Chapitre 2

Renommage dans une séquence répliquée

Sommaire

2.1 Présentation de l'approche

Nous proposons un nouveau CRDT pour la *Sequence* appartenant à l'approche des identifiants densément ordonnées : RenamableLogootSplit [38, 39]. Cette structure de données permet aux pairs d'insérer et de supprimer des éléments au sein d'une séquence répliquée. Nous introduisons une opération *rename* qui permet de (i) réassigner des identifiants plus courts aux différents éléments de la séquence (ii) fusionner les blocs composant la séquence. Ces deux actions permettent à l'opération *rename* de produire un nouvel état minimisant son surcoût en métadonnées.

2.1.1 Modèle du système

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

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

Puisque RenamableLogootSplit est une extension de LogootSplit, cette structure de données partage les mêmes contraintes sur la livraison de messages. La couche de livraison de messages sert donc à livrer les messages à l'application exactement une fois. La couche de livraison de messages a aussi pour tâche de garantir la livraison des opérations de suppression après les opérations d'insertion correspondantes. Aucune autre contrainte n'existe sur l'ordre de livraison des opérations. Finalement, la couche de livraison intègre aussi un mécanisme d'anti-entropie [42]. Ce mécanisme permet aux noeuds de se synchroniser par paires, en détectant et ré-échangeant les messages perdus.

2.1.2 Définition de l'opération de renommage

L'objectif de l'opération *rename* est de réassigner de nouveaux identifiants aux éléments de la séquence répliquée sans modifier son contenu. Puisque les identifiants sont des métadonnées utilisées par la structure de données uniquement afin de résoudre les

conflits, les utilisateurs ignorent leur existence. Les opérations *rename* sont donc des opérations systèmes : elles sont émises et appliquées par les noeuds en coulisses, sans aucune intervention des utilisateurs.

Afin de garantir le respect du modèle de cohérence SEC, nous définissons plusieurs propriétés de sécurité que l'opération *rename* doit respecter. Ces propriétés sont inspirées principalement par celles proposées dans [60].

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

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

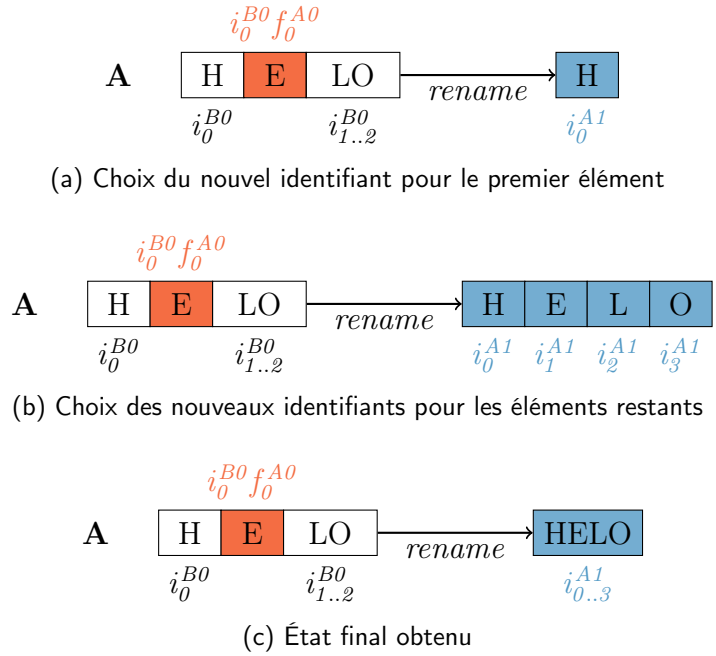
Propriété 3 *(Séquence bien formée) La séquence répliquée doit être bien formée. Appliquée une opération* rename *sur une séquence bien formée doit produire une nouvelle séquence bien formée. Une séquence bien formée doit respecter les propriétés suivantes :*

> ***Propriété 3.1*** *(Préservation de l'unicité) Chaque identifiant doit être unique. Donc, pour une opération* rename *donnée, chaque identifiant doit être associé à un nouvel identifiant distinct.*
> ***Propriété 3.2*** *(Préservation de l'ordre) Les éléments de la séquence doivent être triés en fonction de leur identifiants. L'ordre existant entre les identifiants initiaux doit donc être préservé par l'opération* rename.

