_2 avant qu'il n'effectue m_2.

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

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

Définition 4 (Concurrence)**.** Deux modifications m_1 et m_2 sont concurrentes, noté $m_1 \parallel m_2$, si et seulement si $m_1 \nrightarrow m_2 \wedge m_2 \nrightarrow m_1$.

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

Avant d'illustrer notre propos avec un exemple, nous introduisons la spécification algébrique du type Ensemble dans la Figure 1.1 sur laquelle nous nous basons.

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

La spécification définit deux modifications sur l'ensemble :

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

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

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

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

2. 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.

3. Nous dénotons par le terme *auteur* le noeud à l'origine d'une modification.

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

constructor
empty : $\longrightarrow S$

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

queries
length : $S \longrightarrow \mathbb{N}$
read : $S \longrightarrow S$

FIGURE 1.1 – Spécification algébrique du type abstrait usuel Ensemble

Elle définit aussi deux observateurs :

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

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

Dans le cadre de ce manuscrit, nous travaillons sur des ensembles de caractères. Cette restriction du domaine se fait sans perte en généralité. En se basant sur cette spécification, nous présentons dans la Figure 1.2 un scénario où des noeuds effectuent en concurrence des modifications provoquant un conflit.

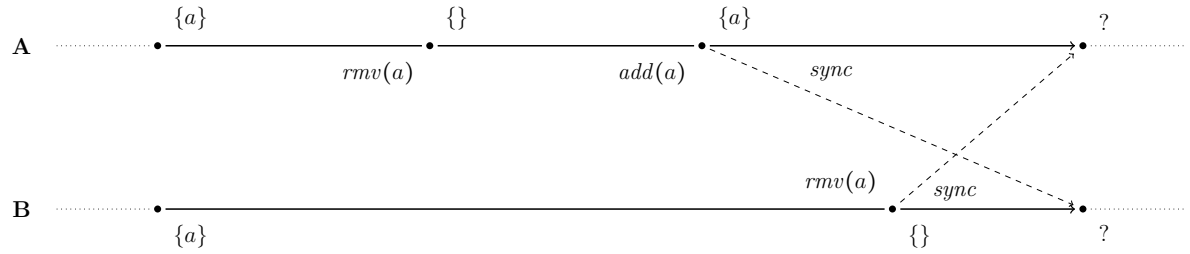


FIGURE 1.2 – Modifications concurrentes d'un Ensemble répliqué provoquant un conflit suite à l'ajout et la suppression d'un même élément

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

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

Les Conflict-free Replicated Data Types (CRDTs) [5, 6, 7] répondent à ce besoin.

Définition 5 (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* [5].

Définition 6 (Convergence forte)**.** La convergence forte est une propriété de sûreté indiquant que l'ensemble des noeuds d'un système ayant intégrés le même ensemble de modifications obtiendront des états équivalents, sans échange de message supplémentaire.

Pour offrir la propriété de *convergence forte*, la spécification des CRDTs reposent sur la théorie des treillis [8] :

Définition 7 (Spécification des CRDTs)**.** Les CRDTs sont spécifiés de la manière suivante :

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

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

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

Malgré leur spécification différente, les CRDTs partagent la même sémantique, c.-à-d. le même comportement, et la même interface que les types séquentiels [4] correspondants du point de vue des utilisateur-rices. Ainsi, les CRDTs partagent le comportement des types séquentiels dans le cadre d'exécutions séquentielles. Cependant, ils définissent aussi une sémantique additionnelle pour chaque type de conflit ne pouvant se produire que dans le cadre d'une exécution distribuée.

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

L'autre aspect définissant un CRDT donné est le modèle qu'il adopte pour propager les modifications. Au fil des années, la littérature a établi et défini plusieurs modèles dit de

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

synchronisation, chacun ayant ses propres besoins et avantages. De fait, plusieurs CRDTs peuvent être proposés pour un même type donné en fonction du modèle de synchronisation choisi.

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

1.2.1 Sémantiques en cas de conflits

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

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

Dans la suite de cette section, nous présentons ces sémantiques génériques ainsi que celles spécifiques à l'Ensemble et, à titre d'exemple, les illustrons à l'aide du scénario présenté dans la Figure 1.2.

Sémantique *Last-Writer-Wins*

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

Cette approche, présentée dans [9], 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 1.3 illustre son application à l'Ensemble pour résoudre le conflit de la Figure 1.2.

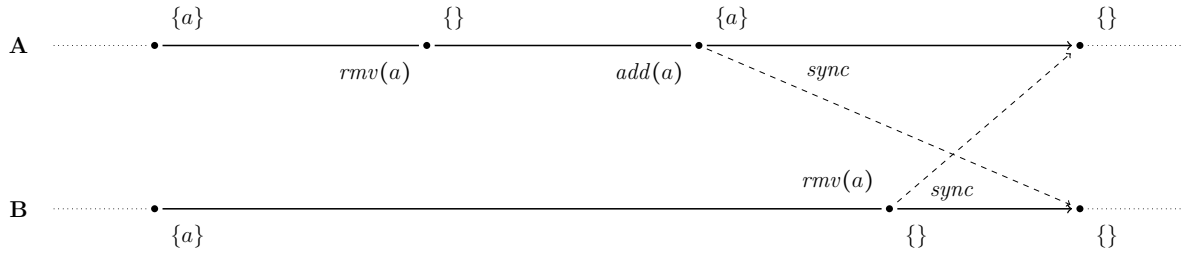


FIGURE 1.3 – Résolution du conflit en utilisant la sémantique LWW

Comme indiqué précédemment, le scénario illustré dans la Figure 1.3 présente un conflit entre les modifications concurrentes $add(a)$ et $remove(a)$ 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 $remove(a)$. La sémantique LWW désigne donc $remove(a)$ 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 $remove(a)$ du noeud B avait eu lieu plus tôt que la modification $add(a)$ du noeud A 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 [4].

Sémantique *Multi-Value*

Une seconde sémantique générique[5] 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 1.4.

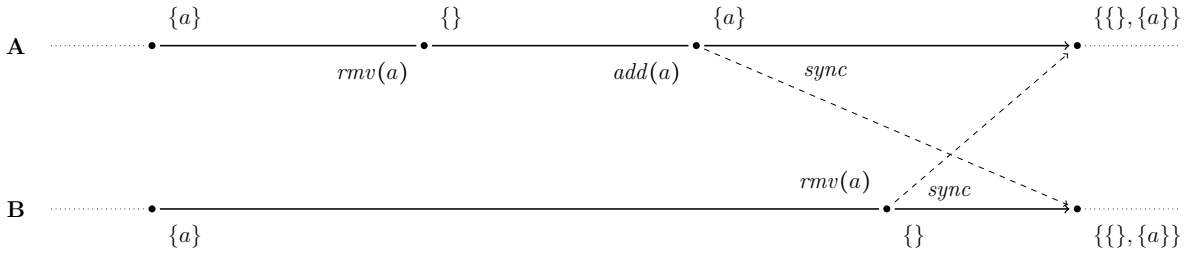


FIGURE 1.4 – Résolution du conflit en utilisant la sémantique MV

La Figure 1.4 présente la gestion du conflit entre les modifications concurrentes $add(a)$ et $remove(a)$ 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 $\{\{\}, \{a\}\}$.

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

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 1.4, résoudre le conflit revient à re-effectuer une modification $add(a)$ ou $remove(a)$ 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. Par exemple, si nous considérons que le type correspondant au CRDT dans la Figure 1.4 est le type $Set\langle V\rangle$, nous observons que la valeur finale obtenue a pour type $Set\langle Set\langle V\rangle\rangle$. Il s'agit à notre connaissance de la seule sémantique opérant ce changement.

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

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

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

(i) Une sémantique où la modification *add* d'une élément prend la précédence sur les modifications concurrentes *remove* 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 *remove* 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 1.5 illustre l'application de chacune de ces sémantiques sur notre exemple.

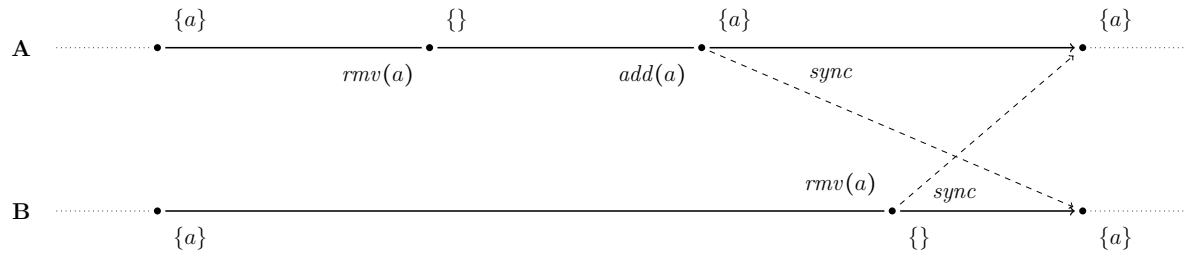
Sémantique *Causal-Length*

Une nouvelle sémantique pour l'Ensemble fut proposée [10] récemment. Cette sémantique se base sur les observations suivantes :

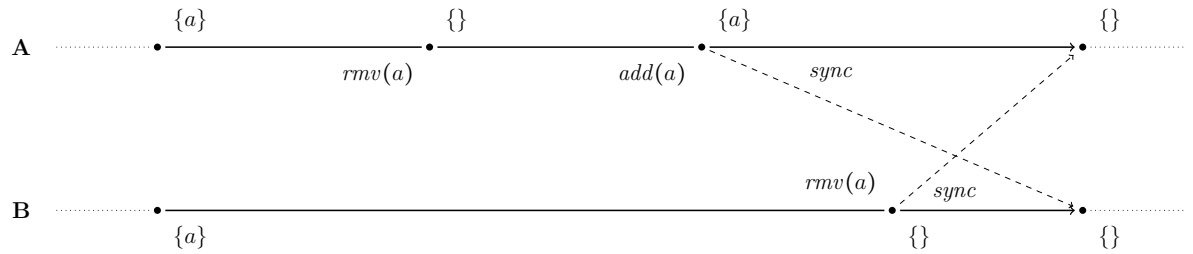
(i) *add* et *remove* d'un élément prennent place à tour de rôle, chaque modification invalidant la précédente.

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

À partir de ces observations, Yu et al. [10] 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'élément est présent ou non dans l'ensemble final. La Figure 1.6 illustre son fonctionnement.



(a) Application de la sémantique AW



(b) Application de la sémantique RW

FIGURE 1.5 – Résolution du conflit en utilisant soit la sémantique AW, soit la sémantique RW

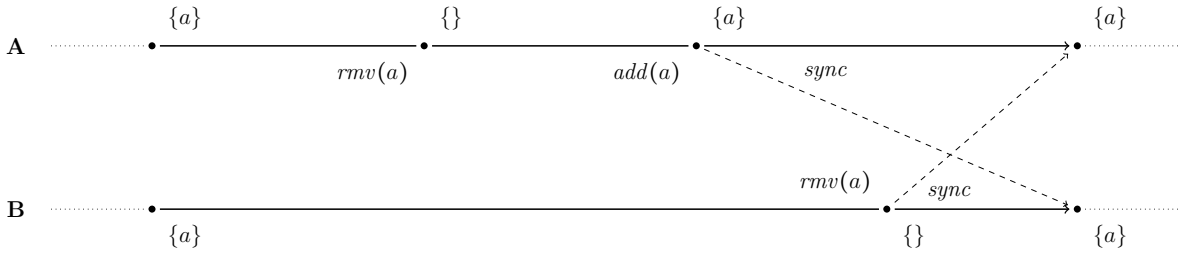


FIGURE 1.6 – Résolution du conflit en utilisant la sémantique CL

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

À l'inverse, la modification $add(a)$ effectuée par A fait suite à sa modification $remove(a)$. La sémantique CL définit alors qu'elle fait suite à toute autre modification $remove(a)$ 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 1.7 illustre ce point.

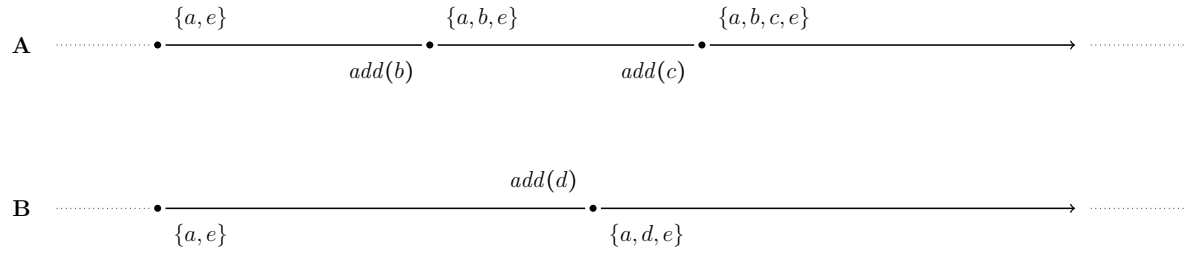


FIGURE 1.7 – Modifications en concurrence d'un Ensemble répliqué par les noeuds A et B

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

Le noeud A effectue les modifications $add(b)$ puis $add(c)$. Il obtient ainsi l'état $\{a, b, c, e\}$. De son côté, le noeud B effectue la modification suivante : $add(d)$. Son état devient donc $\{a, d, e\}$. Ainsi, les noeuds doivent encore s'échanger leur modifications pour converger vers l'état souhaité [6], 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 [5, 11] et une méthode de synchronisation par opérations [5, 11, 12, 13]. Une troisième

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

approche, nommée synchronisation par différence d'états [14, 15], 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 dans nos représentations uniquement sur les messages envoyés par les noeuds. Les métadonnées introduites par chaque modèle de synchronisation sont uniquement évoquées à l'écrit, par souci de clarté et de simplicité de nos exemples.

Synchronisation par états

L'approche de la synchronisation par états propose que les noeuds diffusent leurs modifications en transmettant leur état. Les CRDTs adoptant cette approche doivent définir une fonction `merge`. Cette fonction correspond à la fonction de fusion mentionnée précédemment (cf. Définition 7, page 6) : elle prend en paramètres une paire d'états et génère en retour leur LUB, c.-à-d. l'état correspondant à la borne supérieure des deux états en paramètres. Cette fonction doit être associative, commutative et idempotente [5].

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

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

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

Définition 8 (Pierre tombale)**.** Une pierre tombale est un marqueur de la présence passée d'un élément.

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

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

synchronisation périodique des noeuds, c.-à-d. chaque noeud diffuse périodiquement son état.

Nous illustrons le fonctionnement de cette approche avec la Figure 1.8. Dans cet exemple, après que les noeuds aient effectués leurs modifications respectives, le mécanisme de synchronisation périodique de chaque noeud se déclenche. Le noeud A (resp. B) diffuse alors son état $\{a, b, c, e\}$ (resp. $\{a, d, e\}$) à B (resp. A).

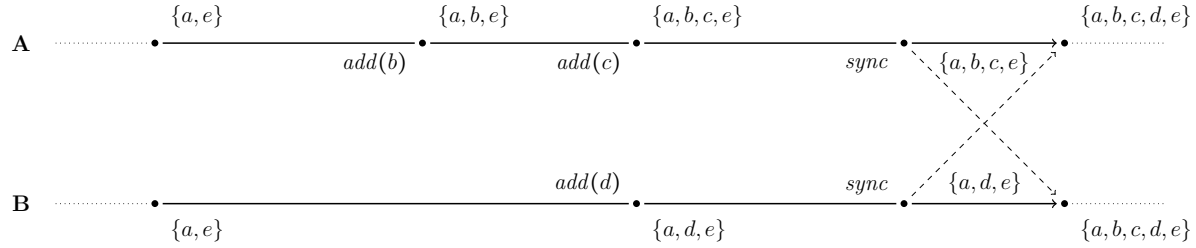


FIGURE 1.8 – Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par états

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 les CRDTs adoptant ce modèle de synchronisation respectent de manière intrinsèque le modèle de cohérence causale [16].

Définition 9 (Modèle de cohérence causale)**.** Le modèle de cohérence causale définit que, pour toute paire de modifications m_1 et m_2 d'une exécution, si $m_1 \rightarrow m_2$, alors l'ensemble des noeuds doit intégrer la modification m_1 avant d'intégrer la modification m_2.

En effet, ce modèle de synchronisation assure l'intégration soit de toutes les modifications connues d'un noeud, soit d'aucune. Par exemple, dans la Figure 1.8, le noeud B ne peut pas recevoir et intégrer l'élément c sans l'élément 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` [13].

La fonction `prepare` a pour but de générer une opération correspondant à la modification effectuée, et commutative avec les potentielles opérations concurrentes. Cette fonction 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 10 (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 [13], les modifications estampillées avec leur information de causalité correspondent à des éléments irréductibles, c.-à-d. à des opérations. La fonction `prepare` peut donc être omise pour cette sous-catégorie de CRDTs synchronisés par opérations.

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

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

Il est à noter que la seule contrainte imposée aux CRDTs synchronisés par opérations est que leurs opérations concurrentes soient commutatives [5]. Ainsi, il n'existe aucune contrainte sur la commutativité des opérations liées causalement. De la même manière, aucune contrainte n'est définie sur l'idempotence des opérations. Ces libertés impliquent qu'il peut être nécessaire que les opérations soient livrées au CRDT en respectant un ordre donné et en garantissant leur livraison en exactement une fois pour garantir la convergence [7]. Ainsi, un intergiciel chargé de la diffusion et de la livraison des opérations est usuellement associé aux CRDTs synchronisés par opérations pour respecter ces contraintes. Il s'agit de la couche de livraison de messages que nous avons introduit dans le cadre de notre modèle du système (cf. section 1.1, page 2).

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

Définition 11 (Modèle de livraison causale)**.** Le modèle de livraison causale définit que, pour toute paire de messages m_1 et m_2 d'une exécution, si $m_1 \rightarrow m_2$, alors la couche de livraison de l'ensemble des noeuds doit livrer le message m_1 à l'application avant de livrer le message m_2.

Ce modèle de livraison permet de respecter le modèle de cohérence causale.

Ce modèle de livraison introduit néanmoins plusieurs effets négatifs. Tout d'abord, ce modèle peut provoquer un délai dans l'intégration des modifications. En effet, la perte

d'une opération par le réseau provoque la mise en attente de la livraison des opérations suivantes. Les opérations mises en attente ne pourront en effet être livrées qu'une fois l'opération perdue re-diffusée et livrée.

De plus, il nécessite que des informations de causalité précises soient attachées à chaque opération. Pour cela, les systèmes reposent généralement sur l'utilisation de vecteurs de versions [17, 18]. 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 [7] [20]. Cependant, nous observons que la livraison dans l'ordre causal de toutes les opérations n'est pas toujours nécessaire pour la convergence. Par exemple, l'ordre d'intégration de deux opérations d'ajout d'éléments différents dans un Ensemble n'a aucun impact sur le résultat obtenu. Nous pouvons alors nous affranchir du modèle de livraison causale pour accélérer la vitesse d'intégration des modifications et pour réduire les métadonnées envoyées.

Pour compenser la perte d'opérations par le réseau et ainsi garantir la livraison à terme des opérations, la couche de livraison des opérations doit mettre en place un mécanisme d'anti-entropie, c.-à-d. un mécanisme permettant de détecter et ré-échanger les messages perdus. Plusieurs mécanismes de ce type ont été proposés dans la littérature [21, 22, 23, 24] 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 1.9. Dans ce nouvel exemple, les noeuds diffusent les modifications qu'ils effectuent sous la forme d'opérations. Nous considèrons que le CRDT utilisé est un CRDT pur synchronisé par opérations, c.-à-d. que les modifications et opérations sont confondues, et qu'il autorise une livraison dans le désordre des opérations *add*.

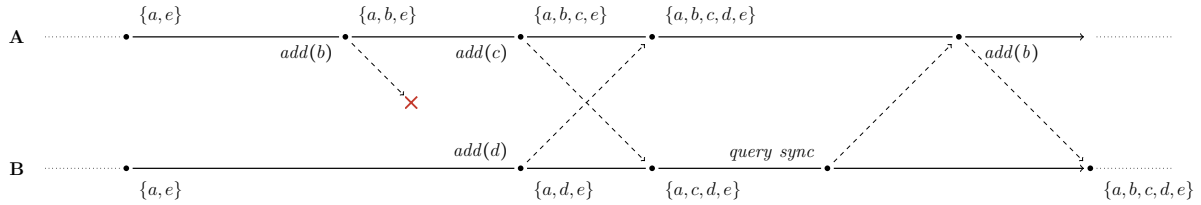


FIGURE 1.9 – Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par opérations

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

De son côté, le noeud B ne reçoit initialement pas l'opération $add(b)$ suite à une perte de message. 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 $\{a, c, d, e\}$.

7. La relation de causalité étant transitive, les opérations et leurs relations de causalité forment un DAG. [19] 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.

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 versions. 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 des 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 [13, 25] 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 [26] pour déterminer les opérations supprimables de manière sûre.

Définition 12 (Stabilité causale)**.** Une opération est stable causalement lorsqu'elle a été intégrée par l'ensemble des noeuds du système. Ainsi, toute opération future dépend causalement des opérations causalement stables, c.-à-d. les noeuds ne peuvent plus générer d'opérations concurrentes aux opérations causalement stables.

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

Pour résumer, cette approche permet de mettre en place un système en composant un CRDT synchronisé par opérations avec une couche de livraison des messages. Mais comme illustré ci-dessus, chaque CRDT synchronisé par opérations établit les propriétés de ses différentes opérations et délègue potentiellement des responsabilités à la couche de livraison. Une partie de la complexité de cette approche réside ainsi dans l'ajustement du couple $\langle CRDT, \textit{couche livraison} \rangle$ pour régler finement et optimiser leur fonctionnement en tandem. Des travaux [13, 25] ont proposé un patron de conception pour modéliser ces deux composants et leurs interactions. Cependant, ce patron repose sur l'hypothèse d'une livraison causale des opérations et n'est donc pas optimal.

Synchronisation par différences d'états

Almeida et al. [14] introduisent un nouveau modèle de synchronisation pour CRDTs. La proposition de ce modèle est nourrie par les observations suivantes :

(i) Les CRDTs synchronisés par opérations sont sujets aux défaillances du réseau et nécessitent généralement pour pallier ce problème une couche de livraison des messages garantissant la livraison en exactement un exemplaire des opérations et réordonnant les opérations pour satisfaire le modèle de livraison causal.

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

Pour pallier les faiblesses de chaque approche et allier le meilleur des deux mondes, les auteurs proposent les CRDTs synchronisés par différences d'états [14, 15, 20]. Il s'agit

en fait d'une sous-famille des CRDTs synchronisés par états. Ainsi, comme ces derniers, ils disposent d'une fonction `merge` associative, commutative et idempotente qui permet de produire la LUB de deux états, c.-à-d. l'état correspond à la borne supérieure de ces deux états.

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

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

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

Cette définition des CRDTs synchronisés par différences d'états, introduite dans [14, 15], fut ensuite précisée dans [20]. Dans cet article, les auteurs précisent qu'utiliser des éléments irréductibles (cf. Définition 10, page 14) comme deltas est optimal du point de vue de la taille des deltas produits.

Concernant la diffusion des modifications, les CRDTs synchronisés par différences d'états autorisent un large éventail de possibilités. Par exemple, les deltas peuvent être diffusés et intégrés de manière indépendante. Une autre approche possible consiste à tirer avantage du fait que les deltas sont des états : il est possible d'agréger plusieurs deltas à l'aide de la fonction `merge`, éliminant leurs éventuelles redondances. Ainsi, la fusion de deltas permet ensuite de diffuser un ensemble de modifications par le biais d'un seul et unique delta, minimal. Et en dernier recours, les CRDTs synchronisés par différences d'états peuvent adopter le même schéma de diffusion que les CRDTs synchronisés par états, c.-à-d. diffuser leur état complet de manière périodique. Chacune de ces approches propose un compromis entre délai d'intégration des modifications, surcoût en métadonnées, calculs et bande-passante [20]. Ainsi, il est possible pour un système utilisant des CRDTs synchronisés par différences d'états de sélectionner la technique de diffusion des modifications la plus adaptée à ses besoins, ou même d'alterner entre plusieurs en fonction de son état courant.

Nous illustrons cette approche avec la Figure 1.10. Dans cet exemple, nous considérons que les noeuds adoptent la seconde approche évoquée, c.-à-d. que périodiquement les noeuds aggrégent les deltas issus de leurs modifications et diffusent le delta résultant.

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

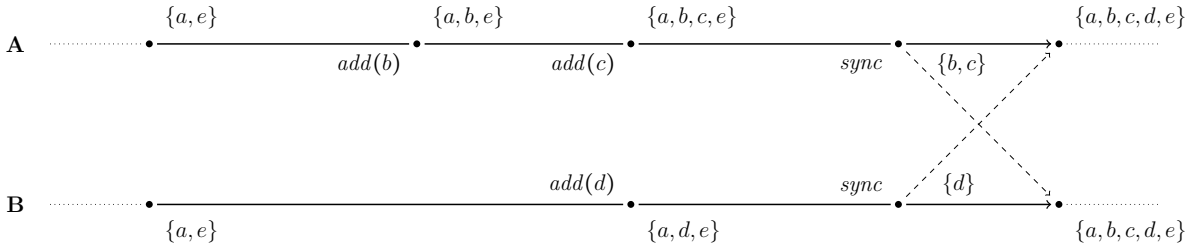


FIGURE 1.10 – Synchronisation des noeuds A et B en adoptant le modèle de synchronisation par différences d'états

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 intégration dans le désordre des modifications par l'un des noeuds. S'ils souhaitent toujours satisfaire le modèle de cohérence causal, les CRDTs synchronisés par différences d'états doivent donc définir et ajouter à leur spécification un mécanisme similaire à la couche de livraison des CRDTs synchronisés par opérations.

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

Synthèse

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

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

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

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

au second et en fusionnant cet état avec l'état du second.

Le modèle de synchronisation par opérations repose sur des opérations pour diffuser les modifications. Les opérations encodent les modifications sous la forme d'un ou plusieurs états spécifiques du sup-demi-trellis : les éléments irréductibles (cf. Définition 10, page 14). L'intégration des modifications d'un noeud par un second se fait alors en diffusant les opérations correspondant aux modifications et en intégrant chacune d'entre elle à la copie locale du second.

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

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

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

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

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

Finalement, la dernière différence entre ces modèles concerne le modèle de cohérence causale (cf. Définition 9, page 13). Par nature, le modèle de synchronisation par états garantit le respect du modèle de cohérence causale. En effet, un état correspond à l'intégration d'un ensemble de modifications. De manière similaire, le résultat de la fusion de deux états correspond à l'intégration de l'union de leur ensemble respectif de modifications. Ce modèle de synchronisation empêche donc l'intégration d'une modification sans avoir intégré aussi les modifications l'ayant précédé d'après la relation *happens-before*.

À l'inverse, par défaut, les modèles de synchronisation par opérations ou différences d'états permettent l'intégration d'un élément irréductible sans avoir intégré au préalable les éléments irréductibles l'ayant précédé d'après la relation *happens-before*. Pour satisfaire le modèle de cohérence causale, les CRDTs adoptant ces modèles de synchronisation

doivent être associés à une couche de livraison de messages garantissant leur livraison causale (cf. Définition 11, page 14).

Nous récapitulons le contenu de cette discussion sous la forme du Tableau 1.1.

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

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

1.3 Séquences répliquées sans conflits

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

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

Dans le cadre de ce manuscrit, nous représentons des séquences de caractères. Cette restriction du domaine se fait sans perte de généralité. Nous illustrons par la Figure 1.11 notre représentation des séquences que nous utiliserons dans nos exemples.

H	E	L	L	O
0	1	2	3	4

FIGURE 1.11 – Représentation de la séquence "HELLO"

Dans la Figure 1.12, nous présentons la spécification algébrique du type Séquence que nous utilisons.

Celle-ci définit deux modifications :

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

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

(ii) $remove(s, i)$, abrégée en *rmv* dans nos figures, qui permet de retirer l'élément situé à l'index i dans une séquence s de taille m. Cette modification renvoie une nouvelle

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

constructor

empty	:		$\longrightarrow S$

mutators

insert	:	$S \times \mathbb{N} \times E$	$\longrightarrow S$
remove	:	$S \times \mathbb{N}$	$\longrightarrow S$

queries

length	:	S	$\longrightarrow \mathbb{N}$
read	:	S	$\longrightarrow Array\langle E \rangle$

FIGURE 1.12 – Spécification algébrique du type abstrait usuel Séquence

séquence construite de la manière suivante :

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

Les modifications définies dans la Figure 1.12, *insert* et *remove*, ne permettent respectivement que l'insertion ou la suppression d'un élément à la fois. Cette simplification du type se fait cependant sans perte de généralité, la spécification pouvant être étendue pour insérer successivement plusieurs éléments à partir d'un index donné ou retirer plusieurs éléments consécutifs.

La spécification définit aussi deux observateurs :

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

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

Cette spécification du type Séquence est une spécification séquentielle. Les modifications sont définies pour être effectuées l'une après l'autre. Si plusieurs noeuds répliquent une même séquence et la modifient en concurrence, l'intégration de leurs opérations respectives dans des ordres différents résulte en des états différents. Nous illustrons ce point avec la Figure 1.13.

Dans cet exemple, deux noeuds A et B partagent et éditent collaborativement une même séquence. Celle-ci correspond initialement à la chaîne de caractères "WRD". Le noeud A insère le caractère "O" à l'index 1, obtenant ainsi la séquence "WORD". En concurrence, le noeud B insère lui le caractère "L" à l'index 2 pour obtenir "WRLD".

Les deux noeuds diffusent ensuite leur opération respective puis intègre celle de leur pair. Nous constatons alors une divergence. En effet, l'intégration de la modification

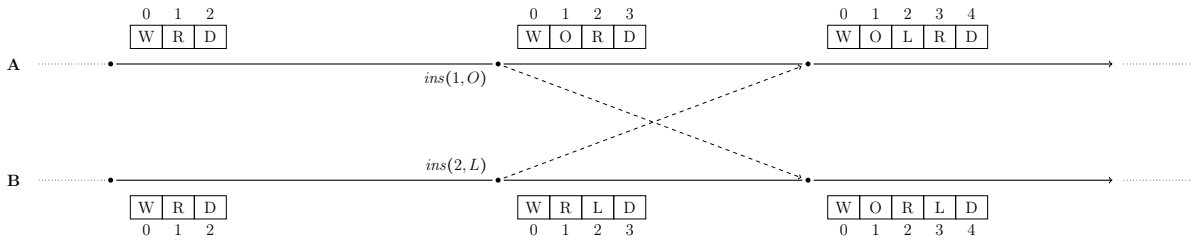


FIGURE 1.13 – Modifications concurrentes d'une séquence

$insert(2, L)$ 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 au fait que la modification *insert* ne commute pas 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 *insert*. Ainsi, l'intégration dans des ordres différents de modifications *insert* 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. $\langle insert, insert\rangle$, $\langle insert, remove\rangle$ et $\langle remove, remove\rangle$, ne commute pas.

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) [27, 28] 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 [29, 30, 31], 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 [27, 29, 32, 33], 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) [33] a été éprouvée pour les systèmes P2P à notre connaissance. De plus, les algorithmes de contrôle compatibles reposent sur une livraison causale des modifications, et donc l'utilisation de vecteurs d'horloges. Cette approche est donc inadaptée aux systèmes P2P dynamiques.

Néanmoins, une contribution importante de l'approche OT fut la définition d'un modèle de cohérence que doivent respecter les systèmes d'édition collaboratif : le modèle Convergence, Causality preservation, Intention preservation (CCI) [34].

Définition 13 (Modèle Convergence, Causality preservation, Intention preservation)**.** Le modèle de cohérence Convergence, Causality preservation, Intention preservation définit qu'un système d'édition collaboratif doit respecter les critères suivants :

Définition 13.1 (Convergence)**.** Le critère de *Convergence* indique que des noeuds ayant intégrés le même ensemble de modifications convergent à un état équivalent.

Définition 13.2 (Préservation de la causalité)**.** Le critère de *Préservation de la causalité* indique que si une modification m_1 précède une autre modification m_2 d'après la relation *happens-before*, c.-à-d. $m_1 \rightarrow m_2$, m_1 doit être intégrée avant m_2 par les noeuds du système.

Définition 13.3 (Préservation de l'intention)**.** Le critère de *Préservation de l'intention* indique que l'intégration d'une modification par un noeud distant doit reproduire l'effet de la modification sur la copie du noeud d'origine, indépendamment des modifications concurrentes intégrées.

De manière similaire à [35], nous considérons qu'un système collaboratif doit, en plus du modèle CCI, assurer sa *capacité de passage à l'échelle* (cf. **??**, page **??**). Nous précisons notre définition de cette propriété ci-dessous :

Définition 14 (Capacité de passage à l'échelle)**.** Le capacité d'un passage à l'échelle d'un système indique que son nombre de noeuds n'a qu'un impact limité, c.-à-d. idéalement constant ou logarithmique, sur sa complexité en temps, en espace et sur le nombre et la taille des messages.

Nous constatons cependant que le critère 13.2 et la propriété 14 peuvent être contradictoires. 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 [36]. Ainsi, dans le cadre de nos travaux sur la conception de systèmes collaboratifs P2P à large échelle, nous cherchons à nous affranchir du modèle de livraison causale des modifications, ce qui peut nécessiter de relaxer le modèle de cohérence causale.

C'est dans une optique similaire que fut proposé WOOT [37], un modèle de séquence répliquée qui pose les fondations des CRDTs. Depuis, plusieurs CRDTs pour le type Séquence furent définies [38, 39, 35]. Ces CRDTs peuvent être répartis en deux approches : l'approche à pierres tombales [37, 38] et l'approche à identifiants densément ordonnés [39, 35]. 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 [40] propose un CRDT pour le type Séquence synchronisé par différence d'états.

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

1.3.1 Approche à pierres tombales

WOOT

WOOT [37] est considéré a posteriori comme le premier CRDT synchronisé par opérations pour le type Séquence [8]. 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.

8. [41] n'ayant formalisé les CRDTs qu'en 2007.

L'intuition de WOOT est la suivante : WOOT modifie la sémantique de la modification *insert* 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 "K" dans la séquence "SY" pour obtenir l'état "SKY", c.-à-d. $insert(1, K)$, devient $insert(S \prec K \prec Y)$, où $\prec$ représente l'ordre créé entre ces éléments.

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

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

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

Dans le cadre de ce manuscrit, nous utiliserons pour former les identifiants WOOT le nom 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. $A1$ [9].

Les modifications *insert* et *remove* génèrent dès lors des opérations tirant profit des identifiants. Par exemple, considérons une séquence WOOT représentant "SY" et qui associe respectivement les identifiants $A1$ et $A2$ aux éléments "S" et "Y". L'insertion de l'élément "E" dans cette séquence pour obtenir l'état "SKY", c.-à-d. $insert(S \prec K \prec Y)$, produit par exemple l'opération $insert(A1 \prec \langle B1, K \rangle \prec A2)$. De manière similaire, la suppression de l'élément "K" dans cette séquence pour obtenir l'état "SA", c.-à-d. $remove(1)$, produit $remove(B1)$.

WOOT utilise des pierres tombales pour que les opérations *insert*, qui nécessite la présence des deux éléments entre lesquels nous insérons un nouvel élément, et *remove* commutent. 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.

Finalement, WOOT définit $<_{id}$, un ordre strict total sur les identifiants associés aux éléments. En effet, la relation $\prec$ n'est pas définie pour deux éléments insérés en concurrence et qui possèdent les mêmes prédecesseur et successeur, e.g. $insert(S \prec K \prec Y)$ et $insert(S \prec L \prec Y)$. 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 16 (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 :

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

9. 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*.

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 commutent. Nous récapitulons son fonctionnement à l'aide de la Figure 1.14.

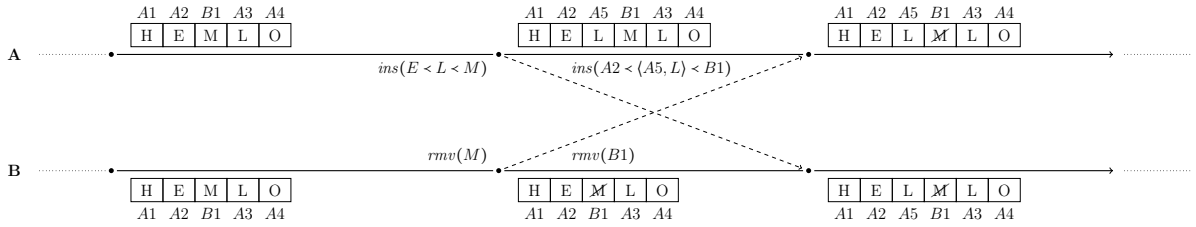


FIGURE 1.14 – Modifications concurrentes d'une séquence répliquée WOOT

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

Le noeud A insère l'élément "L" entre les éléments "E" et "M", c.-à-d. $insert(E \prec L \prec M)$. WOOT convertit cette modification en opération $insert(A2 \prec \langle A5, L\rangle \prec B1)$. 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. $remove(M)$. De la même manière, WOOT génère l'opération correspondante $remove(B1)$. 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, $remove(B1)$ (resp. A, $insert(A2 \prec \langle A5, L\rangle \prec B1)$), 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 17 (Modèle de livraison WOOT)**.** Le modèle de livraison WOOT définit que :

(i) Une opération doit être livrée exactement une fois à chaque noeud [10].

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

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

10. Néanmoins, les algorithmes d'intégration des opérations, notamment celui pour l'opération *insert*, 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*.

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 *remove* 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 qu'en temps.

Oster et al. [37] 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'annulation, une fonctionnalité importante dans le domaine de l'édition collaborative. Cette piste de recherche est développée dans [42].

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 [43]. Plusieurs évolutions de WOOT sont proposées pour mitiger cette limite : WOOTO [44] et WOOTH [43].

Weiss et al. [44] remanient la structure des identifiants associés aux éléments. Cette modification permet un algorithme d'intégration des opérations *insert* avec une meilleure complexité en temps, $\mathcal{O}(H^2)$. Ahmed-Nacer et al. [43] 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 *insert*.

Néanmoins, l'évaluation expérimentale des différentes approches pour l'édition collaborative P2P en temps réel menée dans [43] 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 [45, 46]. 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) [38] est le second CRDT pour le type 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 des RADTs. Le premier d'entre eux est la Commutativité des Opérations (OC).

Définition 18 (Commutativité des Opérations)**.** La Commutativité des Opérations (OC) définit que toute paire possible d'opérations concurrentes du RADT doit être commutative.

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

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

Définition 19 (Transitivité de la Précédence)**.** La Transitivité de la Précédence (PT) définit qu'étant donné une relation de précédence, $\dashrightarrow$, et trois opérations, o_1, o_2 et o_3, si $o_1 \dashrightarrow o_2$ et $o_2 \dashrightarrow o_3$, alors nous avons $o_1 \dashrightarrow o_3$.

avec la relation de précédence $\dashrightarrow$ définie de la manière suivante :

Définition 20 (Relation de précédence)**.** La relation de précédence, notée $\dashrightarrow$, 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 prend la précédence sur o_1.

Ce second principe offre une méthode pour concevoir un ensemble d'opérations commutatives. Il permet aussi d'exprimer la précédence 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 21 (Identifiant S4Vector)**.** Un identifiant S4Vector est de la forme $\langle ssid, sum, ssn, seq \rangle$ avec :

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

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

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

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

Cependant, dans les présentations suivantes de RGA [5, 47], les auteurs utilisent des horloges de Lamport [4] 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) *insert* devient $insert(predId \prec \langle t, elt \rangle)$.

(ii) *remove* devient $remove(t)$.

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 22 (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 *insert* 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 *insert* et *remove*.

Nous récapitulons le fonctionnement de RGA à l'aide de la Figure 1.15.

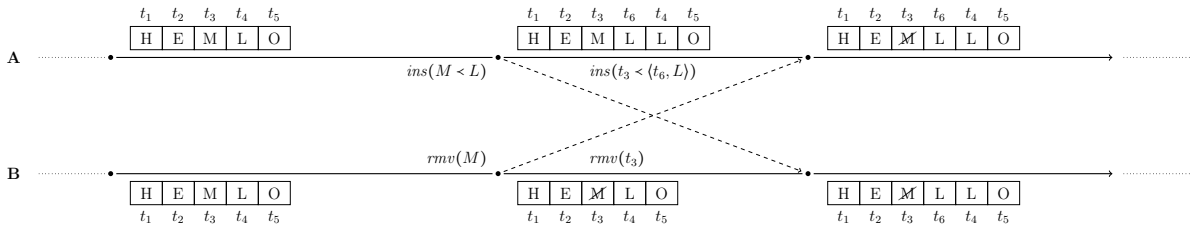


FIGURE 1.15 – Modifications concurrentes d'une séquence répliquée RGA

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

Le noeud A insère l'élément "L" après l'élément et "M", c.-à-d. $insert(M \prec L)$. RGA convertit cette modification en opération $insert(t_3 \prec \langle t_6, L \rangle)$. 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. $remove(M)$. De la même manière, RGA génère l'opération correspondante $remove(t_3)$. 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~~M~~LO". L'opération est ensuite diffusée.

A (resp. B) reçoit ensuite l'opération de B, $remove(t_3)$ (resp. A, $insert(t_3 \prec \langle t_6, L \rangle)$), 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. [38] 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 *remove*, c.-à-d. l'ensemble des noeuds a intégré la suppression de l'élément et ne peut émettre d'opérations utilisant l'élément supprimé comme prédecesseur.

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

L'intuition de la condition (i) est de s'assurer qu'aucune opération *insert* concurrente à l'exécution du mécanisme ne peut utiliser la pierre tombale comme prédecesseur, les opérations *insert* ne pouvant reposer que sur les éléments. L'intuition de la condition (ii) est de s'assurer que l'intégration d'une opération *insert*, 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, [38] 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 [40] indique que RGA pourrait adopter un modèle de livraison similaire à celui de WOOT. Ce modèle consisterait :

Définition 23 (Modèle de livraison RGA)**.** Le modèle de livraison RGA définit que :

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

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

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

Nous secondons cette observation.

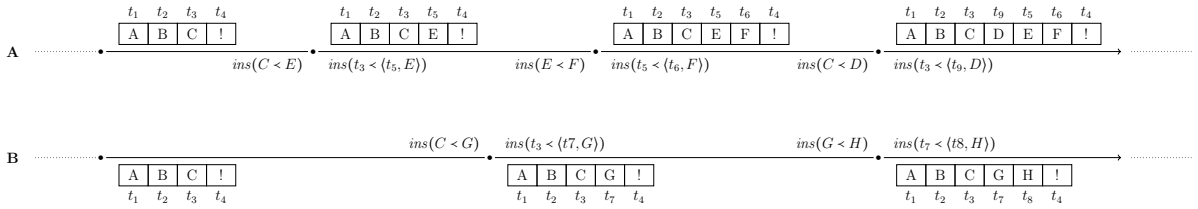
Un des avantages de RGA est son efficacité. En effet, son algorithme d'intégration des insertions offre une meilleure complexité en temps que celui de WOOT : $\mathcal{O}(H)$, avec H le nombre de modifications ayant été effectuées sur le document [43]. De plus, [47, 48] montrent que le modèle de données de RGA est optimal d'un point de vue complexité en espace comme CRDT pour le type Séquence par élément sans mécanisme de GC.

Plusieurs extensions de RGA ont par la suite été proposées. Briot et al. [49] indiquent que les pauvres performances des modifications locales [11] des CRDTs pour le type Séquence constituent une de leurs limites. Il s'agit en effet des performances impactant le plus l'expérience utilisateur, les utilisateur-rices 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.

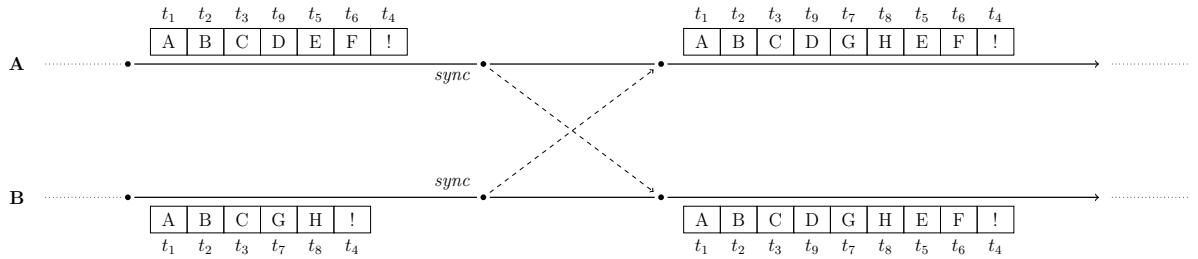
11. Relativement par rapport aux algorithmes de l'approche OT.

Pour cela, ils proposent l'*identifier structure*, une structure de données auxiliaire utilisable par les CRDTs pour le type 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 éléments en blocs [12] tels que proposés par [50, 51], qui permet de réduire la quantité de métadonnées stockées par la séquence répliquée. Cette combinaison aboutit à la définition d'un nouveau CRDT pour le type Séquence, *RGATreeSplit*, qui offre une meilleure complexité en temps et en espace.

Dans [52], les auteurs mettent en lumière un problème récurrent des CRDTs pour le type 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 1.16 présente un tel cas de figure :



(a) Modifications concurrentes d'une séquence répliquée RGA



(b) Synchronisation résultant en un entrelacement des modifications concurrentes

FIGURE 1.16 – Entrelacement d'éléments insérés de manière concurrente

Dans la Figure 1.16a, deux noeuds A et B partagent et éditent collaborativement une séquence répliquée RGA. Initialement, ils possèdent le même état : la séquence contient les éléments "ABC!", et à chaque élément est associé un identifiant, e.g. t_1, t_2, t_3 et t_4.

Le noeud A insère après l'élément "C" les éléments "E" et F. RGA génère les opérations $insert(t_3 \prec \langle t_5, E \rangle)$ et $insert(t_5 \prec \langle t_6, F \rangle)$. En concurrence, le noeud B insère les éléments "G" et "H" de manière similaire, produisant les opérations $insert(t_3 \prec \langle t_7, G \rangle)$ et $insert(t_7 \prec \langle t_8, H \rangle)$. Finalement, toujours en concurrence, le noeud A insère un nouvel élément après l'élément "C", l'élément "D", ce qui résulte en l'opération $insert(t_9 \prec \langle t_3, D \rangle)$. 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 1.16b. Dans cette figure, les noeuds A et B se synchronisent et échangent leurs opérations respectives. À la réception de l'opération de B $insert(t_3 \prec \langle t_7, G \rangle)$, le noeud A compare t_7 avec les identifiants des

12. Nous détaillerons ce mécanisme par la suite.

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 [47, 48], cette nouvelle spécification précise que les éléments insérés en concurrence ne doivent pas s'entrelacer dans l'état final. Kleppmann et al. [52] proposent ensuite une évolution de RGA respectant cette spécification.

Pour cela, les auteurs ajoutent à l'opération *insert* 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 *insert* sont concurrentes ou causalement liées et ainsi déterminer comment ordonner leurs éléments. Cependant, les auteurs ne prouvent pas dans [52] que cette extension empêche tout entrelacement [13].

1.3.2 Approche à identifiants densément ordonnés

Treedoc

[41, 39] proposent une nouvelle approche pour CRDTs pour le type Séquence. La particularité de cette approche est de se baser sur des identifiants de position, respectant un ensemble de propriétés :

Définition 24 (Propriétés des identifiants de position)**.** Les propriétés que les identifiants de position doivent respecter sont les suivantes :

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

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

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

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

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

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

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

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

13. Un travail en cours [53] indique en effet qu'une séquence répliquée empêchant tout entrelacement est impossible.

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

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

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

Ces redéfinitions permettent de proposer une spécification de la séquence avec des modifications concurrentes qui commutent.

À partir de cette spécification, PREGUICA et al. propose un CRDT pour le type 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 1.17 illustre le fonctionnement de cette approche. La racine de l'arbre binaire,

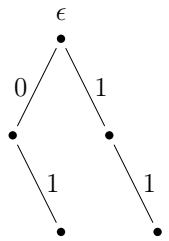


FIGURE 1.17 – Arbre pour générer des identifiants de positions

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 1.17, Treedoc lui associerait l'identifiant $id = \epsilon \oplus 1 \oplus 1 \oplus 1$. Ainsi, Treedoc suit l'ordre du parcours infixe de l'arbre binaire pour ordonner les identifiants de position.

Ce mécanisme souffre néanmoins d'un écueil : en l'état, plusieurs noeuds du système peuvent associer un même identifiant à des éléments insérés en concurrence, contrevenant alors à la propriété (ii). Pour corriger cela, Treedoc ajoute à chaque noeud de l'arbre un désambiguateur par élément : une paire $\langle nodeId, nodeSeq \rangle$. 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 en métadonnée 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 1.18 présente un exemple de cette situation. Dans cet exemple, deux identifiants

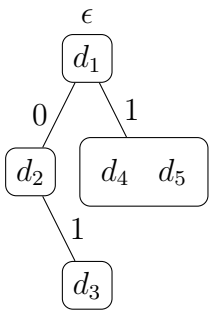


FIGURE 1.18 – Identifiants de position avec désambiguateurs

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

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

En concurrence, le noeud B insère aussi un élément "L" en fin de séquence. Cette modification résulte en l'opération $insert(\langle 1, d_5 \rangle, L)$, qui est intégrée. Le noeud B supprime ensuite l'élément "M" de la séquence, ce qui produit l'opération $remove(\langle \epsilon, d_1 \rangle)$. 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 $insert(\langle 1, d_5 \rangle, L)$ (resp. $insert(\langle 1, d_4 \rangle, L)$), il ajoute cet élément avec son désambiguateur dans son noeud de chemin 1, après (resp. avant) l'élément existant (on considère que $d_4 < d_5$).

B intègre ensuite $insert(1 \oplus \langle 1, d_6 \rangle, O)$. 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".