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

La 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 un souci de simplicité, la présentation de l'opération *rename* est divisée en deux parties. Dans le section 2.2, nous présentons l'opération *rename* proposée avec l'hypothèse qu'aucune opération *rename* concurrente ne peut être générée. Cette hypothèse nous



(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

permet de nous concentrer sur le fonctionnement de l'opération *rename* elle-même ainsi que sur comment gérer les opérations *insert* et *remove* concurrentes. Ensuite, dans le section 2.3, nous supprimons cette hypothèse. Nous présentons alors notre approche pour gérer les scénarios avec des opérations *rename* concurrentes.

2.2 Introduction de l'opération *rename*

2.2.1 Opération de renommage proposée

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

Son comportement est illustré dans la Figure 2.1. Dans cet exemple, le noeud A initie une opération *rename* sur son état local. Tout d'abord, le noeud A génère un nouvel identifiant à partir du premier tuple de l'identifiant du premier élément de la séquence (i_0^{B0}). Pour générer ce nouvel identifiant, le noeud A reprend la position de ce tuple (i) mais utilise son propre identifiant de noeud (**A**) et numéro de séquence actuel (*1*). De plus, son offset est mis à 0. Le noeud A réassigne l'identifiant résultant (i_0^{A1}) au premier élément de la séquence, comme décrit dans 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 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 son *ancien état* aux autres.

Définition 10 (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 11 (rename) *Une opération* rename *est un triplet* ⟨*nodeId, nodeSeq, formerState*⟩ *où*

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

En utilisant ces données, les autres noeuds calculent le nouvel identifiant de chaque identifiant renommé. Concernant les identifiants insérés de manière concurrente au renommage, nous expliquons dans la sous-section 2.2.2 comment les noeuds peuvent les renommer de manière déterministe.

2.2.2 Gestion des opérations concurrentes au renommage

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

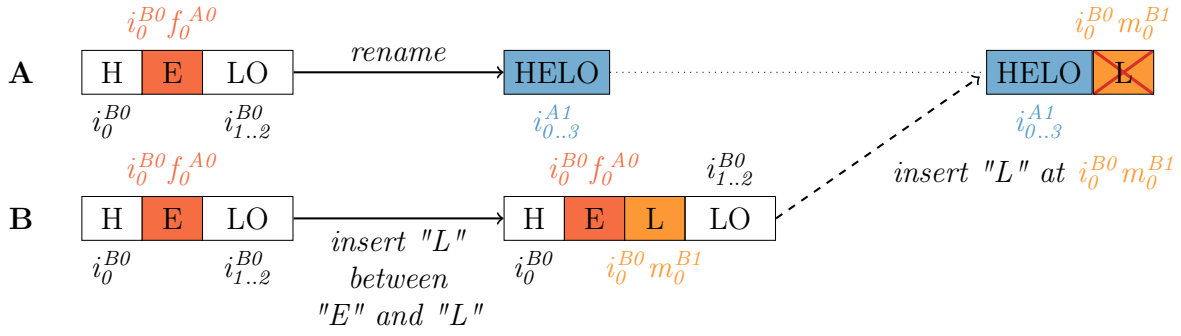


FIGURE 2.2 – Modifications concurrentes menant à une anomalie

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 Figure 2.2 permet aux noeuds de faire progresser leur état de ε_0 à ε_{A1}. Nous définissons les époques de la manière suivante :

Définition 12 (Époque) *Une époque est un couple ⟨nodeId, nodeSeq⟩ où*

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

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

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

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

Les noeuds utilisent la fonction RENAMEID, décrite dans l'Algorithme 1, pour transformer les opérations *insert* et *remove* par rapport aux opérations *rename*. Cet algorithme associe les identifiants d'une époque *parente* aux identifiants correspondant dans l'époque *enfant*. L'idée principale de cet algorithme est de renommer les identifiants inconnus au moment de la génération de l'opération *rename* en utilisant leur 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 les identifiants générés de façon concurrente avant de les insérer dans son état.