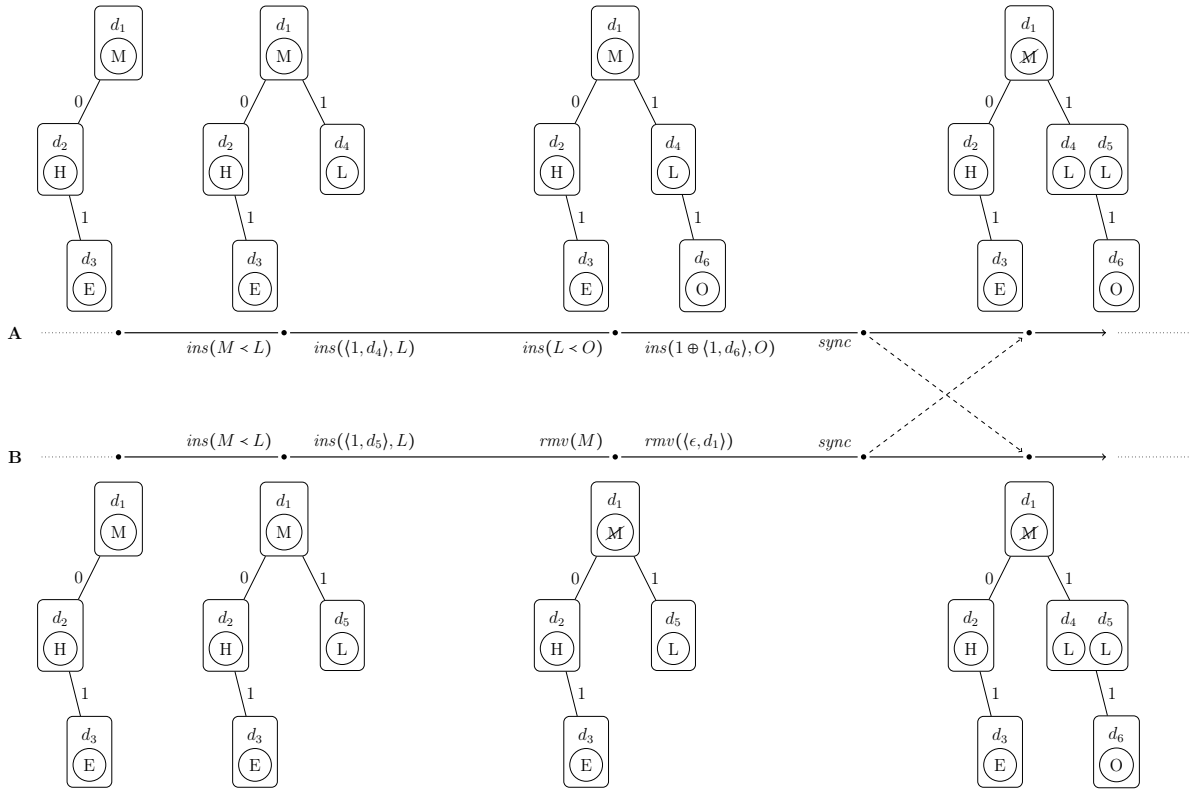


FIGURE 1.19 – Modifications concurrentes d'une séquence répliquée Treedoc

Intéressons-nous dorénavant au modèle de livraison requis par Treedoc. Dans [39], les auteurs indiquent reposer sur le modèle de livraison causal. En pratique, nous pouvons néanmoins relaxer le modèle de livraison comme expliqué dans [40] :

Définition 25 (Modèle de livraison Treedoc)**.** Le modèle de livraison Treedoc définit que :

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

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

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

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

Ensuite, comme illustré dans la Figure 1.19, Treedoc ne peut supprimer un noeud de l'arbre des identifiants lorsque ce dernier a 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 [39] 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 [52]. En effet, nous présentons un contre-exemple correspondant dans l'**??**.

Logoot

En parallèle à Treedoc [39], Weiss et al. [35] proposent Logoot. Ce CRDT pour le type 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 la Définition 24.

Les identifiants de position utilisés par Logoot sont spécifiés de manière différente dans [35] et [54]. Dans ce manuscrit, nous nous basons sur la spécification de [54] :

Définition 26 (Identifiant Logoot)**.** Un identifiant Logoot est une liste de tuples Logoot. Les tuples Logoot sont définis de la manière suivante :

Définition 26.1 (Tuple Logoot)**.** Un tuple Logoot est un triplet $\langle pos, nodeId, nodeSeq\rangle$ 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) *nodeSeq*, le numéro de séquence courant du noeud auteur de l'élément.

Dans le cadre de cette section, nous nous basons sur cette dernière spécification. Nous utiliserons la notation suivante $pos^{nodeId\ \ seq}$ pour représenter un tuple Logoot. Sans perdre en généralité, nous utiliserons des lettres minuscules comme valeurs pour *pos*, des lettres majuscules pour *nodeId* et des entiers naturels pour *nodeSeq*. 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 les uns par rapport aux autres, et ainsi d'ordonner les éléments associés. Pour définir $<_{id}$, Logoot se base sur l'ordre lexicographique.

Définition 27 (Relation $<_{id}$)**.** Étant donné deux identifiants $id = t_1 \oplus t_2 \oplus \cdots \oplus t_n$ et $id' = t'_1 \oplus t'_2 \oplus \cdots \oplus t'_m$, nous avons :

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

avec $<_t$ défini de la manière suivante :

Définition 27.1 (Relation $<_t$)**.** Étant donné deux tuples $t = \langle pos, nodeId, nodeSeq\rangle$ et $t' = \langle pos', nodeId', nodeSeq'\rangle$, nous avons :

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

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 $predId <_{id} id <_{id} succId$. Plusieurs algorithmes peuvent être utilisés pour cela. Notamment, [35] présente un algorithme permettant de générer N identifiants de manière aléatoire entre des identifiants *predId* et *succId*, mais reposant sur une représentation efficace des tuples en mémoire. Par souci de simplicité, nous présentons dans Algorithme 1 un algorithme naïf pour `generateId`.

Algorithme 1 Algorithme de génération d'un nouvel identifiant

1: **function** GENERATEID(predId $\in \mathbb{I}$, succId $\in \mathbb{I}$, nodeId $\in \mathbb{N}$, nodeSeq $\in \mathbb{N}^*$) : $\mathbb{I}$
▷ **precondition :** predId $<_{id}$ succId
2: **if** succId = predId $\oplus \langle \text{pos}_j, \text{nodeId}_j, \text{nodeSeq}_j \rangle \oplus \cdots$ **then**
▷ predId is a prefix of succId
3: pos $\leftarrow$ random $\in]\bot_{\mathbb{N}}, \text{pos}_j[$
4: id $\leftarrow$ predId $\oplus \langle \text{pos}, \text{nodeId}, \text{nodeSeq} \rangle$
5: **else if** predId = common $\oplus \langle \text{pos}_i, \text{nodeId}_i, \text{nodeSeq}_i \rangle \oplus \cdots \wedge$
succId = common $\oplus \langle \text{pos}_j, \text{nodeId}_j, \text{nodeSeq}_j \rangle \oplus \cdots \wedge$
$\text{pos}_j - \text{pos}_i \leq 1$ **then**
▷ Not enough space between predId and succId
▷ to insert new id with same length
▷ common may be empty
6: pos $\leftarrow$ random $\in]\text{pos}_{i+1}, \top_{\mathbb{N}}]$
7: id $\leftarrow$ common $\oplus \langle \text{pos}_i, \text{nodeId}_i, \text{nodeSeq}_i \rangle \oplus \langle \text{pos}, \text{nodeId}, \text{nodeSeq} \rangle$
8: **else**
▷ predId = common $\oplus \langle \text{pos}_i, \text{nodeId}_i, \text{nodeSeq}_i \rangle \oplus \cdots \wedge$
▷ succId = common $\oplus \langle \text{pos}_j, \text{nodeId}_j, \text{nodeSeq}_j \rangle \oplus \cdots \wedge$
▷ $\text{pos}_j - \text{pos}_i > 1$
▷ common may be empty
9: pos $\leftarrow$ random $\in]\text{pos}_i, \text{pos}_j[$
10: id $\leftarrow$ common $\oplus \langle \text{pos}, \text{nodeId}, \text{nodeSeq} \rangle$
11: **end if**
12: **return** id ▷ **postcondition :** predId $<_{id}$ id $<_{id}$ succId
13: **end function**

Pour illustrer cet algorithme, considérons son exécution avec :

(i) *predId* = e^{A1}, *nextId* = m^{B1}, *nodeId* = C et *nodeSeq* = 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 des valeurs de *nodeId* et *nodeSeq*, c.-à-d. *id* = l^{C1} (lignes 8-10).

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

recopier le second tuple de *predId* pour former *id* et y concaténer un nouveau tuple. Pour générer ce nouveau tuple, `generateId` choisit une valeur aléatoire entre la valeur de *pos* du troisième tuple de *predId* et la valeur maximale notée $\top_{\mathbb{N}}$. *predId* n'ayant pas de troisième tuple, `generateId` utilise la valeur minimale pour *pos*, $\bot_{\mathbb{N}}$. `generateId` choisit donc une valeur aléatoire dans $]\bot_{\mathbb{N}}, \top_{\mathbb{N}}]$ [14], 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 des valeurs de *nodeId* et *nodeSeq*, 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) $insert(pred \prec elt \prec succ)$ devient alors $insert(id, elt)$, avec $predId <_{id} id <_{id} succId$.

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

Les auteurs proposent ainsi une séquence répliquée avec des opérations concurrentes qui commutent.

Nous illustrons cela à l'aide de la Figure 1.20.

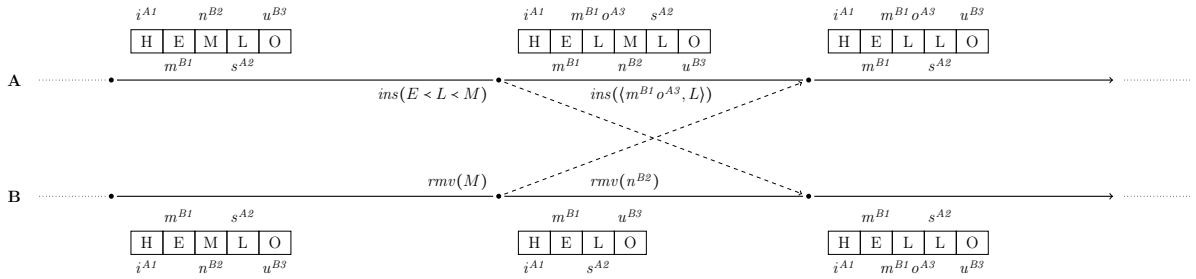


FIGURE 1.20 – Modifications concurrentes d'une séquence répliquée Logoot

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

Le noeud A insère l'élément "L" entre les éléments "E" et "M", c.-à-d. $insert(E \prec L \prec M)$. 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, $insert(m^{B1}\, o^{A3}, L)$, est générée, intégrée à la copie locale et diffusée.

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

À la réception de l'opération $remove(n^{B2})$, 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 $insert(m^{B1}\, o^{A3}, L)$. 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".

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

Concernant le modèle de livraison de Logoot, [35] indique se reposer sur le modèle de livraison causal. Nous constatons cependant que nous pouvons proposer un modèle de livraison moins contraint :

Définition 28 (Modèle de livraison Logoot)**.** Le modèle de livraison Logoot définit que :

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

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

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

Ainsi, Logoot peut adopter le même modèle de livraison que Treedoc, comme indiqué dans [40].

En contrepartie, Logoot souffre d'un problème de croissance de la taille des identifiants. Comme mis en lumière dans la Figure 1.20, Logoot génère des identifiants composés de plus en plus de tuples au fur et à mesure que l'espace des identifiants pour une taille donnée se sature. La croissance des identifiants a cependant plusieurs impacts négatifs :

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

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

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

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

Plusieurs extensions furent proposées pour Logoot. Weiss et al. [54] 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 *insert*, 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 [43]. 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 [54], les auteurs introduisent *Logoot-Undo*, une version de Logoot dotée d'un mécanisme d'annulation des modifications. 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. *insert*, *remove* 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 [15]. 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 *insert* avec l'opération *remove* 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. [51] introduisent *LogootSplit.* Reprenant les idées introduites par [50], ce travail présente un mécanisme d'aggrégation dynamiques des éléments 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 le type Séquence comme base pour les travaux présentés dans ce manuscrit. Nous dédions donc la section 1.4 à sa présentation en détails.

1.3.3 Synthèse

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

Cependant, une faiblesse de la littérature est à notre sens le couplage entre mécanismes de résolution de conflits et choix d'implémentations : plusieurs travaux [39, 35, 51, 49] ne séparent pas l'approche proposée pour rendre les modifications concurrentes commutatives des structures de données et algorithmes choisis pour représenter et manipuler la séquence et les identifiants, e.g. tableau dynamique, liste chaînée, liste chaînée + table de hachage + arbre binaire de recherche... Il en découle que les évaluations proposées par la communautée comparent au final des efforts d'implémentations plutôt que les approches elles-mêmes. En conséquence, la littérature ne permet pas d'établir la supériorité d'une approche sur l'autre.

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

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

15. Nous pouvons dès lors inférer le degré 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).

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

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

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

Définition 29 (Propriétés des identifiants dans approche à pierres tombales)**.** Les propriétés que doivent respecter les identifiants dans l'approche à pierres tombales sont les suivantes :

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

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

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

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

Définition 30 (Propriétés des identifiants dans approche à identifiants densément ordonnés)**.** Les propriétés que doivent respecter les identifiants dans l'approche à identifiants densément ordonnés sont les suivantes :

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

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

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

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

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

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

Pour respecter les propriétés (i) et (ii), les CRDTs reposent généralement sur des paires $\langle nodeId, nodeSeq \rangle$ avec :

(i) *nodeId*, l'identifiant du noeud qui génère le dot. Il est supposé unique.

(ii) *nodeSeq*, un entier propre au noeud, servant d'horloge logique. Il est incrémenté à chaque génération de dot.

Ainsi, un couple de taille fixe, $\langle nodeId, nodeSeq \rangle$, permet de respecter la contrainte d'unicité des identifiants.

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

des éléments insérés. Cette nuance se traduit dans la structure des identifiants, notamment leur taille.

Pour ordonner un identifiant par rapport à ceux générés en concurrence, l'approche à pierres tombales peut définir une relation d'ordre total strict sur leur dot respectif, e.g. en se basant sur l'ordre lexicographique. Un élément tiers peut y être ajouté si nécessaire, e.g. RGA et son horloge de Lamport [4]. Ainsi, les identifiants de cette approche peuvent être définis tout en ayant une taille fixe, c.-à-d. un nombre de composants fixe.

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

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

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

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

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

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

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

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

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

Pour compléter notre étude de ces approches, intéressons nous au modèle de livraison requis par ces dernières. Contrairement à ce qui peut être conjecturé après une lecture de la littérature, nous notons qu'aucune de ces approches ne requiert de manière intrinsèque une livraison causale de ses opérations. Ces deux approches peuvent donc utiliser des modèles de livraison plus faible que la livraison causale et ne nécessitant pas de vecteurs

de versions pour chaque message. Elles sont donc adaptées aux systèmes collaboratifs P2P à large échelle.

Finalement, nous notons que l'ensemble des CRDTs pour le type Séquence proposés souffrent du problème de l'entrelacement présenté dans [52]. Nous conjecturons cependant que les CRDTs pour le type Séquence à pierres tombales sont moins sujets à ce problème. En effet, dans cette approche, l'algorithme d'intégration des nouveaux éléments repose généralement sur l'élément précédent. Ainsi, une séquence d'insertions séquentielles produit une sous-chaîne d'éléments. L'algorithme d'intégration permet ensuite d'intégrer sans entrelacement de telles sous-chaînes générées en concurrence, e.g. dans le cadre de sessions de travail asynchrones. Cependant, il s'agit d'une garantie offerte par l'approche à pierres tombales dont nous ne retrouvons pas d'équivalent dans l'approche à identifiants densément ordonnés. Pour confirmer notre conjecture et évaluer son impact sur l'expérience utilisateur, il conviendrait de mener un ensemble d'expériences utilisateurs dans la lignée de [43, 55, 56].

Nous récapitulons cette discussion dans le Tableau 1.2.

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

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

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

1.4 LogootSplit

LogootSplit [51] est à notre connaissance le dernier CRDT pour le type Séquence appartenant à l'approche à identifiants densément ordonnés proposé. Ce CRDT propose un mécanisme permettant d'aggréger de manière dynamique des éléments en blocs d'éléments.

L'aggrégation des éléments en blocs offre plusieurs bénéfices. Tout d'abord, elle permet de factoriser les métadonnées des éléments aggrégés en un même bloc, ce qui réduit le surcoût en métadonnées du CRDT. Ensuite, la séquence stocke directement les blocs, en place et lieu des éléments, ce qui réduit sa taille et rend sa manipulation plus efficace. Finalement, les blocs permettent de représenter des modifications à l'échelle de plusieurs éléments, ce qui réduit la taille des messages diffusés sur le réseau.

Nous détaillons ci-dessous le fonctionnement de LogootSplit.

1.4.1 Identifiants

LogootSplit associe aux éléments des identifiants définis de la manière suivante :

Définition 31 (Identifiant LogootSplit)**.** Un identifiant LogootSplit est une liste de tuples LogootSplit.

avec les tuples LogootSplit définis de la manière suivante :

Définition 32 (Tuple LogootSplit)**.** Un tuple LogootSplit est un quadruplet $\langle pos, nodeId, nodeSeq, offset\rangle$ avec :

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

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

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

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

Dans ce manuscrit, nous représentons les tuples LogootSplit par le biais de la notation suivante : $position_{offset}^{nodeId\ nodeSeq}$. Sans perdre en généralité, nous utiliserons des lettres minuscules comme valeurs pour *pos*, des lettres majuscules pour *nodeId*, des entiers naturels pour *nodeSeq* et des entiers relatifs pour *offset*. Par exemple, nous représentons l'identifiant $\langle\langle i, A, 1, 0\rangle\langle f, B, 1, 0\rangle\rangle$ par $i_0^{A1} f_0^{B1}$.

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

Définition 33 (Relation $<_{id}$)**.** Étant donné deux identifiants $id = t_1 \oplus t_2 \oplus \cdots \oplus t_n$ et $id' = t'_1 \oplus t'_2 \oplus \cdots \oplus t'_m$, nous avons :

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

avec la relation d'ordre strict total les tuples $<_t$ définie de la manière suivante :

Définition 34 (Relation $<_t$)**.** Étant donné deux tuples $t = \langle pos, nodeId, nodeSeq, offset\rangle$ et $t' = \langle pos', nodeId', nodeSeq', offset'\rangle$, nous avons :

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

Par exemple, nous avons :

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

(ii) $i_0^{B1} <_{id} i_0^{B1} f_0^{A1}$ car le premier identifiant est un préfixe du second identifiant.

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

1.4.2 Aggrégation dynamique d'éléments en blocs

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

Définition 35 (Intervalle d'identifiants)**.** Un intervalle d'identifiants est un couple $\langle idBegin, offsetEnd \rangle$ avec :

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

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

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

LogootSplit regroupe les éléments avec des identifiants *contigus* dans un intervalle.

Définition 36 (Identifiants contigus)**.** Deux identifiants sont contigus si et seulement si les deux identifiants sont identiques à l'exception de leur dernier offset et que leur derniers offsets sont consécutifs.

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

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

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

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

Définition 37 (Bloc)**.** Un bloc est un triplet $\langle idInterval, elts, isAppendable \rangle$ avec :

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

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

(iii) *isAppendable*, un booléen indiquant si l'auteur du bloc peut ajouter de nouveaux éléments en fin de bloc [16].

Nous représentons un exemple de séquence LogootSplit dans la Figure 1.21. Dans la Figure 1.21a, les identifiants i_0^{B1}, i_1^{B1} et i_2^{B1} forment une chaîne d'identifiants contigus. LogootSplit est donc capable de regrouper ces éléments en un bloc représentant l'intervalle d'identifiants $i_{0..2}^{B1}$ pour minimiser les métadonnées stockées, comme illustré dans la Figure 1.21b.

16. De manière similaire, il est possible de permettre à l'auteur du bloc d'ajouter de nouveaux éléments en début de bloc à l'aide d'un booléen *isPrependable*. Cette fonctionnalité est cependant incompatible avec le mécanisme que nous proposons dans le chapitre 2. Nous faisons donc le choix de la retirer.

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

FIGURE 1.21 – Représentation d'une séquence LogootSplit contenant les éléments "HLO"

Au lieu de stocker les éléments directement, une séquence LogootSplit stocke les blocs les contenant [17]. Ce changement de granularité permet d'améliorer les performances de la structure de données sur plusieurs aspects :

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

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

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

Il est intéressant de noter que la paire $\langle nodeId, nodeSeq \rangle$ du dernier tuple d'un identifiant permet d'identifier de manière unique la partie commune des identifiants de l'intervalle d'identifiants auquel il appartient. Ainsi, nous pouvons identifier de manière unique un intervalle d'identifiants avec le quadruplet $\langle nodeId, nodeSeq, offsetBegin, offsetEnd \rangle$. Par exemple, l'intervalle d'identifiants $i_1^{B1} f_{2..4}^{A1}$ peut être référencé à l'aide du quadruplet $\langle A, 1, 2, 4 \rangle$.

1.4.3 Modèle de données

En nous basant sur ANDRÉ et al. [51], nous proposons une définition du modèle de données de LogootSplit dans la Figure 1.22 :

Une séquence LogootSplit est une séquence de blocs. Concernant les modifications définies sur le type, nous nous inspirons de [13] et les séparons en deux étapes :

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

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

La séquence LogootSplit autorise deux types de modifications :

17. Par abus de notation, nous représenterons les blocs de taille 1, c.-à-d. ne contenant qu'un seul élément, par des éléments dans nos schémas.

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

constructor			
empty	:		$\longrightarrow S$
prepare			
insert	:	$S \times \mathbb{N} \times Array\langle E\rangle \times \mathbb{I} \times \mathbb{N}^*$	$\longrightarrow Id \times Array\langle E\rangle$
remove	:	$S \times \mathbb{N} \times \mathbb{N}$	$\longrightarrow Array\langle IdInterval\rangle$
effect			
insert	:	$S \times Id \times Array\langle E\rangle$	$\longrightarrow S$
remove	:	$S \times Array\langle IdInterval\rangle$	$\longrightarrow S$
queries			
length	:	S	$\longrightarrow \mathbb{N}$
read	:	S	$\longrightarrow Array\langle E\rangle$

FIGURE 1.22 – Spécification algébrique du type abstrait LogootSplit

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

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

Nous présentons dans la Figure 1.23 un exemple d'utilisation de cette séquence répliquée.

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 $i^{B1}_{0..3}$ 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^{B1}_1. Cette opération est intégrée à sa copie locale et envoyée au noeud B pour qu'il intègre

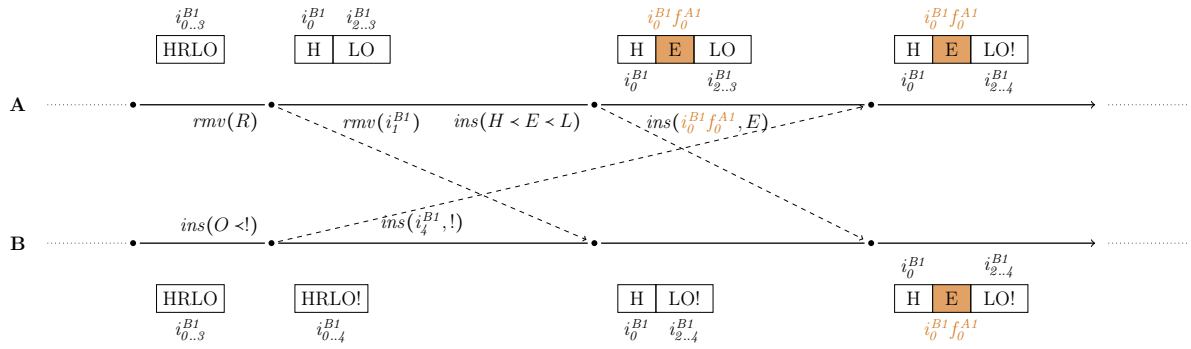


FIGURE 1.23 – Modifications concurrentes d'une séquence répliquée LogootSplit

cette modification à son tour.

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

$$i_0^{B1} <_{id} id <_{id} i_2^{B1}$$

L'espace des identifiants de taille 1 étant saturé entre ces deux identifiants, A génère *id* en reprenant le premier tuple de l'identifiant du prédecesseur et en y concaténant un nouveau tuple : $id = i_0^{B1} \oplus f_0^{A1}$. LogootSplit génère l'opération *insert* correspondante, indiquant l'élément à insérer et sa position grâce à son identifiant. Il intègre cette opération et la diffuse sur le réseau.

En parallèle, le noeud B insère l'élément " !" en fin de la séquence. Comme le noeud B est l'auteur du bloc $i_{0..3}^{B1}$, il peut y ajouter de nouveaux éléments. B associe donc l'identifiant i_4^{B1} à l'élément " !" pour l'ajouter au bloc existant. Il génère l'opération *insert* correspondante, l'intègre puis la diffuse.

Les noeuds se synchronisent ensuite. Le noeud A reçoit l'opération $insert(i_4^{B1}, L)$. Le noeud A détermine que cet élément doit être inséré à la fin de la séquence, puisque $i_3^{B1} <_{id} i_4^{B1}$. Ces deux identifiants étant contigus, il ajoute cet élément au bloc existant.