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

Algorithme 1 Fonctions principales pour renommer un identifiant

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

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

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

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

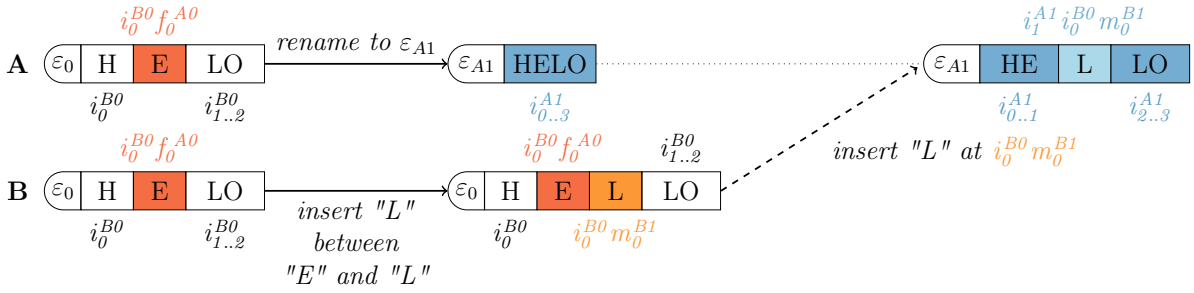


FIGURE 2.3 – Renommage de la modification concurrente avant son intégration en utilisant RENAMEID afin de maintenir l'ordre souhaité

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 1 présente seulement le cas principal de RENAMEID, c.-à-d. le cas où l'identifiant à renommer appartient à l'intervalle des identifiants formant l'ancien état ($firstId \leq_{id} id \leq_{id} lastId$). Les fonctions pour gérer les autres cas, c.-à-d. les cas où

l'identifiant à renommer n'appartient pas à cet intervalle ($id <_{id} firstId$ ou $lastId <_{id} id$), sont présentées dans l'Annexe A.

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

2.2.3 Évolution du modèle de livraison des opérations

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

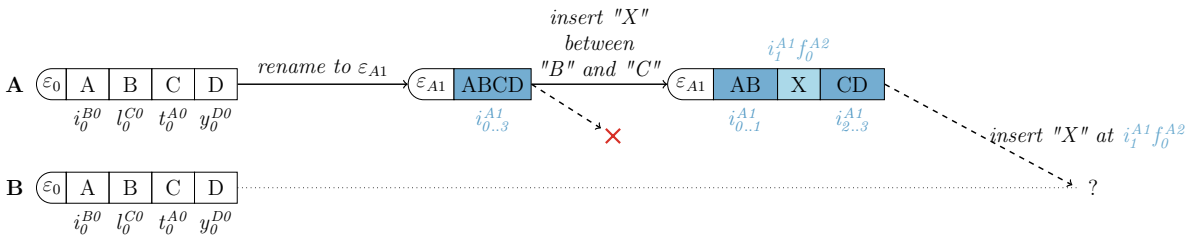


FIGURE 2.4 – Livraison d'une opération *insert* sans avoir reçu l'opération *rename* précédente

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

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

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

Pour cela, nous proposons de faire évoluer le modèle de livraison des opérations de RenamableLogootSplit. Celui-ci repose sur celui de LogootSplit, que nous avions défini dans la Définition 9. Pour rappel, ce modèle requiert que (i) les opérations soient livrées

qu'une seule et unique fois au CRDT, (ii) les opérations *remove* soient livrées au CRDT qu'après les opérations *insert* ajoutant les éléments à supprimer.

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

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

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

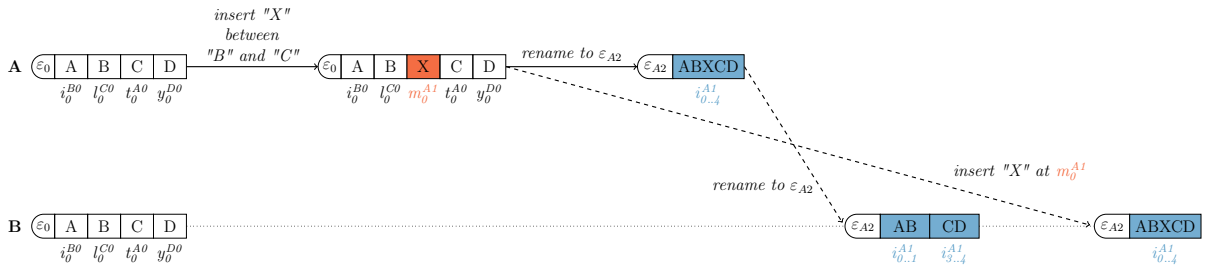


FIGURE 2.5 – Livraison désordonnée d'une opération *rename* et de l'opération *insert* qui la précède

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 est renomme chaque identifiant composant son état.