De son côté, le noeud B reçoit tout d'abord l'opération $remove(i_1^{B1})$. Le noeud B supprime donc l'élément correspondant de son état, "R".

Il reçoit ensuite l'opération $insert(i_0^{B1}f_0^{A1}, E)$. Le noeud B insère cet élément entre les éléments "H" et "L", puisqu'on a :

$$i_1^{B1} <_{id} i_0^{B1}f_0^{A1} <_{id} i_2^{B1}$$

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

1.4.4 Modèle de livraison

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

Livraison des opérations en exactement un exemplaire

Ce premier exemple, représenté par la Figure 1.24, a pour but d'illustrer la nécessité de la propriété de livraison en *exactement un exemplaire* des opérations.

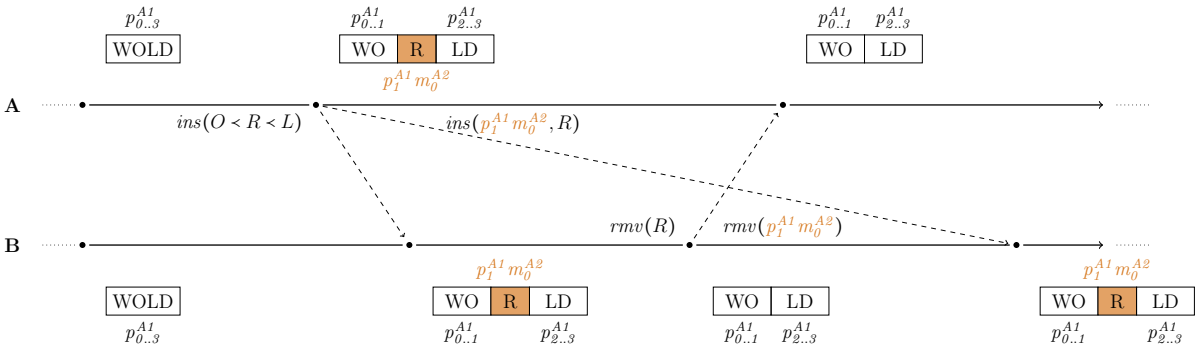


FIGURE 1.24 – Résurgence d'un élément supprimé suite à la relivraison de son opération *insert*

Dans cet exemple, deux noeuds A et B répliquent et éditent collaborativement une séquence. La séquence répliquée contient initialement les éléments "WOLD", qui sont associés à l'intervalle d'identifiants $p^{A1}_{0..3}$.

Le noeud A commence par insérer l'élément "R" dans la séquence entre les éléments "O" et "L". A intègre l'opération résultante, $insert(p^{A1}_1 m^{A2}_0, R)$ puis la diffuse au noeud B.

À la réception de l'opération *insert*, le noeud B l'intègre à son état. Puis il supprime dans la foulée l'élément "R" nouvellement inséré. B intègre l'opération $remove(p^{A1}_1 m^{A2}_0)$ puis l'envoie au noeud A.

Le noeud A intègre l'opération *remove*, ce qui a pour effet de supprimer l'élément "R" associé à l'identifiant $p^{A1}_1 m^{A2}_0$. Il obtient alors un état équivalent à celui du noeud B.

Cependant, l'opération *insert* insérant l'élément "R" à la position $p^{A1}_1 m^{A2}_0$ est de nouveau envoyée au noeud B. De multiples raisons peuvent être à l'origine de ce nouvel envoi : perte du message d'*acknowledgment*, utilisation d'un protocole de diffusion épidémique des messages, déclenchement du mécanisme d'anti-entropie en concurrence... Le noeud B ré-intègre alors l'opération *insert*, ce qui fait revenir l'élément "R" 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.

Livraison de l'opération *remove* après les opérations *insert* correspondantes

La Figure 1.25 présente un second exemple illustrant la nécessité de la contrainte de livraison d'une opération *remove* qu'après la livraison des opérations *insert* correspondantes.

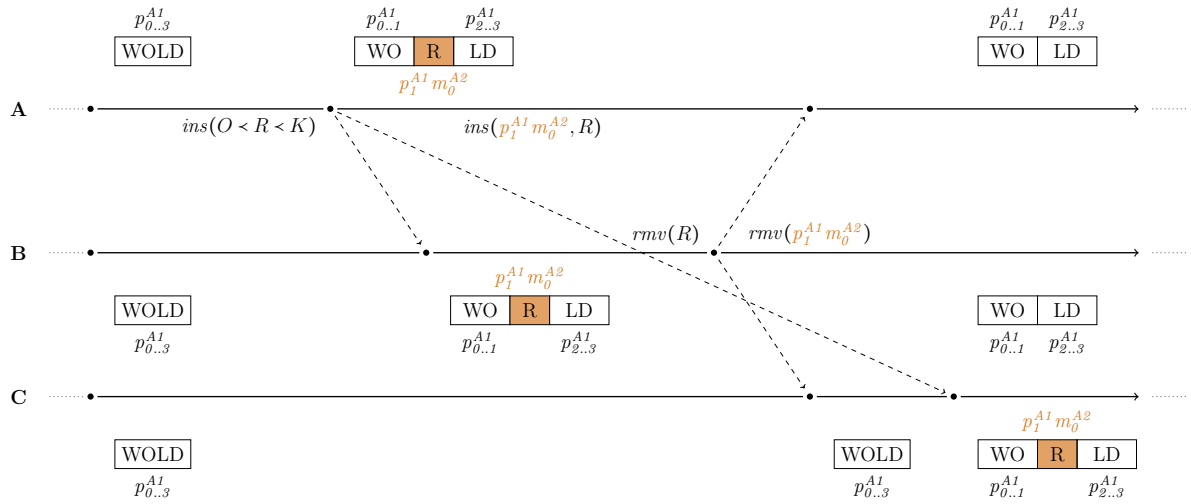


FIGURE 1.25 – Non-effet de l'opération *remove* car reçue avant l'opération *insert* correspondante

Dans cet exemple, trois noeuds A, B et C répliquent et éditent collaborativement une séquence. La séquence répliquée contient initialement les éléments "WOLD", qui sont associés à l'intervalle d'identifiants $p_{0..3}^{A1}$.

Le noeud A commence par insérer l'élément "R" dans la séquence entre les éléments "O" et "L". A intègre l'opération résultante, $insert(p_1^{A1}\, m_0^{A2}, R)$ puis la diffuse.

À la réception de l'opération *insert*, le noeud B l'intègre à son état. Puis il supprime dans la foulée l'élément "R" nouvellement inséré. B intègre l'opération $remove(p_1^{A1}\, m_0^{A2})$ puis la diffuse.

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

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

Nous constatons alors que l'état à terme du noeud C diverge de celui des noeuds A et B, et cela malgré que les noeuds A, B et C aient intégré le même ensemble d'opérations. Ce résultat transgresse la propriété Cohérence forte à terme (SEC) [5] 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.

Définition du modèle de livraison

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

Définition 38 (Modèle de livraison LogootSplit)**.** Le modèle de livraison LogootSplit définit que :

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

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

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

Il est à noter que Elvinger [40] a récemment proposé dans ses travaux de thèse Dotted LogootSplit, un nouveau CRDT pour le type Séquence dont la synchronisation est basée sur les différences d'états. Inspiré de Logoot et LogootSplit, ce nouveau CRDT associe une séquence à identifiants densément ordonnés à un contexte causal. Le contexte causal est une structure de données permettant à Dotted LogootSplit de représenter et de maintenir efficacement les informations des modifications déjà intégrées à l'état courant. Cette association permet à Dotted LogootSplit de fonctionner de manière autonome, sans imposer de contraintes particulières à la couche livraison autres que la livraison à terme.

1.4.5 Limites de LogootSplit

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

Croissance non-bornée de la taille des identifiants

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

Cependant, LogootSplit introduit un mécanisme favorisant la croissance des identifiants : les intervalles d'identifiants. Considérons l'exemple présenté dans la Figure 1.26.

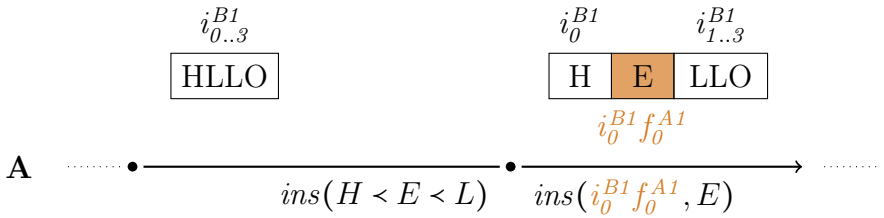


Figure 1.26 – Insertion menant à une augmentation de la taille des identifiants

Dans cet exemple, le noeud A insère un nouvel élément dans un intervalle d'identifiants existant, c.-à-d. entre deux identifiants contigus : i^{B1}_0 et i^{B1}_1. Ces deux identifiants étant contigus, il n'est pas possible de générer id, un identifiant de même taille tel que $i^{B1}_0 <_{id} id <_{id} i^{B1}_1$. Pour respecter l'ordre souhaité, LogootSplit génère donc un identifiant à partir de l'identifiant du prédecesseur et en y ajoutant un nouveau tuple, e.g. $i^{B1}_0 f^{A1}_0$.

Par conséquent, la taille des identifiants croît à chaque fois qu'un intervalle d'identifiants est scindé. Comme présenté précédemment (cf. sous-section 1.3.3, page 39), cette croissance augmente le surcoût en métadonnées, en calculs et en bande-passante du CRDT.

Fragmentation de la séquence en blocs courts

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

Définition 39 (Contraintes sur l'ajout d'éléments à un bloc existant)**.** L'ajout d'éléments à un bloc existant doit respecter les règles suivantes :

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

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

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

La Figure 1.27 illustre ces règles.

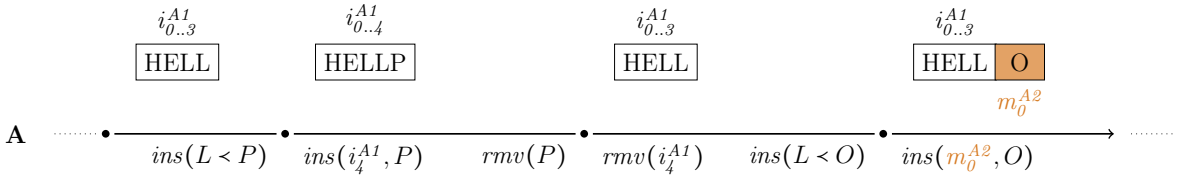


FIGURE 1.27 – Insertion menant à une augmentation de la taille des identifiants

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

Synthèse

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

Initialement, nous nous sommes focalisés sur un aspect du problème : la croissance du surcoût en métadonnées de la structure. Afin de quantifier ce problème, nous avons évalué par le biais de simulations [18] l'évolution de la taille de la séquence. La Figure 1.28 présente le résultat obtenu.

Sur cette figure, nous représentons l'évolution au fur et à mesure que des modifications sont effectuées sur une séquence LogootSplit de la taille de son contenu, sous la forme d'une ligne pointillée bleu, et de la taille de la séquence LogootSplit complète, sous la

18. Nous détaillons le protocole expérimental que nous avons défini pour ces simulations dans le chapitre 2.

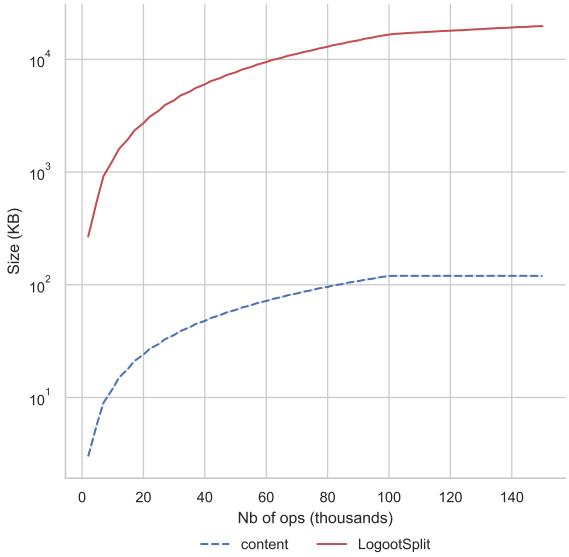


FIGURE 1.28 – Taille du contenu comparé à la taille de la séquence LogootSplit

forme d'une ligne continue rouge. Nous constatons que le contenu représente à terme moins de 1% de taille de la structure de données. Les 99% restants correspondent aux métadonnées utilisées par la séquence répliquée, c.-à-d. la taille des identifiants, les blocs composant la séquence LogootSplit, mais aussi la structure de données utilisée en interne pour représenter la séquence de manière efficace.

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

1.5 Mitigation du surcoût des séquences répliquées sans conflits

L'augmentation du surcoût des CRDTs pour le type Séquence, qu'il soit dû à des pierres tombales ou à des identifiants de taille non-bornée, est un problème bien identifié dans la littérature [38, 39, 57, 58, 59, 60]. Plusieurs approches ont donc été proposées pour réduire sa croissance.

1.5.1 Mécanisme de Garbage Collection des pierres tombales

Pour réduire l'impact des pierres tombales sur les performances de RGA, [38] propose un mécanisme de Garbage Collection (GC) des pierres tombales. Pour rappel, ce mécanisme nécessite qu'une pierre tombale ne puisse plus être utilisée comme prédecesseur par une opération *insert* reçue dans le futur pour pouvoir être supprimée définitivement. En d'autres termes, ce mécanisme repose sur la stabilité causale de l'opération de suppression pour supprimer la pierre tombale correspondante.

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

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

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

Concernant l'approche à identifiants densément ordonnés, Letia et al. [57] puis Zawirski et al. [58] proposent un mécanisme de ré-équilibrage de l'arbre des identifiants de position pour Treedoc [39]. Pour rappel, Treedoc souffre des problèmes suivants :

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

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

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

Notons que l'opération *rebalance* contrevient à une des propriétés des identifiants de position densément ordonnés : leur *immutabilité* (cf. Définition 30, page 40). L'opération *rebalance* est donc intrinséquement non-commutative avec les opérations *insert* et *remove* concurrentes. Pour assurer la convergence à terme des copies, les auteurs mettent en place un mécanisme de *catch-up*. Ce mécanisme consiste à transformer les opérations concurrentes aux opérations *rebalance* avant de les intégrer, de façon à prendre en compte les effets des ré-équilibrages.

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

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

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

Ainsi, cette solution permet d'adapter l'utilisation d'un protocole de consensus à un système P2P dynamique. Cependant, elle requiert de disposer de noeuds stables et bien

connectés dans le système pour former le *core*. Cette condition est un obstacle pour le déploiement et la pérennité de cette solution.

1.5.3 Ralentissement de la croissance des identifiants de position

L'approche LSEQ [59, 60] est une approche visant à ralentir la croissance des identifiants dans les Séquences CRDTs à identifiants densément ordonnés. Au lieu de réduire périodiquement la taille des métadonnées liées aux identifiants à l'aide d'un mécanisme coûteux de ré-équilibrage de l'arbre des identifiants de position [58], les auteurs définissent de nouvelles stratégies d'allocation des identifiants pour réduire leur vitesse de croissance.

Dans ces travaux, les auteurs notent que les stratégies d'allocation des identifiants proposées pour Logoot dans [35] et [54] ne sont adaptées qu'à un seul comportement d'édition : l'édition séquentielle. Si les insertions sont effectuées en suivant d'autres comportements, les identifiants générés saturent rapidement l'espace des identifiants pour une taille donnée. Les insertions suivantes déclenchent alors une augmentation de la taille des identifiants. En conséquent, la taille des identifiants dans Logoot augmente de façon linéaire au nombre d'insertions, au lieu de suivre la progression logarithmique attendue.

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

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

Nous conjecturons cependant que cette approche perd ses bienfaits lorsqu'elle est couplée avec un CRDT pour le type Séquence à granularité variable. En effet, comme évoqué précédemment, toute insertion au sein d'un bloc provoque une augmentation de la taille de l'identifiant résultant (cf. section 1.4.5, page 50).

1.5.4 Synthèse

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

(i) Les approches présentées dans [38, 57, 58] reposent chacune sur des contraintes fortes dans les systèmes P2P dynamiques, c.-à-d. respectivement la stabilité causale des opérations et l'utilisation d'un protocole de consensus. Dans un système dans lequel nous n'avons aucun contrôle sur les noeuds et notamment leur disponibilité, ces contraintes nous apparaissent rédhibitoires.

(ii) L'approche présentée dans [59, 60] est conçue pour les CRDTs pour le type Séquence à identifiants densément ordonnés à granularité fixe. L'introduction de mécanismes d'aggrégation dynamique des éléments en blocs comme ceux présentés dans [51, 49], avec les contraintes qu'ils introduisent, nous semble contrarier les efforts effectués pour réduire la croissance des identifiants de position.

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

1.6 Synthèse

Les systèmes distribués adoptent le modèle de la réplication optimiste [2] pour offrir de meilleures garanties à leurs utilisateur-rices, en termes de disponibilité, latence et capacité de tolérance aux pannes [61].

Dans ce modèle, chaque noeud du système possède une copie de la donnée et peut la modifier sans coordination avec les autres noeuds. Il en résulte des divergences temporaires entre les copies respectives des noeuds. Pour résoudre les potentiels conflits provoqués par des modifications concurrentes et assurer la convergence à terme des copies, les systèmes ont tendance à utiliser les CRDTs [5] en place et lieu des types de données séquentiels.

Plusieurs CRDTs pour le type Séquence ont été proposés, notamment pour permettre la conception d'éditeurs collaboratifs pair-à-pair. Ces CRDTs peuvent être regroupés en deux catégories en fonction de leur mécanisme de résolution de conflits : l'approche à pierres tombales [37, 44, 43, 38, 49, 52] et l'approche à identifiants densément ordonnées [39, 35, 54, 51, 40].

Chacune de ces approches introduit néanmoins un surcoût croissant, au moins en termes de métadonnées et de calculs, pénalisant leurs performances à terme. Pour résoudre ce problème, plusieurs travaux ont été proposés, notamment [57, 58]. Cette approche présente un mécanisme de ré-équilibrage de l'arbre des identifiants de position pour les CRDTs pour le type Séquence à identifiants densément ordonnés.

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

1.7 Proposition

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

Ce mécanisme se distingue des travaux existants, notamment de [57, 58], par les aspects suivants :

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

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

Nous concevons ce mécanisme pour le CRDT LogootSplit. Toutefois, le principe de notre approche est générique. Ainsi, ce mécanisme peut être adapté pour proposer un équivalent pour d'autres CRDTs pour le type Séquence, e.g. RGASplit [49].

Nous présentons et détaillons ce mécanisme dans le chapitre suivant.

Chapitre 2

Renommage dans une séquence répliquée

Sommaire

Nous proposons un nouveau CRDT pour le type Séquence appartenant à l'approche à identifiants densément ordonnés et à granularité variable : RenamableLogootSplit [62, 63]. Cette structure de données permet aux noeuds d'insérer et de supprimer des éléments au sein d'une séquence répliquée. Elle incorpore un mécanisme de renommage qui se déclenche par le biais d'une nouvelle opération : l'opération *rename*. Cette dernière permet de :

(i) Réassigner des identifiants plus courts aux différents éléments de la séquence.

(ii) Fusionner les blocs composant la séquence.

Ces deux actions permettent au mécanisme de renommage de produire un nouvel état minimisant le surcoût du CRDT en termes de métadonnées et de calculs pour les modifications suivantes.

Une particuliarité de l'opération *rename* est qu'elle ne modifie pas le contenu de la séquence répliquée, mais uniquement ses identifiants. Puisque les identifiants sont des métadonnées utilisées par la structure de données uniquement afin de résoudre les conflits, les utilisateurs ignorent leur existence. Les opérations *rename* sont donc des opérations systèmes : elles sont émises et appliquées par les noeuds en coulisses, sans aucune intervention des utilisateurs.

Afin de garantir le respect du modèle de cohérence SEC, nous définissons plusieurs propriétés de sécurité que l'opération *rename* doit respecter. Ces propriétés sont inspirées principalement par celles proposées dans [58].

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

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

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

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

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

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

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

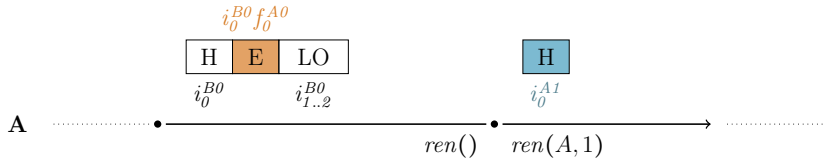
Dans ce chapitre, nous présentons donc le mécanisme de renommage intégré à RenamableLogootSplit et le mécanisme de résolution de conflits additionnel qu'il utilise. Dans la section 2.1, nous présentons l'opération *rename*, c.-à-d. son fonctionnement et son impact sur le modèle de livraison du CRDT, ainsi que le mécanisme de résolution de conflits pour gérer les opérations *insert* et *remove* concurrentes. Ensuite, nous présentons le mécanisme de résolution de conflits pour gérer les opérations *rename* concurrentes dans la section 2.2. Dans la section 2.3, nous présentons un mécanisme de Garbage Collection (GC) des métadonnées propres au mécanisme de renommage. Dans la section 2.4, nous présentons une validation de notre approche par le biais d'une analyse en complexité et d'une évaluation expérimentale. Nous présentons ensuite plusieurs limites et perspectives de notre approche dans la section 2.5. Finalement, nous comparons notre contribution et ses résultats aux approches existantes dans la section 2.6.

2.1 Introduction de l'opération de renommage

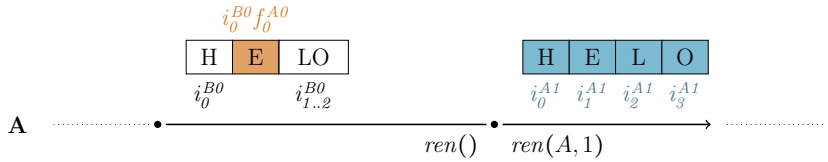
2.1.1 Opération de renommage proposée

L'opération *rename*, abrégée en *ren* dans nos figures, réassigne des identifiants arbitraires aux éléments de la séquence de réduire son surcoût en métadonnées.

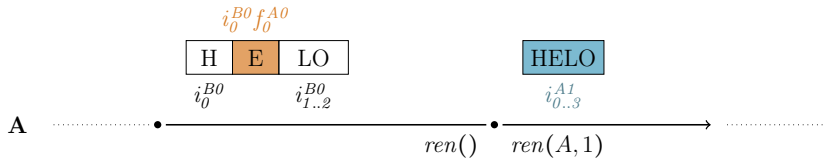
Son comportement est illustré dans la Figure 2.1. Dans cet exemple, le noeud A initie une opération *rename* sur son état local. Tout d'abord, le noeud A génère un nouvel identifiant à partir du premier tuple de l'identifiant du premier élément de la séquence (i_0^{B0}). Pour générer ce nouvel identifiant, le noeud A reprend la position de ce tuple (i) mais utilise son propre identifiant de noeud (**A**) et numéro de séquence actuel (*1*). De plus, son offset est mis à 0. Le noeud A réassigne l'identifiant résultant (i_0^{A1}) au premier élément de la séquence, comme décrit dans la Figure 2.1a. Ensuite, le noeud A dérive des identifiants contigus pour tous les éléments restants en incrémentant de manière successive l'offset (i_1^{A1}, i_2^{A1}, i_3^{A1}), comme présenté dans la Figure 2.1b. Comme nous assignons des identifiants consécutifs à tous les éléments de la séquence, nous pouvons au final aggréger



(a) Choix du nouvel identifiant pour le premier élément



(b) Choix des nouveaux identifiants pour les éléments restants



(c) État final obtenu

FIGURE 2.1 – Renommage de la séquence sur le noeud A

ces éléments en un seul bloc, comme illustré en Figure 2.1c. Ceci permet aux noeuds de bénéficier au mieux de la fonctionnalité des blocs et de minimiser le surcoût en métadonnés de l'état résultat.

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

Définition 40 (Ancien état)**.** Un *ancien état* est la liste des intervalles d'identifiants qui composent l'état courant de la séquence répliquée au moment du renommage.

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

Définition 41 (rename)**.** Une opération *rename* est un triplet $\langle nodeId, nodeSeq, formerState \rangle$ où

(i) *nodeId* est l'identifiant du noeud qui a généré l'opération *rename*.

(ii) *nodeSeq* est le numéro de séquence du noeud au moment de la génération de l'opération *rename*.

(iii) *formerState* est l'ancien état du noeud au moment du renommage.

En utilisant ces données, les autres noeuds calculent le nouvel identifiant de chaque identifiant renommé. Concernant les identifiants insérés de manière concurrente au renommage, nous expliquons dans la sous-section 2.1.2 comment les noeuds peuvent les renommer de manière déterministe.

2.1.2 Gestion des opérations d'insertion et de suppression concurrentes au renommage

Après avoir appliqué des opérations *rename* sur leur état local, les noeuds peuvent recevoir des opérations concurrentes. La Figure 2.2 illustre de tels cas.

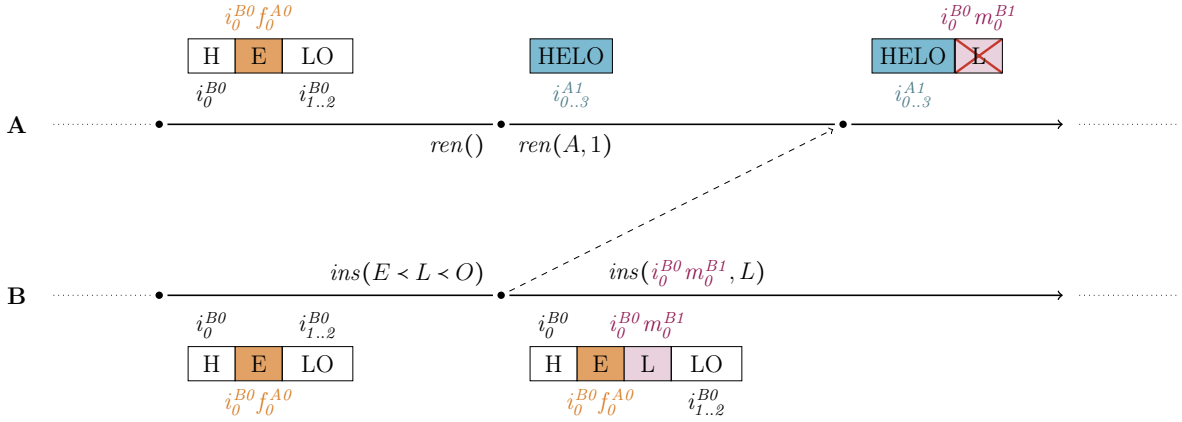


FIGURE 2.2 – Modifications concurrentes menant à une anomalie

Dans cet exemple, le noeud B insère un nouvel élément "L", lui assigne l'identifiant $i_0^{B0}\, m_0^{B1}$ et diffuse cette modification, de manière concurrente à l'opération *rename* décrite dans la Figure 2.2. À la réception de l'opération *insert*, le noeud A ajoute l'élément inséré au sein de sa séquence, en utilisant l'identifiant de l'élément pour déterminer sa position. Cependant, puisque les identifiants ont été modifiés par l'opération *rename* concurrente, le noeud A insère le nouvel élément à la fin de sa séquence (puisque $i_3^{A1} <_{id} i_0^{B0}\, m_0^{B1}$) au lieu de l'insérer à la position souhaitée. Comme illustré par cet exemple, appliquer naivement les modifications concurrentes provoquerait des anomalies. Il est donc nécessaire de traiter les opérations concurrentes aux opérations *rename* de manière particulière.

Tout d'abord, les noeuds doivent détecter les opérations concurrentes aux opérations *rename*. Pour cela, nous utilisons un système basé sur des *époques*. Initialement, la séquence répliquée débute à l'époque *origine* notée ε_0. Chaque opération *rename* introduit une nouvelle époque et permet aux noeuds d'y avancer depuis l'époque précédente. Par exemple, l'opération *rename* décrite dans la Figure 2.2 permet aux noeuds de faire progresser leur état de ε_0 à ε_{A1}. Nous définissons les époques de la manière suivante :

Définition 42 (Époque)**.** Une époque est un couple $\langle nodeId, nodeSeq \rangle$ où

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

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

Au fur et à mesure que les noeuds reçoivent des opérations *rename*, ils construisent et maintiennent localement la *chaîne des époques*. Cette structure de données ordonne les époques en fonction de leur relation *parent-enfant* et associe à chaque époque l'*ancien*

état correspondant (c.-à-d. l'*ancien état* inclus dans l'opération *rename* qui a généré cette époque). De plus, les noeuds marquent chaque opération avec leur époque courante au moment de génération de l'opération. À la réception d'une opération, les noeuds comparent l'époque de l'opération à l'époque courante de leur séquence.

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

Les noeuds utilisent la fonction RENAMEID, décrite dans l'Algorithme 2, pour transformer les opérations *insert* et *remove* par rapport aux opérations *rename*. Cet algorithme associe les identifiants d'une époque *parente* aux identifiants correspondant dans l'époque *enfant*. L'idée principale de cet algorithme est de renommer les identifiants inconnus au moment de la génération de l'opération *rename* en utilisant leur prédecesseur. Un exemple est présenté dans la Figure 2.3. Cette figure décrit le même scénario que la Figure 2.2, à l'exception que le noeud A utilise RENAMEID pour renommer l'identifiants généré de façon concurrente avant de l'insérer dans son état.

Algorithme 2 Fonctions principales pour renommer un identifiant

```
 1: function RENAMEID(id ∈ 𝕀, renamedIds ∈ Array⟨𝕀⟩, nodeId ∈ ℕ, nodeSeq ∈ ℕ) : 𝕀
                                                  ▷ renamedIds = [id_0, id_1, ···, id_{n-2}, id_{n-1}]
 2:     firstId ← id_0
 3:     lastId ← id_{n-1}
                                                  ▷ firstId = ⟨pos, _, _, _⟩ ⊕ suffix
 4:
 5:     if id <_id firstId then
 6:         newFirstId ← ⟨pos, nodeId, nodeSeq, 0⟩
 7:         return renIdLessThanFirstId(id, newFirstId)
 8:     else if id ∈ renamedIds then
                                                  ▷ id = id_i
 9:         return ⟨pos, nodeId, nodeSeq, i⟩
10:     else if lastId <_id id then
11:         newLastId ← ⟨pos, nodeId, nodeSeq, n − 1⟩
12:         return renIdGreaterThanLastId(id, newLastId)
13:     else
14:         return renIdFromPredId(id, renamedIds, pos, nodeId, nodeSeq)
15:     end if
16: end function
17:
18: function RENIDFROMPREDID(id ∈ 𝕀, renamedIds ∈ Array⟨𝕀⟩, pos ∈ ℕ, nodeId ∈ ℕ, nodeSeq ∈ ℕ) : 𝕀
                                                  ▷ renamedIds = [id_0, id_1, ···, id_{n-2}, id_{n-1}]
19:     Find id_i ∈ renamedIds such that id_i <_id id <_id id_{i+1}
20:     newPredId ← ⟨pos, nodeId, nodeSeq, i⟩
21:     return newPredId ⊕ id
22: end function
```

L'algorithme procède de la manière suivante. Tout d'abord, le noeud récupère le prédecesseur de l'identifiant donné $i_0^{B0} m_0^{B1}$ dans l'ancien état : $i_0^{B0} f_0^{A0}$ (ligne 19). Ensuite, il calcule l'équivalent de $i_0^{B0} f_0^{A0}$ dans l'état renommé : i_1^{A1} (ligne 20). Finalement, le noeud A concatène (noté $\oplus$) cet identifiant et l'identifiant donné pour générer l'identifiant cor-

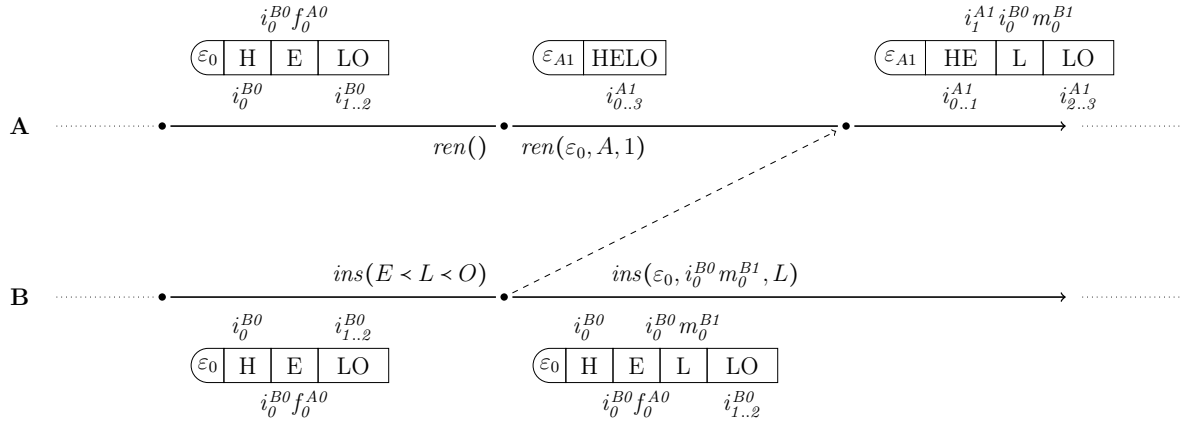


FIGURE 2.3 – Renommage de la modification concurrente avant son intégration en utilisant RENAMEID afin de maintenir l'ordre souhaité

respondant dans l'époque *enfant* : $i_1^{A1}\, i_0^{B0}\, m_0^{B1}$ (ligne 21). En réassignant cet identifiant à l'élément inséré de manière concurrente, le noeud A peut l'insérer à son état tout en préservant l'ordre souhaité.

RENAMEID permet aussi aux noeuds de gérer le cas contraire : intégrer des opérations *rename* distantes sur leur copie locale alors qu'ils ont précédemment intégré des modifications concurrentes. Ce cas correspond à celui du noeud B dans la Figure 2.3. À la réception de l'opération *rename* du noeud A, le noeud B utilise RENAMEID sur chacun des identifiants de son état pour le renommer et atteindre un état équivalent à celui du noeud A.

L'Algorithme 2 présente seulement le cas principal de RENAMEID, c.-à-d. le cas où l'identifiant à renommer appartient à l'intervalle des identifiants formant l'ancien état ($firstId \leq_{id} id \leq_{id} lastId$). Les fonctions pour gérer les autres cas, c.-à-d. les cas où l'identifiant à renommer n'appartient pas à cet intervalle ($id <_{id} firstId$ ou $lastId <_{id} id$), sont présentées dans l'**??**.

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

2.1.3 Évolution du modèle de livraison des opérations

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

Dans la Figure 2.4, les noeuds A et B répliquent tous deux une même séquence, contenant les éléments "ABCD". Tout d'abord, le noeud A procède au renommage de cet état.

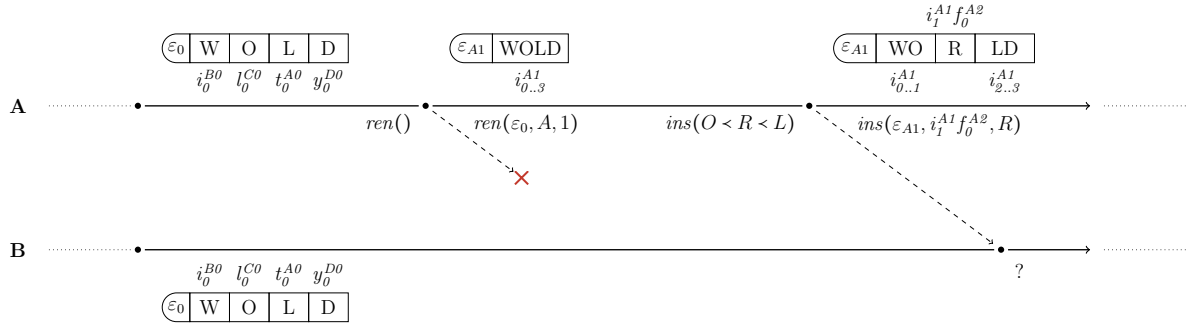


FIGURE 2.4 – Livraison d'une opération *insert* sans avoir reçu l'opération *rename* précédente

Puis il insère un nouvel élément, "X", entre "B" et "C". Les opérations correspondantes aux actions du noeud A sont diffusées sur le réseau.

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

Comme nous l'avions déjà illustré par la Figure 2.2, les identifiants d'une époque ne peuvent être comparés qu'aux identifiants de la même époque. Tenter d'intégrer une opération *insert* ou *remove* provenant d'une époque encore inconnue ne résulterait qu'en un état incohérent et une transgression de l'intention utilisateur (cf. Propriété 2, page 58). Il est donc nécessaire d'empêcher ce scénario de se produire.

Pour cela, nous proposons de faire évoluer le modèle de livraison des opérations de RenamableLogootSplit. Celui-ci repose sur celui de LogootSplit (cf. Définition 38, page 49). Pour rappel, ce modèle requiert que

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

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

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

Pour prévenir les scénarios tels que celui illustré par la Figure 2.4 nous y ajoutons la règle suivante : les opérations *rename* doivent être livrées à la structure de données avant les opérations qui ont une dépendance causale vers ces dernières. Nous obtenons donc le modèle de livraison suivant :

Définition 43 (Modèle de livraison RenamableLogootSplit)**.** Le modèle de livraison RenamableLogootSplit définit les 4 règles suivantes sur la livraison des opérations :

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

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

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

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

Il est cependant intéressant de noter que la livraison de l'opération *rename* ne requiert pas de contraintes supplémentaires. Notamment, une opération *rename* peut être livrée dans le désordre par rapport aux opérations *insert* et *remove* dont elle dépend causalement. La Figure 2.5 présente un exemple de ce cas figure.

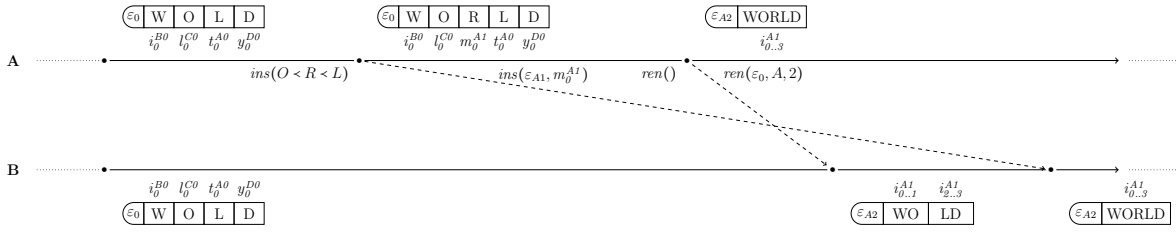


FIGURE 2.5 – Livraison désordonnée d'une opération *rename* et de l'opération *insert* qui la précède

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

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

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

2.2 Gestion des opérations de renommage concurrentes

2.2.1 Conflits en cas de renommages concurrents

Nous considérons à présent les scénarios avec des opérations *rename* concurrentes. Figure 2.6 développe le scénario décrit précédemment dans la Figure 2.3.

Après avoir diffusé son opération *insert*, le noeud B effectue une opération *rename* sur son état. Cette opération réassigne à chaque élément un nouvel identifiant à partir de l'identifiant du premier élément de la séquence (i_0^{B0}), de l'identifiant du noeud (**B**) et de son numéro de séquence courant (2). Cette opération introduit aussi une nouvelle époque : ε_{B2}. Puisque l'opération *rename* de A n'a pas encore été livrée au noeud B à ce moment, les deux opérations *rename* sont concurrentes.

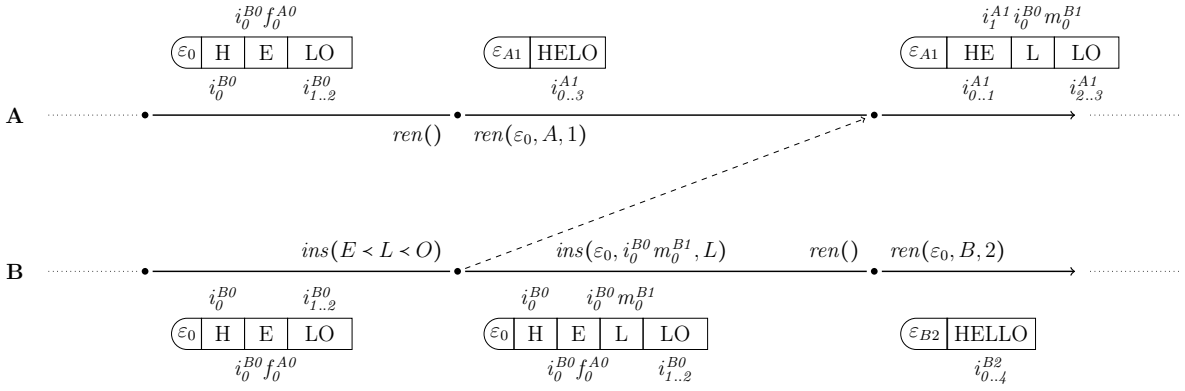


FIGURE 2.6 – Opérations *rename* concurrentes menant à des états divergents

Puisque des époques concurrentes sont générées, les époques forment désormais un *arbre des époques*. Nous représentons dans la Figure 2.7 l'*arbre des époques* que les noeuds obtiennent une fois qu'ils se sont synchronisés à terme. Les époques sont representées sous la forme de noeuds de l'arbre et la relation *parent-enfant* entre elles est illustrée sous la forme de flèches noires.

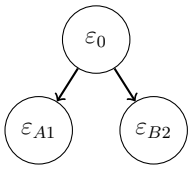


FIGURE 2.7 – *Arbre des époques* correspondant au scénario décrit dans la Figure 2.6

À l'issue du scénario décrit dans la Figure 2.6, les noeuds A et B sont respectivement aux époques ε_{A1} et ε_{B2}. Pour converger, tous les noeuds devraient atteindre la même époque à terme. Cependant, la fonction RENAMEID décrite dans l'Algorithme 2 permet seulement aux noeuds de progresser d'une époque *parente* à une de ses époques *enfants*. Le noeud A (resp. B) est donc dans l'incapacité de progresser vers l'époque du noeud B (resp. A). Il est donc nécessaire de faire évoluer notre mécanisme de renommage pour sortir de cette impasse.

Tout d'abord, les noeuds doivent se mettre d'accord sur une époque commune de l'*arbre des époques* comme époque cible. Afin d'éviter des problèmes de performance dûs à une coordination synchrone, les noeuds doivent sélectionner cette époque de manière non-coordonnée, c.-à-d. en utilisant seulement les données présentes dans l'*arbre des époques*. Nous présentons un tel mécanisme dans la sous-section 2.2.2.

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

revertRenameId, qui remplit cet objectif dans la sous-section 2.2.3.

2.2.2 Relation de priorité entre renommages

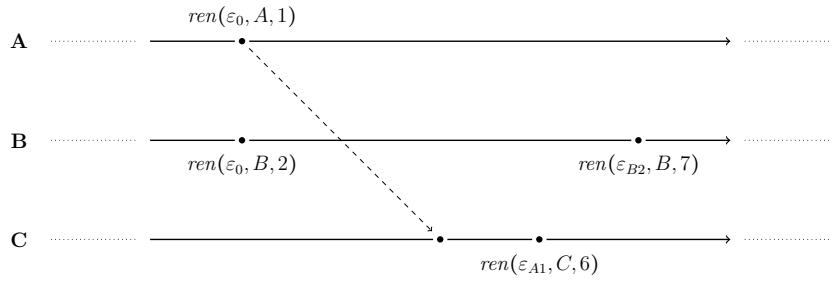
Pour que chaque noeud sélectionne la même époque cible de manière non-coordonnée, nous définissons la relation de priorité sur les époques $<_\varepsilon$.

Définition 44 (Relation *priority* $<_\varepsilon$)**.** La relation *priority* $<_\varepsilon$ est un ordre strict total sur l'ensemble des époques. Elle permet aux noeuds de comparer n'importe quelle paire d'époques.

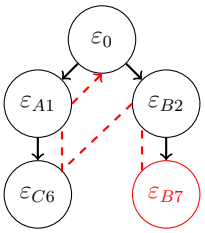
En utilisant la relation *priority*, nous définissons l'époque cible de la manière suivante :

Définition 45 (Époque cible)**.** L'époque cible est l'époque de l'ensemble des époques vers laquelle les noeuds doivent progresser. Les noeuds sélectionnent comme époque cible l'époque maximale d'après l'ordre établi par *priority*.

Pour définir la relation *priority*, nous pouvons choisir entre plusieurs stratégies. Dans le cadre de ce travail, nous utilisons l'ordre lexicographique sur le chemin des époques dans l'*arbre des époques*. La Figure 2.8 fournit un exemple.



(a) Exécution d'opérations *rename* concurrentes



(b) *Arbre des époques* final correspondant avec la relation *priority* illustrée

Figure 2.8 – Sélectionner l'époque cible d'une exécution d'opérations *rename* concurrentes

La Figure 2.8a décrit une exécution dans laquelle trois noeuds A, B et C génèrent plusieurs opérations avant de se synchroniser à terme. Comme seules les opérations *rename* sont pertinentes pour le problème qui nous occupe, nous représentons seulement ces opérations dans cette figure. Initialement, le noeud A génère une opération *rename* qui introduit l'époque ε_{A1}. Cette opération est livrée au noeud C, qui génère ensuite sa propre opération

rename qui introduit l'époque ε_{C6}. De manière concurrente à ces opérations, le noeud B génère deux opérations *rename*, introduisant ε_{B2} et ε_{B7}.

Une fois que les noeuds se sont synchronisés, ils obtiennent l'*arbre des époques* représenté dans la Figure 2.8b. Dans cette figure, la flèche tireté rouge représente l'ordre entre les époques d'après la relation *priority* tandis que l'époque cible choisie est représentée sous la forme d'un noeud rouge.

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

D'autres stratégies pourraient être proposées pour définir la relation *priority*. Par exemple, l'ordre proposé *priority* pourrait se baser sur une représentation du travail effectué sur le document, à l'aide de métriques additionnelles intégrées au sein des opérations *rename*. Cela permettrait de favoriser la branche de l'*arbre des époques* avec le plus d'activité, que nous conjecturons corrélé avec le nombre de noeuds actifs, pour minimiser la quantité globale de calculs effectués par les noeuds du système. Nous approfondissons ce sujet dans la sous-section 2.5.4.

2.2.3 Algorithme d'annulation de l'opération de renommage

À présent, nous développons le scénario présenté dans la Figure 2.6. Dans la Figure 2.9, le noeud A reçoit l'opération *rename* du noeud B. Cette opération est concurrente à l'opération *rename* que le noeud A a appliqué précédemment. D'après la relation *priority* proposée, le noeud A sélectionne l'époque introduite ε_{B2} comme l'époque cible ($\varepsilon_{A1} <_\varepsilon \varepsilon_{B2}$). Mais pour pouvoir renommer son état vers l'époque ε_{B2}, il doit au préalable faire revenir son état courant de l'époque ε_{A1} à un état équivalent à l'époque ε_0. Nous devons définir un mécanisme permettant aux noeuds d'annuler les effets d'une opération *rename* appliquée précédemment.

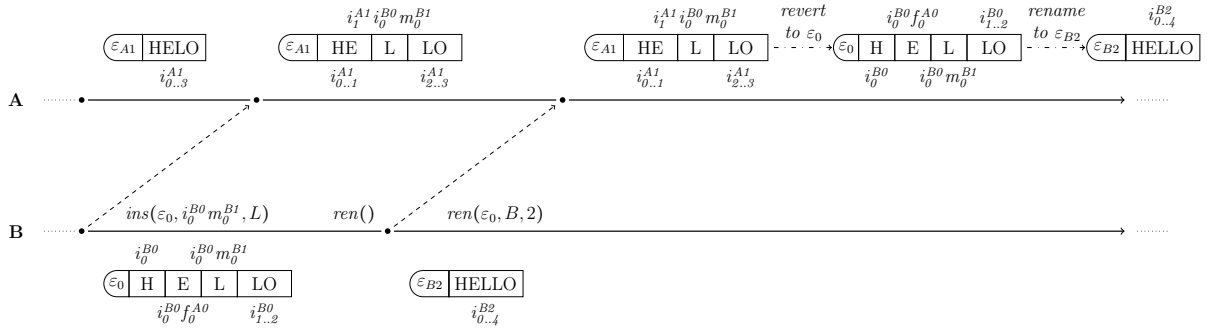


FIGURE 2.9 – Annulation d'une opération *rename* intégrée précèdemment en présence d'un identifiant inséré en concurrence

C'est précisément le but de REVERTRENAMEID, qui associe les identifiants de l'époque *enfant* aux identifiants correspondant dans l'époque *parente*. Nous décrivons cette fonction dans l'Algorithme 3.

Les objectifs de REVERTRENAMEID sont les suivants :

Algorithme 3 Fonctions principales pour annuler le renommage appliqué précèdemment à un identifiant

```
1: function revertRenameId(id ∈ 𝕀, renamedIds ∈ Array⟨𝕀⟩, nodeId ∈ ℕ, nodeSeq ∈ ℕ) : 𝕀
                                          ▷ renamedIds = [id_0, id_1, ···, id_{n-2}, id_{n-1}]
2:     firstId ← id_0
3:     lastId ← id_{n-1}
                                                  ▷ firstId = ⟨pos, _, _, _⟩ ⊕ suffix
4:     newFirstId ← ⟨pos, nodeId, nodeSeq, 0⟩
5:     newLastId ← ⟨pos, nodeId, nodeSeq, n − 1⟩
6:     if id <_id newFirstId then
7:         return revRenIdLessThanNewFirstId(id, firstId, newFirstId)
8:     else if id = ⟨pos, nodeId, nodeSeq, i⟩ then
                                          ▷ id obtained through Algorithme 2, ligne 9
9:         return id_i
10:    else if newLastId <_id id then
11:        return revRenIdGreaterThanNewLastId(id, lastId)
12:    else
                                          ▷ id = ⟨pos, nodeId, nodeSeq, i⟩ ⊕ suffix
13:        return revRenIdfromPredId(id, renamedIds, i)
14:    end if
15: end function
16:
17: function revRenIdfromPredId(id ∈ 𝕀, renamedIds ∈ Array⟨𝕀⟩, index ∈ ℕ) : 𝕀
                                          ▷ renamedIds = [id_0, id_1, ···, id_{n-2}, id_{n-1}]
                                          ▷ id = ⟨pos, nodeId, nodeSeq, index⟩ ⊕ tail
18:    predId ← id_index
19:    succId ← id_{index+1}
20:    if tail <_id predId then
                                          ▷ id has been inserted causally after the rename op
21:        return predId ⊕ ⊥_t ⊕ tail                    ▷ with ⊥_t the minimal tuple
22:    else if succId <_id tail then
                                          ▷ id has been inserted causally after the rename op
                                          ▷ succId = prefix ⊕ ⟨pos_j, nodeId_j, nodeSeq_j, offset_j⟩
23:        predOfSuccId ← prefix ⊕ ⟨pos_j, nodeId_j, nodeSeq_j, offset_j − 1⟩
24:        return predOfSuccId ⊕ ⊤_t ⊕ tail             ▷ with ⊤_t the maximal tuple
25:    else
26:        return tail
27:    end if
28: end function
```

(i) Restaurer à leur ancienne valeur les identifiants générés causalement avant l'opération *rename* annulée.