Ensuite, le noeud B reçoit l'opération *insert*. Comme l'époque de génération de l'opération *insert* (ε_0) est différente de celle de son état courant (ε_{A2}), le noeud B utilise RENAMEID pour renommer l'identifiant avant de l'insérer. m_0^{A1} faisant partie de l'*ancien*

état, le noeud B utilise l'index de cet identifiant dans l'*ancien état* (2) pour calculer son équivalent à l'époque ε_{A2} (i_2^{A2}). Le noeud B insère l'élément "X" avec ce nouvel identifiant et converge alors avec le noeud A, malgré la livraison dans le désordre des opérations.

2.3 Gestion des opérations *rename* concurrentes

2.3.1 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 Figure 2.3.

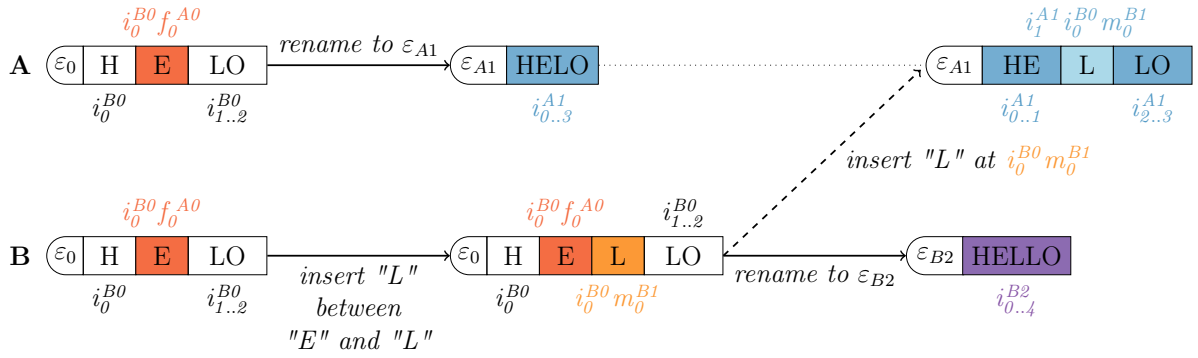


FIGURE 2.6 – Opérations *rename* concurrentes menant à des états divergents

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

Puisque des époques concurrentes sont générées, les époques forment désormais l'*arbre des époques*. Nous représentons dans la Figure 2.7 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.

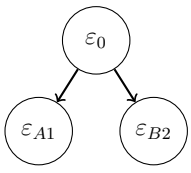


FIGURE 2.7 – *Arbre des époques* correspondant au scénario décrit dans la Figure 2.6

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 1 permet

seulement aux noeuds de progresser d'une époque *parente* à une de ses époques *enfants*. Le noeud A (resp. B) est donc dans l'incapacité de progresser vers l'époque du noeud B (resp. A). Il est donc nécessaire de faire évoluer notre mécanisme de renommage pour sortir de cette impasse.

Tout d'abord, les noeuds doivent se mettre d'accord sur une époque commune de l'*arbre des époques* comme époque cible. Afin d'éviter des problèmes de performances 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 2.3.2.

Ensuite, les noeuds doivent se déplacer à travers l'*arbre des époques* afin d'atteindre l'époque cible. La fonction RENAMEID permet déjà aux noeuds de descendre dans l'arbre. Les cas restants à gérer sont ceux où les noeuds se trouvent actuellement à une époque *soeur* ou *cousine* de l'époque cible. Dans ces cas, les noeuds doivent être capable de remonter dans l'*arbre des époques* pour retourner au Plus Petit Ancêtre Commun (PPAC) de l'époque courante et l'époque cible. Ce déplacement est en fait similaire à annuler l'effet des opérations *rename* précédemment appliquées. Nous proposons un algorithme, REVERTRENAMEID, qui remplit cet objectif dans la sous-section 2.3.3.