(ii) Restaurer à leur ancienne valeur les identifiants générés de manière concurrente à l'opération *rename* annulée.

(iii) Assigner de nouveaux identifiants respectant l'ordre souhaité aux éléments qui ont été insérés causalement après l'opération *rename* annulée.

Le cas (i) est le plus trivial. Pour retrouver la valeur de *id* à partir de *newId* [19], revertRenameId utilise simplement la valeur de offset de *newId*. En effet, cette va-

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

leur correspond à l'index de *id* dans l'*ancien état* (c.-à-d. $renamedIds[offset] = id$). Par exemple, dans la Figure 2.9, l'identifiant i_0^{A1} a pour offset 0, revertRenameId renvoie donc $renamedIds[0] = i_0^{B0}$ (ligne 9).

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

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

et nous devons obtenir :

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

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

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

Dans cette situation, *newId* a été obtenu en utilisant renIdFromPredId et nous avons $id = tail$. Nous observons qu'en renvoyant *tail*, revertRenameId valident les deux contraintes, c.-à-d. préserver l'ordre souhaité et restaurer à sa valeur initiale l'identifiant. Pour illustrer ce cas, considérons l'identifiant $i_1^{A1} i_0^{B0} m_0^{B1}$ dans la Figure 2.9. Pour cet identifiant, nous avons :

— $newPredId = i_1^{A1}$, donc $predId = i_0^{B0} f_0^{A0}$ d'après le cas (i).

— $newSuccId = i_2^{A1}$, donc $succId = i_1^{B0}$ d'après le cas (i).

Nous avons donc bien :

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

et $tail = i_0^{B0} m_0^{B1}$. Renvoyer cette valeur (ligne 26) nous permet ainsi de conserver l'ordre entre les identifiants puisque :

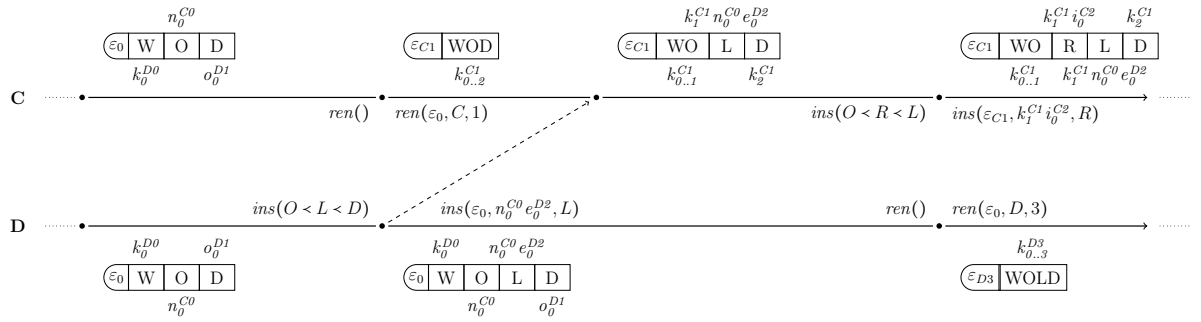
$$i_0^{B0} f_0^{A0} <_{id} i_0^{B0} m_0^{B1} <_{id} i_1^{B0}$$

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

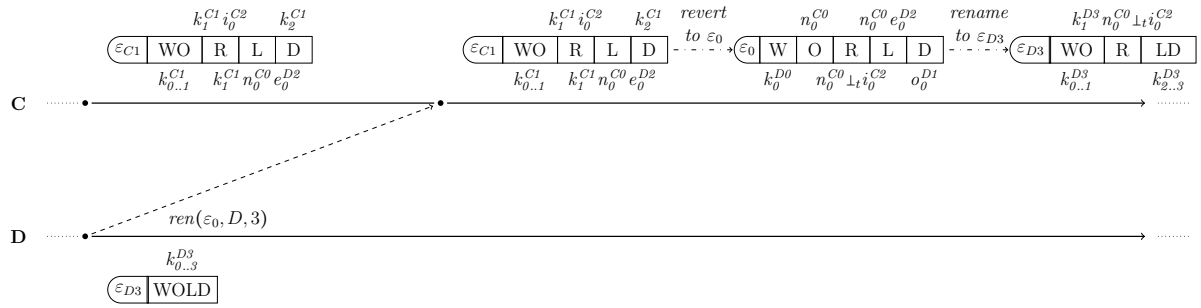
$$newId \in]newPredId, newPredId\ predId[$$

Puisque *newId* a été inséré causalement après l'opération *rename*, il n'existe pas de contrainte sur la valeur à retourner autre que la Propriété 3.2. Pour gérer ce cas, nous introduisons deux nouveaux tuples exclusifs au mécanisme de renommage : *minTuple* et *maxTuple*, notés respectivement $\bot_t$ et $\top_t$. Ils sont respectivement le tuple minimal et maximal utilisables pour générer des identifiants. En utilisant *minTuple*, revertRenameId est capable de renvoyer une valeur pour *id* adaptée à l'ordre souhaité (avec $id = predId \bot_t tail$, ligne 21). Nous justifions ce comportement à l'aide de la Figure 2.10.

Dans la Figure 2.10, les noeuds C et D répliquent une même séquence contenant les éléments "WOD". Dans la Figure 2.10a, le noeud C commence par renommer son état. En



(a) Génération d'une opération *insert* dépendante causalement d'une opération *rename*



(b) Annulation de l'opération *rename* précédente au profit d'une opération *rename* concurrente

FIGURE 2.10 – Annulation d'une opération *rename* intégrée précèdemment en présence d'un identifiant inséré causalement après

concurrence, le noeud D insère l'élément "L" entre les éléments "O" et "D". L'opération *insert* correspondante est livrée au noeud C, qui l'intègre en suivant le comportement défini en sous-section 2.1.2. Le noeud C procède ensuite à l'insertion de l'élément "R" entre les éléments "O" et "L". Cette insertion dépend donc causalement de l'opération *rename* effectuée précédemment par C. En parallèle, le noeud D effectue un renommage de son état. Cette opération *rename* est donc concurrente à l'opération *rename* générée précédemment par C.

Dans la Figure 2.10b, l'opération *rename* de D est livrée au noeud C. L'époque introduite par cette opération étant prioritaire par rapport à l'époque actuelle de C ($\varepsilon_{C1} <_{\varepsilon} \varepsilon_{D3}$), le noeud C procède à l'annulation de son opération *rename*.

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

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

D'après les règles présentées précédemment :

— k_1^{C1} est transformé en n_0^{C0} (cas (i), ligne 9).

— $k_1^{C1}\, n_0^{C0}\, e_0^{D2}$ est transformé en $n_0^{C0}\, e_0^{D2}$ (cas (ii), ligne 26).

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

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

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

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

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

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

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

Comme pour l'Algorithme 2, l'Algorithme 3 ne présente seulement que le cas principal de REVERTRENAMEID. Il s'agit du cas où l'identifiant à restaurer appartient à l'intervalle des identifiants renommés ($newFirstId \leq_{id} id \leq_{id} newLastId$). Les fonctions pour gérer les cas restants sont présentées dans l'**??**.

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

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

2.3 Mécanisme de Garbage Collection des anciens états obsolètes

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

Pour proposer un tel mécanisme, nous nous reposons sur la notion de *stabilité causale des opérations* [26]. Une opération est causalement stable une fois qu'elle a été livrée à tous les noeuds. Dans le contexte de l'opération *rename*, cela implique que tous les noeuds ont progressé à l'époque introduite par cette opération ou à une époque plus grande d'après la relation *priority*. À partir de ce constat, nous définissons les *potentielles époques courantes* :

Définition 46 (Potentielles époques courantes)**.** L'ensemble des époques auxquelles les noeuds peuvent se trouver actuellement et à partir desquelles ils peuvent émettre des opérations, du point de vue du noeud courant. Il s'agit d'un sous-ensemble de l'ensemble des époques, composé de l'époque maximale introduite par une opération *rename* causalement stable et de toutes les époques plus grande que cette dernière d'après la relation *priority*.

Les prochaines opérations livrées seront donc générées à partir de ces époques seulement [20]. Pour traiter ces prochaines opérations, les noeuds doivent maintenir les chemins entre toutes les époques de l'ensemble des *potentielles époques courantes*. Nous appelons *époques requises* l'ensemble des époques correspondant.

Définition 47 (Époques requises)**.** L'ensemble des époques qu'un noeud doit conserver pour traiter les potentielles prochaines opérations. Il s'agit de l'ensemble des époques qui forment les chemins entre chaque époque appartenant à l'ensemble des *potentielles époques courantes* et leur Plus Petit Ancêtre Commun (PPAC).

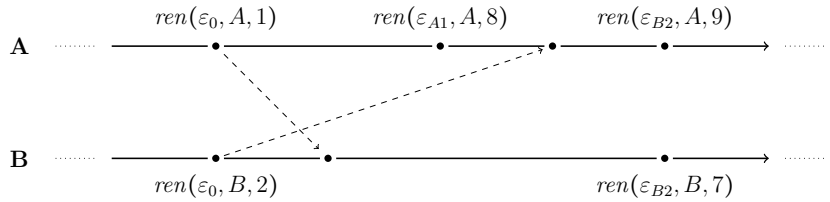
Il s'ensuit que toute époque qui n'appartient pas à l'ensemble des *époques requises* peut être retirée par les noeuds. La Figure 2.11 illustre un cas d'utilisation du mécanisme de récupération de mémoire proposé.

Dans la Figure 2.11a, nous représentons une exécution au cours de laquelle deux noeuds A et B génère respectivement plusieurs opérations *rename*. Dans la Figure 2.11b, nous représentons les *arbre des époques* respectifs de chaque noeud. Les époques introduites par des opérations *rename* causalement stables sont representées en utilisant des doubles cercles. L'ensemble des *potentielles époques courantes* est montré sous la forme d'un rectangle noir tireté, tandis que l'ensemble des *époques requises* est représenté par un rectangle vert pointillé.

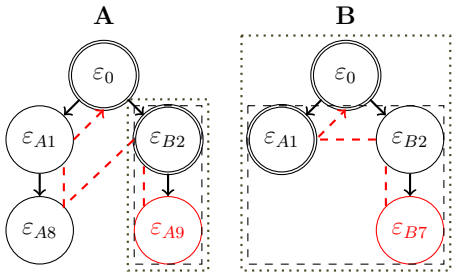
Le noeud A génère tout d'abord une opération *rename* vers ε_{A1} et ensuite une opération *rename* vers ε_{A8}. Il reçoit ensuite une opération *rename* du noeud B qui introduit ε_{B2}. Puisque ε_{B2} est plus grand que son époque courante actuelle ($\varepsilon_{e0}\varepsilon_{A1}\varepsilon_{A8}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

De manière concurrente, le noeud B génère l'opération *rename* vers ε_{B2}. Il reçoit ensuite l'opération *rename* vers ε_{A1} du noeud A. Cependant, le noeud B conserve ε_{B2}

20. Nous considérons ici que l'ordre de livraison des opérations satisfait le modèle de livraison FIFO, c.-à-d. que les messages d'un noeud sont livrés aux autres noeuds dans le même ordre qu'ils ont été envoyés, et le modèle de livraison RenamableLogootSplit (cf. Définition 43, page 64) Ainsi, une opération de renommage ne peut être causalement stable pour un noeud que si ce dernier a reçu toutes les opérations ayant eu lieu avant d'après la relation *happens-before* et toutes les opérations concurrentes.



(a) Exécution d'opérations *rename*



(b) *Arbres des époques* respectifs avec les ensembles *potentielles époques courantes* et *époques requises* illustrés

FIGURE 2.11 – Suppression des époques obsolètes et récupération de la mémoire des *anciens états* associés

comme époque courante (puisque $\varepsilon_{e0}\varepsilon_{A1} < \varepsilon_{e0}\varepsilon_{B2}$). Après, le noeud B génère une autre opération *rename* vers ε_{B7}.

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

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

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

Notons que le mécanisme de récupération de mémoire peut être simplifié dans les systèmes empêchant les opérations *rename* concurrentes. Puisque les époques forment

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

Pour déterminer qu'une opération *rename* donnée est causalement stable, les noeuds doivent être conscients des autres et de leur avancement. Un protocole de gestion de groupe tel que [64, 65] est donc requis.

La stabilité causale peut prendre un certain temps à être atteinte. En attendant, les noeuds peuvent néanmoins décharger les anciens états sur le disque dur puisqu'ils ne sont seulement nécessaires que pour traiter les opérations concurrentes aux opérations *rename*. Nous approfondissons ce sujet dans la sous-section 2.5.2.

2.4 Validation

Dans cette section, nous présentons notre validation de notre contribution.

Cette validation prend deux formes. Dans un premier temps, nous effectuons une évaluation théorique de ses performances. Ainsi, nous présentons dans la sous-section 2.4.1 une évaluation de la complexité en temps des opérations de RenamableLogootSplit, ainsi que de son mécanisme de GC des métadonnées des anciens états obsolètes.

Puis nous effectuons dans un second temps une évaluation empirique pour confirmer ces résultats. Dans la sous-section 2.4.2, nous présentons le protocole expérimental que nous avons mis en place pour cette évaluation empirique. Puis nous présentons et analysons les résultats obtenus dans la sous-section 2.4.3.

2.4.1 Complexité en temps des opérations

Hypothèses

Les complexités en temps des opérations de RenamableLogootSplit dépendent de plusieurs paramètres, e.g. nombre d'identifiants et de blocs stockés au sein de la structure, taille des identifiants ou encore structures de données utilisées. Afin d'établir les valeurs de complexité des différentes opérations, nous prenons les hypothèses suivantes vis-à-vis des paramètres.

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

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

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

La complexité de l'intégration de l'opération *insert* locale est inchangée par rapport à celle obtenue pour LogootSplit. Son intégration consiste toujours à déterminer entre quels identifiants se situe les nouveaux éléments insérés, à générer de nouveaux identifiants correspondants à l'ordre souhaité puis à insérer le bloc dans l'arbre AVL. D'après [51], nous obtenons donc une complexité de $\mathcal{O}(\log b)$ pour cette opération locale, où b représente le nombre de blocs dans la séquence.

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

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

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

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

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

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

l'arbre AVL.

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

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

Complexité en temps de l'opération *rename*

L'opération *rename* locale se décompose en 2 étapes :

(i) La génération de l'*ancien état* à intégrer au message de l'opération (cf. Définition 41, page 60).

(ii) Le remplacement de la séquence courante par une séquence équivalente, renommée.

La première étape consiste à parcourir la séquence actuelle pour en extraire les intervalles d'identifiants. Elle s'effectue donc en $\mathcal{O}(b)$. La seconde étape consiste à instancier une nouvelle séquence vide, et à y insérer un bloc qui associe le contenu actuel de la séquence à l'intervalle d'identifiants $pos_{0..n-1}^{nodeId\ \ nodeSeq}$, avec *pos* la position du premier tuple du premier id de l'état, *nodeId* et *nodeSeq* l'identifiant et le numéro de séquence actuel du noeud et n la taille du contenu. Cette seconde étape s'effectue en $\mathcal{O}(1)$. L'opération *rename* locale a donc une complexité de $\mathcal{O}(b)$.

L'intégration de l'opération *rename* se décompose en les étapes suivantes :

(i) L'insertion de la nouvelle époque et de l'*ancien état* associé dans l'*arbre des époques*.

(ii) La récupération des n identifiants formant l'état courant.

(iii) Le calcul de l'époque PPAC entre l'époque courante et l'époque cible.

(iv) L'identification des k opérations *rename* à inverser et des l opérations *rename* à jouer.

(v) Le renommage de chacun des identifiants à l'aide de revertRenameId et renameId.

(vi) L'insertion de chacun des identifiants renommés dans une nouvelle séquence.

L'*arbre des époques* étant représenté à l'aide d'une table de hachage, la première étape s'effectue en $\mathcal{O}(1)$. La seconde étape nécessite elle de parcourir l'arbre AVL et de convertir chaque intervalle d'identifiants en liste d'identifiants, ce qui nécessite $\mathcal{O}(n)$ instructions.

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

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

Nous pouvons néanmoins améliorer ce premier résultat. Notamment, nous pouvons tirer parti des faits suivants :

(i) Le fonctionnement de renameId repose sur l'utilisation de l'identifant prédecesseur comme préfixe.

(ii) Les identifiants de l'état courant et de l'*ancien état* forment tous deux des listes triées.

Ainsi, plutôt que d'effectuer une recherche dichotomique sur l'*ancien état* pour trouver le prédecesseur de l'identifiant à renommer, nous pouvons parcourir les deux listes en parallèle. Ceci nous permet de renommer l'intégralité des identifiants en un seul parcours de l'état courant et de l'*ancien état*, c.-à-d. en $\mathcal{O}(n)$ instructions. Ensuite, plutôt que d'insérer les identifiants un à un dans la nouvelle séquence, nous pouvons recomposer au préalable les différents blocs en parcourant la liste des identifiants et en les aggrégeant au fur et à mesure. Il ne reste plus qu'à constituer la nouvelle séquence à partir des blocs obtenus. Ces actions s'effectuent respectivement en $\mathcal{O}(n)$ et $\mathcal{O}(b)$ instructions.

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

Récapitulatif

Nous récapitulons les complexités en temps présentées précédemment dans le Tableau 2.1.

TABLE 2.1 – Complexité en temps des différentes opérations

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

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

Complexité en temps du mécanisme de GC des anciens états obsolètes

Pour compléter notre analyse théorique des performances de RenamableLogootSplit, nous proposons une analyse en complexité en temps du mécanisme présenté en section 2.3 qui permet de supprimer les époques devenues obsolètes et de récupérer la mémoire occupée par leur *ancien état* respectif.

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

La seconde étape consiste à parcourir l'arbre des époques de manière inverse à l'ordre défini par la relation *priority*. Ce parcours s'effectue jusqu'à trouver l'époque maximale

causalement stable, c.-à-d. la première époque pour laquelle l'opération *rename* associée est causalement stable. Pour chaque époque parcourue, le mécanisme de récupération de mémoire calcule et stocke son chemin jusqu'à la racine. Cette étape s'exécute donc en $\mathcal{O}(e \cdot h)$, avec e le nombre d'époques composant l'arbre des époques et h la hauteur de l'arbre.

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

L'algorithme peut alors calculer l'ensemble des *époques requises*. Pour cela, il parcourt les chemins calculés au cours de la seconde étape. Pour chaque chemin, il ajoute les époques se trouvant après l'époque PPAC à l'ensemble des *époques requises*. De nouveau, cette étape s'exécute en $\mathcal{O}(e \cdot h)$.

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

Ainsi, nous obtenons que la complexité en temps du mécanisme de récupération de mémoire des époques est en $\mathcal{O}(p^2+e \cdot h)$. Nous récapitulons ce résultat dans la Tableau 2.2.

TABLE 2.2 – Complexité en temps du mécanisme de GC des anciens états obsolètes

Étape	Temps
calculer le vecteur de versions des opérations causalement stables	$\mathcal{O}(p^2)$
calculer les chemins de la racine aux potentielles époques courantes	$\mathcal{O}(e \cdot h)$
identifier le PPAC	$\mathcal{O}(e \cdot h)$
calculer l'ensembe des époques requises	$\mathcal{O}(e \cdot h)$
supprimer les époques obsolètes	$\mathcal{O}(e)$
total	$\mathcal{O}(p^2 + e \cdot h)$

p : nombre de noeuds (ou pairs) du système, e : nombre d'époques dans l'*arbre des époques*, h : hauteur de l'*arbre des époques*

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

2.4.2 Expérimentations

Afin de valider l'approche que nous proposons, nous avons procédé à une évaluation expérimentale. Les objectifs de cette évaluation étaient de mesurer

(i) Le surcoût mémoire de la séquence répliquée.

(ii) Le surcoût en calculs ajouté aux opérations *insert* et *remove* par le mécanisme de renommage.

(iii) Le coût d'intégration des opérations *rename*.

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

Scénario d'expérimentation

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

Dans un premier temps, les pairs spécifient principalement le contenu de l'article. Quelques opérations *remove* sont tout même générées pour simuler des fautes de frappes. Une fois que le document atteint une taille critique (définie de manière arbitraire), les pairs passent à la seconde phase de la collaboration. Lors de cette seconde phase, les pairs arrêtent d'ajouter du nouveau contenu mais se concentre à la place sur la reformulation et l'amélioration du contenu existant. Ceci est simulé en équilibrant le ratio entre les opérations *insert* et *remove*.

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

Implémentation des simulations

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

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

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

De plus, dans le cas de RenamableLogootSplit, 1 à 4 agents sont désignés de façon arbitraire comme des *renaming bots* en fonction de la session. Le rôle des *renaming bots* est générer des opérations *rename* toutes les 7.5k ou toutes les 30k opérations qu'ils observent,

en fonction des paramètres de la simulation. Ces opérations *rename* sont générées de manière à assurer qu'elles soient concurrentes.

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

2.4.3 Résultats

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

Convergence

Nous avons tout d'abord vérifié la convergence de l'état des noeuds à l'issue des simulations. Pour chaque simulation, nous avons comparé l'état final de chaque noeud à l'aide de leur instantanés respectifs. Nous avons pu confirmer que les noeuds convergaient sans aucune autre forme de communication que les opérations, satisfaisant donc le modèle de la Cohérence forte à terme (SEC) [5].

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

Consommation mémoire

Nous avons ensuite procédé à l'évaluation de l'évolution de la consommation mémoire du document au cours des simulations, en fonction du CRDT utilisé et du nombre de *renaming bots*. Nous présentons les résultats obtenus dans la Figure 2.12.

Pour chaque graphique dans la Figure 2.12, nous représentons 4 données différentes. La ligne tiretée bleue correspond à la taille du contenu du document, c.-à-d. du texte, tandis que la ligne continue rouge représente la taille complète du document LogootSplit.

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

La ligne pointillée orange représente la taille du document RenamableLogootSplit dans son pire cas. Dans ce scénario, les noeuds considèrent que les opérations *rename* ne deviennent jamais causalement stables. Les noeuds doivent alors conserver de façon permanente les métadonnées introduites par le mécanisme de renommage. Les performances de RenamableLogootSplit diminuent donc au fur et à mesure que le nombre de *renaming*

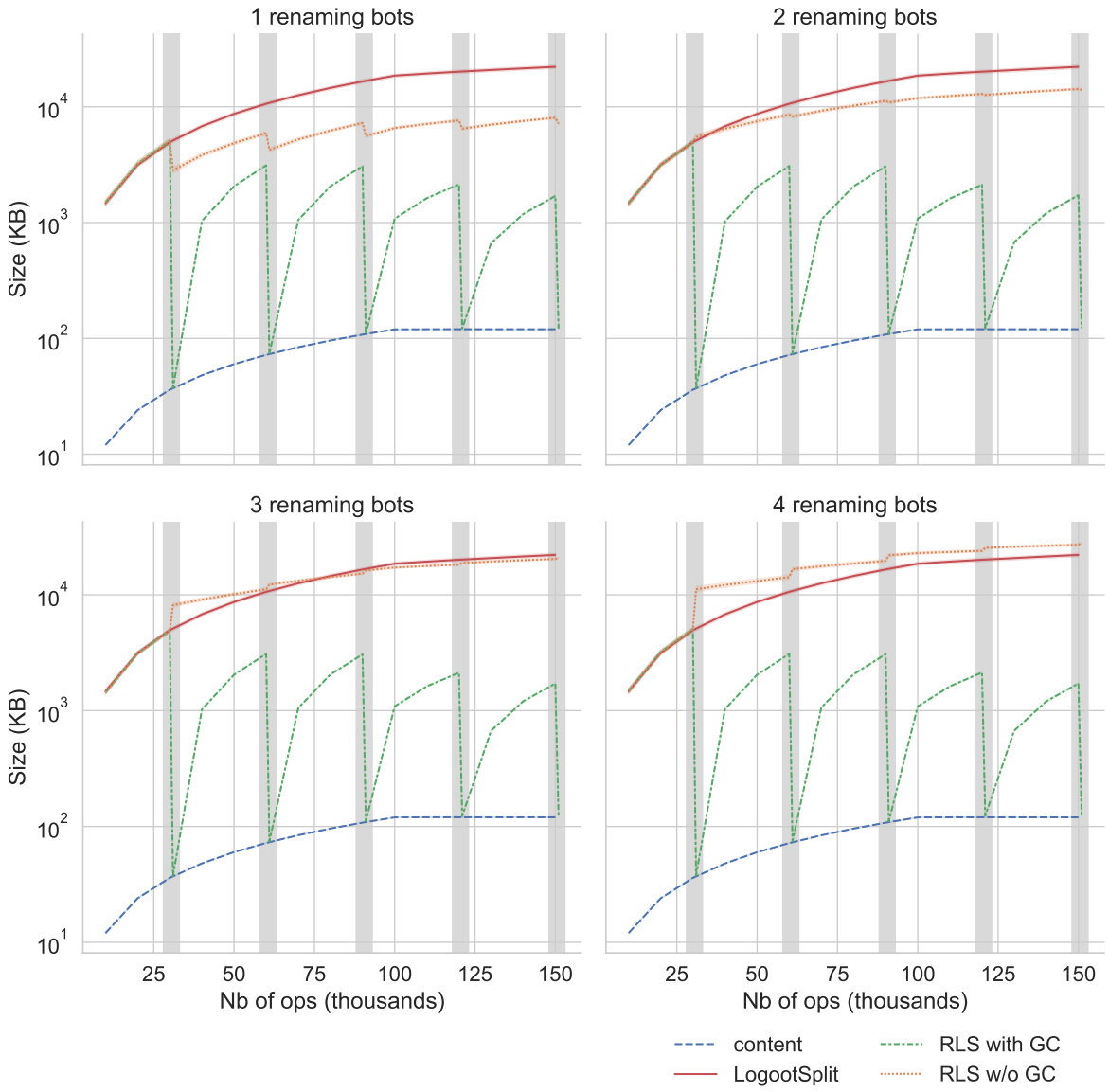


FIGURE 2.12 – Évolution de la taille du document en fonction du CRDT utilisé et du nombre de *renaming bots* dans la collaboration

bots et d'opérations *rename* générées augmente. Néanmoins, même dans ces conditions, nous observons que RenamableLogootSplit offre de meilleures performances que LogootSplit tant que le nombre de *renaming bots* reste faible (1 ou 2). Ce résultat s'explique par le fait que le mécanisme de renommage permet aux noeuds de supprimer les métadonnées de la structure de données utilisée en interne pour représenter la séquence (c.-à-d. l'arbre AVL).

Pour récapituler les résultats présentés, le mécanisme de renommage introduit un surcoût temporaire en métadonnées qui augmente avec chaque opération *rename*. Mais ce surcoût se résorbe à terme une fois que le système devient quiescent et que les opérations

rename deviennent causalement stables. Dans la sous-section 2.5.2, nous détaillerons l'idée que les *anciens états* peuvent être déchargés sur le disque en attendant que la stabilité causale soit atteinte pour atténuer l'impact du surcoût temporaire en métadonnées.

Temps d'intégration des opérations d'insertion et de suppression

Nous avons ensuite comparé l'évolution du temps d'intégration des opérations standards, c.-à-d. les opérations *insert* et *remove*, sur des documents LogootSplit et RenamableLogootSplit. Puisque les deux types d'opérations partagent la même complexité en temps, nous avons seulement utilisé des opérations *insert* dans nos benchmarks. Nous faisons par contre la différence entre les mises à jours *locales* et *distantes*. Conceptuellement, les modifications locales peuvent être décomposées comme présenté dans [13] en les deux étapes suivantes :

(i) La génération de l'opération correspondante.

(ii) L'application de l'opération correspondante sur l'état local.

Cependant, pour des raisons de performances, nous avons fusionné ces deux étapes dans notre implémentation. Nous distinguons donc les résultats des modifications *locales* et des modifications *distantes* dans nos benchmarks. La Figure 2.13 présente les résultats obtenus.

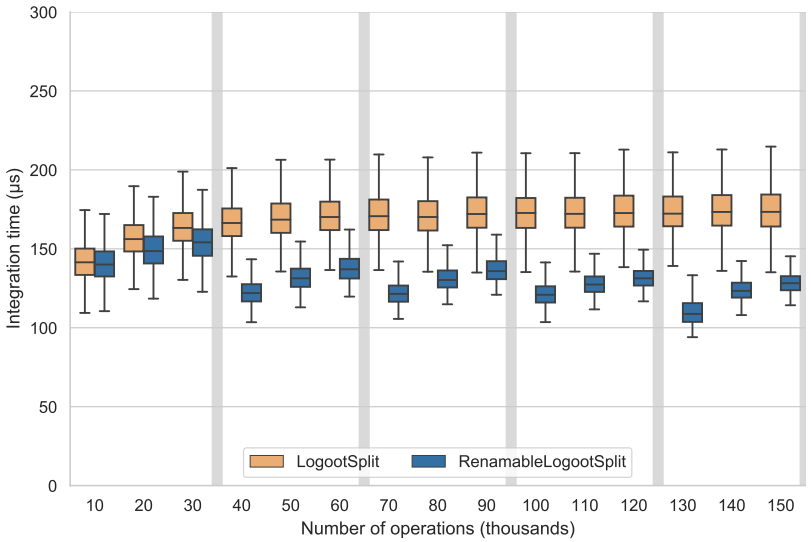
Dans ces figures, les boxplots oranges correspondent aux temps d'intégration sur des documents LogootSplit, les boxplots bleues sur des documents RenamableLogootSplit. Bien que les temps d'intégration soient initialement équivalents, les temps d'intégration sur des documents RenamableLogootSplit sont ensuite réduits par rapport à ceux de LogootSplit une fois que des opérations *rename* ont été intégrées. Cette amélioration s'explique par le fait que l'opération *rename* optimise la représentation interne de la séquence (c.-à-d. elle réduit le nombre de blocs stockés dans l'arbre AVL).

Dans le cadre des opérations distantes, nous avons mesuré des temps d'intégration spécifiques à RenamableLogootSplit : le temps d'intégration d'opérations distantes provenant d'époques *parentes* et d'époques *soeurs*, respectivement affiché sous la forme de boxplots blanches et rouges dans la Figure 2.13b.

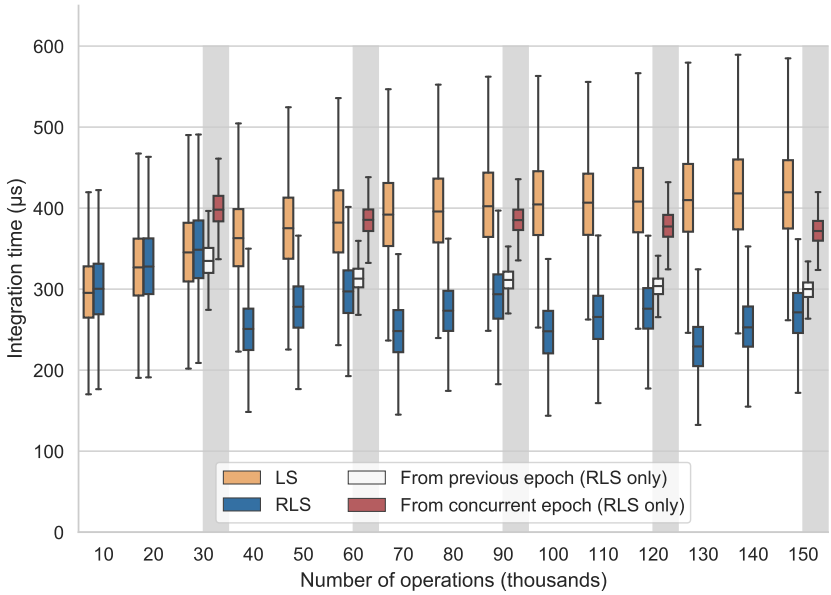
Les opérations distantes provenant d'époques *parentes* sont des opérations générées de manière concurrente à l'opération *rename* mais appliquées après cette dernière. Puisque l'opération doit être transformée au préalable en utilisant RENAMEID, nous observons un surcoût computationnel par rapport aux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

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

Pour récapituler, les fonctions de transformation ajoutent un surcoût aux temps d'intégration des opérations concurrentes aux opérations *rename*. Malgré ce surcoût, RenamableLogootSplit offre de meilleures performances que LogootSplit pour intégrer ces



(a) Modifications locales



(b) Modifications distantes

FIGURE 2.13 – Temps d'intégration des opérations standards

opérations grâce aux réductions de la taille de l'état effectuées par les opérations *rename*. Cependant, cette affirmation n'est vraie que tant que la distance entre l'époque de génération de l'opération et l'époque courante du noeud reste limitée, puisque les performances de RenamableLogootSplit dépendent linéairement de cette dernière (cf. Tableau 2.1, page 78). Néanmoins, ce surcoût ne concerne que les opérations concurrentes aux opérations *rename*. Il ne concerne pas la majorité des opérations, c.-à-d. les opérations générées entre deux séries d'opérations *rename*. Ces opérations, elles, ne souffrent d'aucun surcoût tout en bénéficiant des réductions de taille de l'état.

Temps d'intégration de l'opération de renommage

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

Nous rappellons que le traitement d'une opération *rename* dépend de l'ordre défini par la relation *priority* entre l'époque qu'elle introduit et l'époque courante du noeud qui intègre l'opération. Le cas des opérations *rename* distantes se décompose donc en trois catégories. Les opérations *distantes directes* désignent les opérations *rename* distantes qui introduisent une nouvelle époque *enfant* de l'époque courante du noeud. Les opérations *concurrentes introduisant une époque plus* (resp. *moins*) *prioritaire* désignent les opérations *rename* qui introduisent une époque *soeur* de l'époque courante du noeud. D'après la relation *priority*, l'époque introduite est plus (resp. moins) prioritaire que l'époque courante du noeud. Les résultats obtenus sont présentés dans le Tableau 2.3.

TABLE 2.3 – Temps d'intégration de l'opération *rename*

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

Le principal résultat de ces mesures est que les opérations *rename* sont particulièrement coûteuses quand comparées aux autres types d'opérations. Les opérations *rename* locales s'intègrent en centaines de millisecondes tandis que les opérations *distantes directes* et *concurrentes introduisant une plus grande époque* s'intègrent en secondes lorsque la taille du document dépasse 40k éléments. Ces résultats s'expliquent facilement par la complexité en temps de l'opération *rename* qui dépend supra-linéairement du nombre de blocs et d'éléments stockés dans l'état (cf. Tableau 2.1, page 78). Il est donc nécessaire de prendre en compte ce résultat et de

(i) Concevoir des stratégies de génération des opérations *rename* pour éviter d'impacter négativement l'expérience utilisateur.

(ii) Proposer des versions améliorées des algorithmes RENAMEID et REVERTRENAMEID pour réduire ces temps d'intégration.

Nous identifions plusieurs pistes d'amélioration de RENAMEID et REVERTRENAMEID :

— Au lieu d'utiliser RENAMEID, qui renomme l'état identifiant par identifiant, nous pourrions définir et utiliser RENAMEBLOCK. Cette fonction permettrait de renommer l'état bloc par bloc, offrant ainsi une meilleur complexité en temps. De plus, puisque les opérations *rename* fusionnent les blocs existants en un unique bloc, RENAMEBLOCK permettrait de mettre en place un cercle vertueux où chaque opération *rename* réduirait le temps d'exécution de la suivante.

— Puisque chaque appel à REVERTRENAMEID et REVERTRENAMEID est indépendant des autres, ces fonctions sont adaptées à la programmation parallèle. Au lieu de renommer les identifiants (ou blocs) de manière séquentielle, nous pourrions diviser la séquence en plusieurs parties et les renommer en parallèle.

Un autre résultat intéressant de ces benchmarks est que les opérations *concurrentes introduisant une plus petite époque* sont rapides à intégrer. Puisque ces opérations introduisent une époque qui n'est pas sélectionnée comme nouvelle époque cible, les noeuds ne procèdent pas au renommage de leur état. L'intégration des opérations *concurrentes introduisant une plus petite époque* consiste simplement à ajouter l'époque introduite et l'*ancien état* correspondant à l'*arbre des époques*. Les noeuds peuvent donc réduire de manière significative le coût d'intégration d'un ensemble d'opérations *rename* concurrentes en les appliquant dans l'ordre le plus adapté en fonction du contexte. Nous développons ce sujet dans la sous-section 2.5.5.

Temps pour rejouer le journal des opérations

Afin de comparer les performances de RenamableLogootSplit et de LogootSplit de manière globale, nous avons mesuré le temps nécessaire pour un nouveau noeud pour rejouer l'entièreté du journal des opérations d'une session de collaboration, en fonction du nombre de *renaming bots* de la session. Nous présentons les résultats obtenus dans la Figure 2.14.

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

Dans un cas réel d'utilisation, ce scénario (c.-à-d. rejouer l'entièreté du log) ne correspond pas au scénario principal et peut être mitigé, par exemple en utilisant un mécanisme de compression du journal des opérations. Dans la sous-section 2.5.6, nous présentons comment mettre en place un tel mécanisme en se basant justement sur les possibilités offertes par l'opération *rename*.

Impact de la fréquence de l'opération de renommage sur les performances

Pour évaluer l'impact de la fréquence de l'opération *rename* sur les performances, nous avons réalisé un benchmark supplémentaire. Ce benchmark consiste à rejouer les

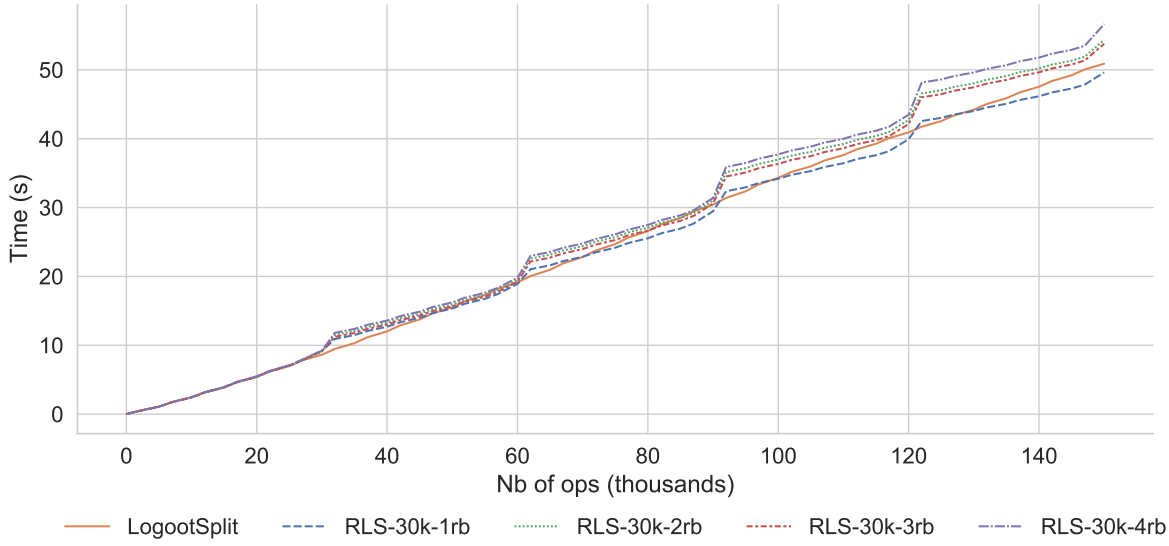


FIGURE 2.14 – Progression du nombre d'opérations du journal rejouées en fonction du temps

logs d'opérations des simulations en utilisant divers CRDTs et configurations : LogootSplit, RenamableLogootSplit effectuant des opérations *rename* toutes les 30k opérations, RenamableLogootSplit effectuant des opérations *rename* toutes les 7.5k opérations. Au fur et à mesure que le benchmark rejoue le journal des opérations, il mesure le temps d'intégration des opérations ainsi que leur taille. Nous présentons les résultats obtenus dans le Tableau 2.4 et le Tableau 2.5.

Paramètres		Temps d'intégration (μs)				
Type	CRDT	Moyenne	Médiane	IQR	1er Percent.	99ème Percent.
insert	LS	471	460	130	224	768
	RLS - 30k	397	323	66.7	171	587
	RLS - 7.5k	393	265	54.5	133	381
remove	LS	280	270	71.4	140	435
	RLS - 30k	247	181	39	97.9	308
	RLS - 7.5k	296	151	34.8	74.9	214
Paramètres		Temps d'intégration (ms)				
Type	CRDT	Moyenne	Médiane	IQR	1er Percent.	99ème Percent.
rename	RLS - 30k	1022	1188	425	540	1276
	RLS - 7.5k	861	974	669	123	1445

TABLE 2.4 – Temps d'intégration par type et par fréquence d'opérations *rename*

D'après les résultats du Tableau 2.4, nous observons que des opérations *rename* plus fréquentes permettent d'améliorer les temps d'intégration des opérations *insert* et *remove*. Cela confirme les résultats attendus puisque l'opération *rename* réduit la taille des iden-

tifiants de la structure ainsi que le nombre de blocs composant la séquence.

Nous remarquons aussi que la fréquence n'a aucun impact significatif sur le temps d'intégration des opérations *rename*. Il s'agit là aussi d'un résultat attendu puisque la complexité en temps de l'implémentation de l'opération *rename* dépend du nombre d'éléments dans la séquence, un facteur qui n'est pas impacté par les opérations *rename*.

Paramètres		Taille (o)				
Type	CRDT	Moyenne	Médiane	IQR	1er Percent.	99ème Percent.
insert	LS	593	584	184	216	1136
	RLS - 30k	442	378	92	314	958
	RLS - 7.5k	389	378	0	314	590
remove	LS	632	618	184	250	1170
	RLS - 30k	434	412	0	320	900
	RLS - 7.5k	401	412	0	320	596
Paramètres		Taille (Ko)				
Type	CRDT	Moyenne	Médiane	IQR	1er Percent.	99ème Percent.
rename	RLS - 30k	1366	1258	514	635	3373
	RLS - 7.5k	273	302	132	159	542

TABLE 2.5 – Taille des opérations par type et par fréquence d'opérations *rename*

Le Tableau 2.5 permet d'observer que les opérations *insert* et *remove* de RenamableLogootSplit sont initialement plus lourdes que les opérations correspondantes de LogootSplit. Ce résultat s'explique de part le fait qu'elles intègrent leur époque de génération comme donnée additionnelle. Mais alors que la taille des opérations de LogootSplit augmentent indéfiniment, celle des opérations de RenamableLogootSplit est bornée. La valeur de cette borne est définie par la fréquence de l'opération *rename*. Cela permet à RenamableLogootSplit d'atteindre un coût moindre par opération.

D'un autre côté, le coût des opérations *rename* est bien plus important (1000x) que celui des autres types d'opérations. Ceci s'explique par le fait que l'opération *rename* intègre l'*ancien état*, c.-à-d. la liste de tous les blocs composant l'état de la séquence au moment de la génération de l'opération. Cependant, nous observons le même phénomène pour les opérations *rename* que pour les autres opérations : la fréquence des opérations *rename* permet d'établir une borne pour la taille des opérations *rename*. Nous pouvons donc choisir d'émettre fréquemment des opérations *rename* pour limiter leur taille respective. Ceci implique néanmoins un surcoût en calculs pour chaque opération *rename* dans l'implémentation actuelle. Nous présentons une autre approche possible pour limiter la taille des opérations *rename* dans la sous-section 2.5.3. Cette approche consiste à implémenter un mécanisme de compression pour les opérations *rename* pour ne transmettre que les composants nécessaires à l'identifiant de chaque bloc de l'*ancien état*.

2.5 Discussion

2.5.1 Stratégie de génération des opérations de renommage

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

Plusieurs aspects doivent être pris en compte lors de la définition de la stratégie de génération des opérations *rename*. Le premier porte sur la taille de la structure de données. Comme illustré dans la Figure 2.12, les métadonnées augmentent de manière progressive jusqu'à représenter 99% de la structure de données. En utilisant les opérations *rename*, les noeuds peuvent supprimer les métadonnées et ainsi réduire la taille de la structure à un niveau acceptable. Pour déterminer quand générer des opérations *rename*, les noeuds peuvent donc monitorer le nombre d'opérations effectuées depuis la dernière opération *rename*, le nombre de blocs qui composent la séquence ou encore la taille des identifiants.

Un second aspect à prendre en compte est le temps d'intégration des opérations *rename*. Comme indiqué dans le Tableau 2.3, l'intégration des opérations *rename* distantes peut nécessiter des secondes si elles sont retardées trop longtemps. Bien que les opérations *rename* travaillent en coulisses, elles peuvent néanmoins impacter négativement l'expérience utilisateur. Notamment, les noeuds ne peuvent pas intégrer d'autres opérations *distantes* tant qu'ils sont en train de traiter des opérations *rename*. Du point de vue des utilisateurs, les opérations *rename* peuvent alors être perçues comme des pics de latence. Dans le domaine de l'édition collaborative temps réel, Ignat et al. [55, 56] ont montré que le délai dégradait la qualité des collaborations. Il est donc important de générer fréquemment des opérations *rename* pour conserver leur temps d'intégration sous une limite perceptible.

Finalement, le dernier aspect à considérer est le nombre d'opérations *rename* concurrentes. La Figure 2.12 montre que les opérations *rename* concurrentes accroissent la taille de la structure de données tandis que la Figure 2.14 illustre qu'elles augmentent le temps nécessaire pour rejouer le journal des opérations. La stratégie proposée doit donc viser à minimiser le nombre d'opérations *rename* concurrentes générées. Cependant, elle doit éviter d'utiliser des coordinations synchrones entre les noeuds pour cela (e.g. algorithmes de consensus), pour des raisons de performances. Pour réduire la probabilité de générer des opérations *rename* concurrentes, plusieurs méthodes peuvent être proposées. Par exemple, les noeuds peuvent monitorer à quels autres noeuds ils sont connectés actuellement et déléguer au noeud ayant le plus grand *identifiant de noeud* la responsabilité de générer les opérations *rename*.

Pour récapituler, nous pouvons proposer plusieurs stratégies de génération des opérations *rename*, pour minimiser de manière individuelle chacun des paramètres présentés. Mais bien que certaines de ces stratégies convergent (minimiser la taille de la structure de données et minimiser le temps d'intégration des opérations *rename*), d'autres entrent en conflit (générer une opération *rename* dès qu'un seuil est atteint vs. minimiser le nombre d'opérations *rename* concurrentes générées). Les concepteurs de systèmes doivent proposer un compromis entre les différents paramètres en fonction des contraintes du système

concerné (application temps réel ou asynchrone, limitations matérielles des noeuds...). Il est donc nécessaire d'analyser le système pour évaluer ses performances sur chaque aspect, ses usages et trouver le bon compromis entre tous les paramètres de la stratégie de renommage. Par exemple, dans le contexte des systèmes d'édition collaborative temps réel, [55] a montré que le délai diminue la qualité de la collaboration. Dans de tels systèmes, nous viserions donc à conserver le temps d'intégration des opérations (en incluant les opérations *rename*) en dessous du temps limite correspondant à leur perception par les utilisateurs.

2.5.2 Stockage des états précédents sur disque

Les noeuds doivent conserver les *anciens états* associés aux opérations *rename* pour transformer les opérations issues d'époques précédentes ou concurrentes. Les noeuds peuvent recevoir de telles opérations dans deux cas précis :

(i) Des noeuds ont émis récemment des opérations *rename*.

(ii) Des noeuds se sont récemment reconnectés.

Entre deux de ces évènements spécifiques, les *anciens états* ne sont pas nécessaires pour traiter les opérations.

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

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

2.5.3 Compression et limitation de la taille de l'opération de renommage

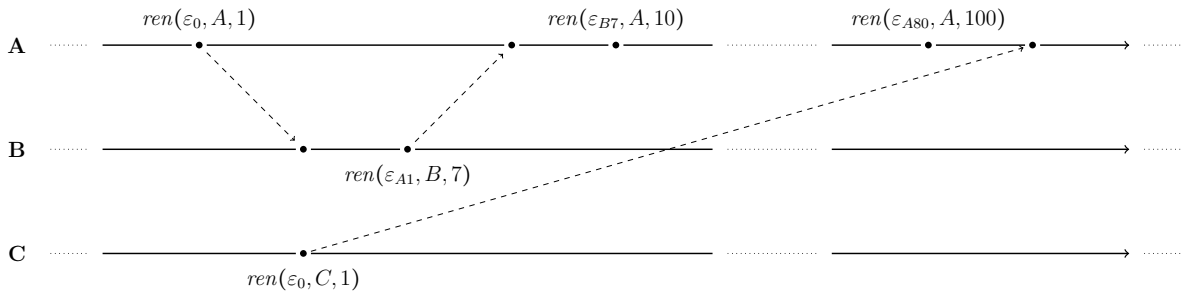
Pour limiter la consommation en bande passante des opérations *rename*, nous proposons la technique de compression suivante. Au lieu de diffuser les identifiants complets formant l'*ancien état*, les noeuds peuvent diffuser seulement les éléments nécessaires pour identifier de manière unique les intervalles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

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

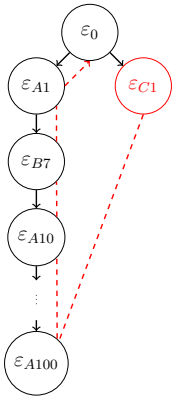
Grâce à cette méthode de compression, nous pouvons instaurer une taille maximale à l'opération *rename*. En effet, les noeuds peuvent émettre une opération *rename* dès que leur état courant atteint un nombre donné d'intervalles d'identifiants, bornant ainsi la taille du message à diffuser.

2.5.4 Définition de relations de priorité pour minimiser les traitements

Bien que la relation *priority* proposée dans la sous-section 2.2.2 soit simple et garantisse que tous les noeuds désignent la même époque comme époque cible, elle introduit un surcoût computationnel significatif dans certains cas. La Figure 2.15 présente un tel cas.



(a) Exécution d'opérations *rename* concurrentes



(b) *Arbre des époques* final correspondant avec la relation *priority* illustrée

FIGURE 2.15 – Livraison d'une opération *rename* d'un noeud

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

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

Afin d'offrir une plus grande flexibilité dans la conception de la relation *priority*, il est nécessaire de retirer la contrainte interdisant aux noeuds de rejouer une opération *rename*. Pour cela, un couple de fonctions réciproques doit être proposée pour RENAMEID et REVERTRENAMEID. Une solution alternative est de proposer une implémentation du mécanisme de renommage qui repose sur les identifiants originaux plutôt que sur ceux transformés, par exemple en utilisant le journal des opérations.

2.5.5 Report de la transition vers la nouvelle époque cible

Comme illustré par le Tableau 2.3, intégrer des opérations *rename* distantes est généralement coûteux. Ce traitement peut générer un surcoût computationnel significatif en cas de multiples opérations *rename* concurrentes. En particulier, un noeud peut recevoir et intégrer les opérations *rename* concurrentes dans l'ordre inverse défini par la relation *priority* sur leur époques. Dans ce scénario, le noeud considérerait chaque nouvelle époque introduite comme la nouvelle époque cible et renommerait son état en conséquence à chaque fois.

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

Durant cette période d'incertitude introduite par le report, les noeuds peuvent recevoir des opérations provenant d'époques différentes, notamment de l'époque cible. Néanmoins, les noeuds peuvent toujours intégrer les opérations *insert* et *remove* en utilisant RENA-

meId et revertRenameId au prix d'un surcoût computationnel pour chaque identifiant. Cependant, ce coût est négligeable (plusieurs centaines de microsecondes par identifiant d'après la Figure 2.13b) comparé au coût de renommer, de manière inutile, complètement l'état (plusieurs centaines de milliseconds à des secondes complètes d'après le Tableau 2.3).

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

2.5.6 Utilisation de l'opération de renommage comme mécanisme de compression du journal des opérations

Lorsqu'un nouveau pair rejoint la collaboration, il doit tout d'abord récupérer l'état courant du document avant de pouvoir participer. Le nouveau pair utilise un mécanisme d'anti-entropie [21] pour récupérer l'ensemble des opérations via un autre pair. Puis il reconstruit l'état courant en appliquant successivement chacune des opérations. Ce processus peut néanmoins s'avérer coûteux pour les documents comprenant des milliers d'opérations.

Pour pallier ce problème, des mécanismes de compression du journal ont été proposés dans la littérature. Les approches présentées dans [66, 67] consistent à remplacer un sous-ensemble des opérations du journal par une opération équivalente, par exemple en aggrégeant les opérations *insert* adjacentes. Une autre approche, présentée dans [12], définie une relation *obsolete* sur les opérations. La relation *obsolete* permet de spécifier qu'une nouvelle opération rend obsolètes des opérations précédentes et permet de les retirer du log. Pour donner un exemple, une opération d'ajout d'un élément donné dans un OR-Set CRDT rend obsolètes toutes les opérations précédentes d'ajout et de suppression de cet élément.

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

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

(i) L'opération *rename* ayant introduite son époque courante.

(ii) Les opérations *insert* et *remove* dont l'opération *rename* courante ne dépend pas causalement.

Notons que les données contenues dans l'opération *rename* telle que nous l'avons définie précédemment (cf. Définition 41) sont insuffisantes pour cette utilisation. En effet, les données incluses (*ancien état* au moment du renommage, identifiant du noeud auteur de l'opération *rename* et son numéro de séquence au moment de la génération) nous permettent seulement de recréer la structure de la séquence après le renommage. Mais le contenu de la séquence est omis, celui-ci n'étant jusqu'ici d'aucune utilité pour l'opération *rename*. Afin de pouvoir utiliser l'opération *rename* comme état initial, il est nécessaire d'y inclure cette information.

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

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

Définition 48 (rename enrichie)**.** Une opération *rename enrichie* est un quintuplet $\langle nodeId, nodeSeq, formerState, versionVector, content\rangle$ où

(i) *nodeId* est l'identifiant du noeud qui a généré l'opération *rename*.

(ii) *nodeSeq* est le numéro de séquence du noeud au moment de la génération de l'opération *rename*.

(iii) *formerState* est l'ancien état du noeud au moment du renommage.

(iv) *versionVector* est le vecteur de versions représentant l'ancien état du noeud au moment du renommage.

(v) *content* est le contenu du document au moment du renommage.

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

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

2.5.7 Implémentation alternative de l'intégration de l'opération de renommage basée sur le journal des opérations

Nous avons décrit précédemment l'algorithme que nous utilisons pour intégrer une opération de renommage (cf. section 2.4.1, page 77). Pour rappel, cet algorithme se compose principalement des étapes suivantes :

(i) L'identification de l'époque PPAC entre l'époque courante et l'époque cible, de façon à determiner les opérations de renommage à annuler et celles à appliquer.

(ii) L'utilisation successive des fonctions REVERTRENAMEID et RENAMEID sur l'ensemble des identifiants de l'état courant pour obtenir l'état correspondant à l'époque cible.

(iii) La création d'une nouvelle séquence à partir des identifiants calculés et du contenu courant.

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

Cette implémentation se base sur les observations suivantes :

(i) L'état courant est obtenu en intégrant successivement l'ensemble des opérations.

(ii) L'opération *rename* est une opération subsumant les opérations passées : elle prend un état donné (l'*ancien état*), somme des opérations précédentes, et génère un nouvel état équivalent compacté.

(iii) L'ordre d'intégration des opérations concurrentes n'a pas d'importance sur l'état final obtenu.

Ainsi, pour intégrer une opération *rename* distante, un noeud peut :

(i) Générer l'état correspondant au renommage de l'*ancien état*.

(ii) Identifier le chemin entre l'époque courante et l'époque cible.

(iii) Identifier les opérations concurrentes à l'opération *rename* présentes dans son journal des opérations.

(iv) Transformer et intégrer successivement les opérations concurrentes à l'opération *rename* à ce nouvel état.

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

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

Un autre avantage de cette approche est qu'elle permet de récupérer et de réutiliser les identifiants originaux des opérations. Lorsqu'une suite de transformations est appliquée

sur les identifiants d'une opération, elle est appliquée sur les identifiants originaux et non plus sur leur équivalents présents dans l'état courant. Ceci permet de réinitialiser les transformations appliquées à un identifiant et d'éviter le cas de figure mentionné dans la sous-section 2.2.3 : le cas où REVERTRENAMEID est utilisé pour retirer l'effet d'une opération *rename* sur un identifiant, avant d'utiliser RENAMEID pour ré-intégrer l'effet de la même opération *rename*. Cette implémentation supprime donc la contrainte de définir un couple de fonctions réciproques RENAMEID et REVERTRENAMEID, ce qui nous offre une plus grande flexibilité dans le choix de la relation $<_\varepsilon$ et du couple de fonctions RENAMEID et REVERTRENAMEID.

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

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

Finalement, cette approche implique aussi de parcourir le journal des opérations à la recherche d'opérations concurrentes. Comme dit précédemment, la taille du journal croît de façon monotone au fur et à mesure que les noeuds émettent des opérations. Cette étape du nouvel algorithme d'intégration de l'opération *rename* devient donc de plus en plus coûteuse. Des méthodes permettent néanmoins de réduire son coût computationnel. Notamment, chaque noeud traquent les informations de progression des autres noeuds afin de supprimer les métadonnées du mécanisme de renommage (cf. section 2.3, page 72). Ces informations permettent de déterminer la stabilité causale des opérations et donc d'identifier les opérations qui ne peuvent plus être concurrentes à une nouvelle opération *rename*. Les noeuds peuvent ainsi maintenir, en plus du journal complet des opérations, un journal composé uniquement des opérations non stables causalement. Lors du traitement d'une nouvelle opération *rename*, les noeuds peuvent alors parcourir ce journal réduit à la recherche des opérations concurrentes.

2.6 Comparaison avec les approches existantes

Dans cette section, nous comparons RenamableLogootSplit aux travaux existants de la littérature qui visent à réduire la croissance monotone du surcoût en métadonnées et en calculs des CRDTs pour le type Séquence à identifiants densément ordonnés. Pour cela, nous rappelons tout d'abord les contributions de ces travaux. Puis, nous mettons en lumière comment RenamableLogootSplit se différencie de ces derniers.

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

[57] puis [58] définissent un mécanisme de ré-équilibrage de l'arbre des identifiants de position pour Treedoc [39]. Ce mécanisme, qui prend la forme d'une nouvelle opération *rebalance*, permet aux noeuds de réassigner des identifiants plus courts aux éléments du document. Cependant, cette nouvelle opération n'est ni commutative avec les opérations *insert* et *remove*, ni avec elle-même. Pour assurer la convergence à terme [3] du système, [58] fait le choix d'empêcher la génération d'opérations *rebalance* concurrentes. Pour cela, cette approche requiert un consensus entre les noeuds pour générer les opérations *rebalance*. Les opérations *insert* et *remove* sont elles toujours générées sans coordination entre les noeuds et peuvent donc être concurrentes aux opérations *rebalance*. Pour gérer les opérations concurrentes aux opérations *rebalance*, les auteurs proposent de transformer les opérations concernées par rapport aux effets des opérations*rebalance*, à l'aide un mécanisme de *catch-up*, avant de les appliquer.

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

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

Dans les systèmes disposant d'un *core*, nous pouvons donc combiner RenamableLogootSplit avec un protocole de consensus pour éviter la génération d'opérations *rename* concurrentes. Cette approche offre plusieurs avantages. Elle permet de se passer de tout ce qui à attrait au support d'opérations *rename* concurrentes, c.-à-d. la définition d'une relation *priority* et l'implémentation de RevertRenameId. Elle permet aussi de simplifier l'implémentation du mécanisme de récupération de mémoire des époques et *anciens états* pour reposer seulement sur la stabilité causale des opérations. Concernant ses performances, cette approche se comporte de manière similaire à RenamableLogootSplit avec

un seul *renaming bot* (cf. sous-section 2.4.3, page 81), mais avec un surcoût correspondant au coût du protocole de consensus sélectionné.

2.6.2 Ralentissement de la croissance des identifiants de position

L'approche LSEQ [59, 60] est une approche visant à ralentir la croissance des identifiants dans les CRDTs pour le type Séquence à identifiants densément ordonnés. Au lieu de réduire périodiquement la taille des métadonnées liées aux identifiants à l'aide d'un mécanisme de renommage coûteux, les auteurs définissent de nouvelles stratégies d'allocation des identifiants pour ralentir leur vitesse de croissance. Dans ces travaux, les auteurs notent que la stratégie d'allocation des identifiants proposée dans Logoot [35] n'est adaptée qu'à un seul comportement d'édition : de gauche à droite, de haut en bas. Si les insertions sont effectuées en suivant d'autres comportements, les identifiants générés saturent rapidement l'espace des identifiants pour une taille donnée. Les insertions suivantes déclenchent alors une augmentation de la taille des identifiants. En conséquent, la taille des identifiants dans Logoot augmente de façon linéaire au nombre d'insertions, au lieu de suivre la progression logarithmique attendue.

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

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

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

2.7 Conclusion

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

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

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

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

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

RenamableLogootSplit souffre néanmoins de plusieurs limitations. La première d'entre elles est le besoin d'observer la stabilité causale des opérations *rename* pour supprimer de manière définitive les métadonnées associées. Il s'agit d'une contrainte forte, notamment dans les systèmes dynamiques à grande échelle dans lesquels nous n'avons aucune garantie et aucun contrôle sur les noeuds. Il est donc possible qu'un noeud déconnecté ne se reconnecte jamais, bloquant ainsi la progression de la stabilité causale pour l'ensemble des opérations. Il s'agit toutefois d'une limite partagée avec les autres mécanismes de réduction des métadonnées pour Sequence CRDTs proposés dans la littérature [38, 58], à l'exception de l'approche LSEQ [60]. En pratique, il serait intéressant d'étudier la mise en place d'un mécanisme d'éviction des noeuds inactifs pour répondre à ce problème.

La seconde limitation de RenamableLogootSplit concerne la génération d'opérations *rename* concurrentes. Chaque opération *rename* est coûteuse, aussi bien en terme de métadonnées à stocker et diffuser qu'en terme de traitements à effectuer. Il est donc important de chercher à minimiser le nombre d'opérations *rename* concurrentes émises par les noeuds. Une approche possible est d'adopter une architecture du type *core-nebula*[58]. Mais pour les systèmes incompatibles avec ce type d'architecture système, il serait intéressant de proposer d'autres approches ne nécessitant aucune coordination entre les noeuds. Mais par définition, ces approches ne pourraient offrir de garanties fortes sur le nombre d'opérations concurrentes possibles.

Bibliographie

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

[2] 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.

[3] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer et Carl H Hauser. « Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System ». In : *SIGOPS Oper. Syst. Rev.* 29.5 (déc. 1995), p. 172–182. issn : 0163-5980. doi : 10.1145/224057.224070. url : https://doi.org/10.1145/224057.224070.

[4] 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.

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

[6] 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.

[7] 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.

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

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

[10] 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.

[11] Marc SHAPIRO, Nuno PREGUIÇA, Carlos BAQUERO et Marek ZAWIRSKI. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, jan. 2011, p. 50. URL : https://hal.inria.fr/inria-00555588.

[12] 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.

[13] Carlos BAQUERO, Paulo Sergio ALMEIDA et Ali SHOKER. *Pure Operation-Based Replicated Data Types*. 2017. arXiv : 1710.04469 [cs.DC].

[14] 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.

[15] 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.

[16] Prince MAHAJAN, Lorenzo ALVISI, Mike DAHLIN et al. « Consistency, availability, and convergence ». In : *University of Texas at Austin Tech Report* 11 (2011), p. 158.

[17] Friedemann MATTERN et al. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[18] Colin FIDGE. « Logical Time in Distributed Computing Systems ». In : *Computer* 24.8 (août 1991), p. 28–33. ISSN : 0018-9162. DOI : 10.1109/2.84874. URL : https://doi.org/10.1109/2.84874.

[19] 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.

[20] Vitor ENES, Paulo Sérgio ALMEIDA, Carlos BAQUERO et João LEITÃO. « Efficient Synchronization of State-Based CRDTs ». In : *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, p. 148–159. DOI : 10.1109/ICDE.2019.00022.

[21] D. S. PARKER, G. J. POPEK, G. RUDISIN, A. STOUGHTON, B. J. WALKER, E. WALTON, J. M. CHOW, D. EDWARDS, S. KISER et C. KLINE. « Detection of Mutual Inconsistency in Distributed Systems ». In : *IEEE Trans. Softw. Eng.* 9.3 (mai 1983), p. 240–247. ISSN : 0098-5589. DOI : 10.1109/TSE.1983.236733. URL : https://doi.org/10.1109/TSE.1983.236733.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall et Werner Vogels. « Dynamo : Amazon's highly available key-value store ». In : *ACM SIGOPS operating systems review* 41.6 (2007), p. 205–220.

[23] 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.

[24] Ricardo Jorge Tomé Gonçalves, Paulo Sérgio Almeida, Carlos Baquero et Vitor Fonte. « DottedDB : Anti-Entropy without Merkle Trees, Deletes without Tombstones ». In : *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. 2017, p. 194–203. doi : 10.1109/SRDS.2017.28.

[25] 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.

[26] Carlos Baquero, Paulo Sérgio Almeida et Ali Shoker. « Making Operation-Based CRDTs Operation-Based ». In : *Distributed Applications and Interoperable Systems*. Sous la dir. de Kostas Magoutis et Peter Pietzuch. Berlin, Heidelberg : Springer Berlin Heidelberg, 2014, p. 126–140.

[27] Clarence A. Ellis et Simon J. Gibbs. « Concurrency Control in Groupware Systems ». In : *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. Portland, Oregon, USA : Association for Computing Machinery, 1989, p. 399–407. isbn : 0897913175. doi : 10.1145/67544.66963. url : https://doi.org/10.1145/67544.66963.

[28] 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.

[29] 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.

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

[31] 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.

[32] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang et David Chen. « Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems ». In : *ACM Transactions on Computer-Human Interaction (TOCHI)* 5.1 (1998), p. 63–108.

[33] Gérald Oster, Pascal Molli, Pascal Urso et Abdessamad Imine. « Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems ». In : *2006 International Conference on Collaborative Computing : Networking, Applications and Worksharing*. 2006, p. 1–10. doi : 10.1109/COLCOM.2006.361867.

[34] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang et David Chen. « Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems ». In : *ACM Trans. Comput.-Hum. Interact.* 5.1 (mar. 1998), p. 63–108. issn : 1073-0516. doi : 10.1145/274444.274447. url : https://doi.org/10.1145/274444.274447.

[35] 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.

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

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

[38] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim et Joonwon Lee. « Replicated abstract data types : Building blocks for collaborative applications ». In : *Journal of Parallel and Distributed Computing* 71.3 (2011), p. 354–368. issn : 0743-7315. doi : https://doi.org/10.1016/j.jpdc.2010.12.006. url : http://www.sciencedirect.com/science/article/pii/S0743731510002716.

[39] Nuno Preguica, Joan Manuel Marques, Marc Shapiro et Mihai Letia. « A Commutative Replicated Data Type for Cooperative Editing ». In : *2009 29th IEEE International Conference on Distributed Computing Systems*. Juin 2009, p. 395–403. doi : 10.1109/ICDCS.2009.20.

[40] 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.

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

[42] Charbel Rahhal, Stéphane Weiss, Hala Skaf-Molli, Pascal Urso et Pascal Molli. *Undo in Peer-to-peer Semantic Wikis*. Research Report RR-6870. INRIA, 2009, p. 18. URL : https://hal.inria.fr/inria-00366317.

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

[44] 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.

[45] 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.

[46] 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.

[47] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang et Marek Zawirski. « Specification and Complexity of Collaborative Text Editing ». In : *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. PODC '16. Chicago, Illinois, USA : Association for Computing Machinery, 2016, p. 259–268. ISBN : 9781450339643. DOI : 10.1145/2933057.2933090. URL : https://doi.org/10.1145/2933057.2933090.

[48] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang et Marek Zawirski. « Specification and space complexity of collaborative text editing ». In : *Theoretical Computer Science* 855 (2021), p. 141–160. ISSN : 0304-3975. DOI : https://doi.org/10.1016/j.tcs.2020.11.046. URL : http://www.sciencedirect.com/science/article/pii/S0304397520306952.

[49] 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.

[50] 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.

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

[52] Martin KLEPPMANN, Victor B. F. GOMES, Dominic P. MULLIGAN et Alastair R. BERESFORD. « Interleaving Anomalies in Collaborative Text Editors ». In : *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '19. Dresden, Germany : Association for Computing Machinery, 2019. ISBN : 9781450362764. DOI : 10.1145/3301419.3323972. URL : https://doi.org/10.1145/3301419.3323972.

[53] Matthew WEIDNER. *There Are No Doubly Non-Interleaving List CRDTs*. Last Accessed : 2022-10-07. URL : https://mattweidner.com/assets/pdf/List_CRDT_Non_Interleaving.pdf.

[54] 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.

[55] Claudia-Lavinia IGNAT, Gérald OSTER, Meagan NEWMAN, Valerie SHALIN et François CHAROY. « Studying the Effect of Delay on Group Performance in Collaborative Editing ». In : *Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014, Springer 2014 Lecture Notes in Computer Science*. Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014. Seattle, WA, United States, sept. 2014, p. 191–198. DOI : 10.1007/978-3-319-10831-5_29. URL : https://hal.archives-ouvertes.fr/hal-01088815.

[56] Claudia-Lavinia IGNAT, Gérald OSTER, Olivia FOX, François CHAROY et Valerie SHALIN. « How Do User Groups Cope with Delay in Real-Time Collaborative Note Taking ». In : *European Conference on Computer Supported Cooperative Work 2015*. Sous la dir. de Nina BOULUS-RODJE, Gunnar ELLINGSEN, Tone BRATTETEIG, Margunn AANESTAD et Pernille BJORN. Proceedings of the 14th European Conference on Computer Supported Cooperative Work. Oslo, Norway : Springer International Publishing, sept. 2015, p. 223–242. DOI : 10.1007/978-3-319-20499-4_12. URL : https://hal.inria.fr/hal-01238831.

[57] 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.

[58] 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.

[59] Brice Nédelec, Pascal Molli, Achour Mostéfaoui et Emmanuel Desmontils. « LSEQ : an adaptive structure for sequences in distributed collaborative editing ». In : *Proceedings of the 2013 ACM Symposium on Document Engineering*. DocEng 2013. Sept. 2013, p. 37–46. DOI : 10.1145/2494266.2494278.

[60] Brice Nédelec, Pascal Molli et Achour Mostéfaoui. « A scalable sequence encoding for collaborative editing ». In : *Concurrency and Computation : Practice and Experience* (), e4108. DOI : 10.1002/cpe.4108. eprint : https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4108. URL : https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4108.

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

[62] Matthieu Nicolas, Gérald Oster et Olivier Perrin. « Efficient Renaming in Sequence CRDTs ». In : *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'20)*. Heraklion, Greece, avr. 2020. URL : https://hal.inria.fr/hal-02526724.

[63] Matthieu Nicolas, Gerald Oster et Olivier Perrin. « Efficient Renaming in Sequence CRDTs ». In : *IEEE Transactions on Parallel and Distributed Systems* 33.12 (déc. 2022), p. 3870–3885. DOI : 10.1109/TPDS.2022.3172570. URL : https://hal.inria.fr/hal-03772633.

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

[65] 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.

[66] Haifeng Shen et Chengzheng Sun. « A log compression algorithm for operation-based version control systems ». In : *Proceedings 26th Annual International Computer Software and Applications*. 2002, p. 867–872. DOI : 10.1109/CMPSAC.2002.1045115.

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

Résumé

Un système collaboratif permet à plusieurs utilisateur-rices de créer ensemble un contenu. Afin de supporter des collaborations impliquant des millions d'utilisateurs, ces systèmes adoptent une architecture décentralisée pour garantir leur haute disponibilité, tolérance aux pannes et capacité de passage à l'échelle. Cependant, ces sytèmes échouent à garantir la confidentialité des données, souveraineté des données, pérennité et résistance à la censure. Pour répondre à ce problème, la littérature propose la conception d'applications Local-First Software (LFS) : des applications collaboratives pair-à-pair (P2P).

Une pierre angulaire des applications LFS sont les Conflict-free Replicated Data Types (CRDTs). Il s'agit de nouvelles spécifications des types de données, tels que l'Ensemble ou la Séquence, permettant à un ensemble de noeuds de répliquer une donnée. Les CRDTs permettent aux noeuds de consulter et de modifier la donnée sans coordination préalable, et incorporent un mécanisme de résolution de conflits pour intégrer les modifications concurrentes. Cependant, les CRDTs pour le type Séquence souffrent d'une croissance monotone du surcoût de leur mécanisme de résolution de conflits. Pouvons-nous proposer un mécanisme de réduction du surcoût des CRDTs pour le type Séquence qui soit compatible avec les applications LFS ? Dans cette thèse, nous proposons un nouveau CRDT pour le type Séquence, RenamableLogootSplit. Ce CRDT intègre un mécanisme de renommage qui minimise périodiquement le surcoût de son mécanisme de résolution de conflits ainsi qu'un mécanisme de résolution de conflits pour intégrer les modifications concurrentes à un renommage. Finalement, nous proposons un mécanisme de Garbage Collection (GC) qui supprime à terme le propre surcoût du mécanisme de renommage.

Abstract

A collaborative system enables multiple users to work together to create content. To support collaborations involving millions of users, these systems adopt a decentralised architecture to ensure high availability, fault tolerance and scalability. However, these systems fail to guarantee the data confidentiality, data sovereignty, longevity and resistance to censorship. To address this problem, the literature proposes the design of Local-First Software (LFS) applications : collaborative peer-to-peer applications.

A cornerstone of LFS applications are Conflict-free Replicated Data Types (CRDTs). CRDTs are new specifications of data types, e.g. Set or Sequence, enabling a set of nodes to replicate a data. CRDTs enable nodes to access and modify the data without prior coordination, and incorporate a conflict resolution mechanism to integrate concurrent modifications. However, Sequence CRDTs suffer from a monotonous growth in the overhead of their conflict resolution mechanism. Can we propose a mechanism for reducing the overhead of Sequence-type CRDTs that is compatible with LFS applications ? In this thesis, we propose a novel CRDT for the Sequence type, RenamableLogootSplit. This CRDT embeds a renaming mechanism that periodically minimizes the overhead of its conflict resolution mechanism as well as a conflict resolution mechanism to integrate concurrent modifications to a rename. Finally, we propose a mechanism of Garbage Collection (GC) that eventually removes the own overhead of the renaming mechanism.