2.3.2 Relation de priorité entre renommages

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

Définition 14 (Relation *priority* $<_\varepsilon$) *La relation* priority $<_\varepsilon$ *est un ordre strict total sur l'ensemble des époques. Elle permet aux noeuds de comparer n'importe quelle paire d'époques.*

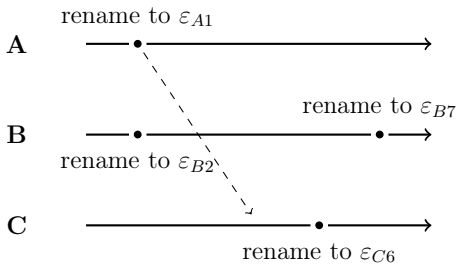
En utilisant la relation *priority*, nous définissons l'époque cible de la manière suivante :

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

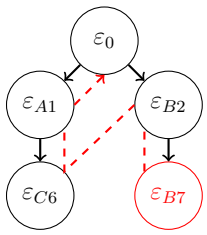
Pour définir la relation *priority*, nous pouvons choisir entre plusieurs stratégies. Dans le cadre de ce travail, nous utilisons l'ordre lexicographique sur le chemin des époques dans l'*arbre des époques*. La Figure 2.8 fournit un exemple.

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

Une fois que les noeuds se sont synchronisés, ils obtiennent l'*arbre des époques* représenté dans la Figure 2.8b. Dans cette figure, la flèche tireté rouge représente l'ordre entre



(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

les époques d'après la relation *priority* tandis que l'époque cible choisie est représentée sous la forme d'un noeud rouge.

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

D'autres stratégies pourraient être proposées pour définir la relation *priority*. Par exemple, *priority* pourrait reposer sur des métriques intégrées au sein des opérations *rename* pour représenter le travail accumulé sur le document. Cela permettrait de favoriser la branche de l'*arbre des époques* avec le plus de collaborateurs actifs pour minimiser la quantité globale de calculs effectués par les noeuds du système. Nous approfondissons ce sujet dans la sous-section 2.5.4.

2.3.3 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.

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

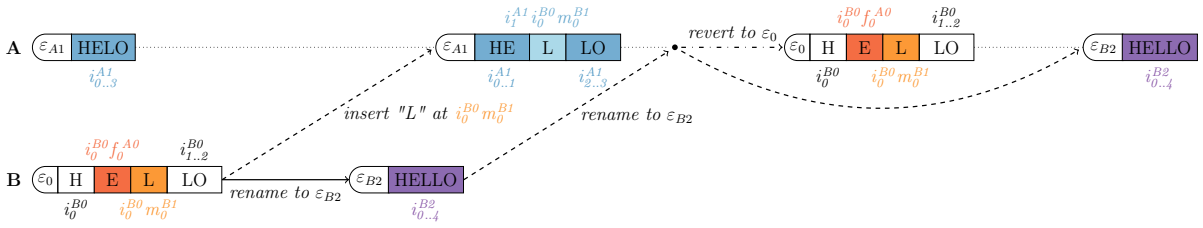


FIGURE 2.9 – Annulation d'une opération *rename* intégrée précèdemment en présence d'un identifiant inséré en concurrence

dans l'Algorithme 2.

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

Le cas (i) est le plus trivial. Pour retrouver la valeur de *id* à partir de *newId* [1], REVERTRENAMEID utilise simplement la valeur de offset de *newId*. En effet, cette valeur correspond à l'index de *id* dans l'*ancien état* (c.-à-d. $renamedIds[offset] = id$). Par exemple, dans la Figure 2.9, l'identifiant i_0^{A1} a pour offset 0, REVERTRENAMEID renvoie donc $renamedIds[0] = i_0^{B0}$.

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

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

et nous devons obtenir :

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

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

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

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

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

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

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

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

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

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

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

Nous avons donc bien :

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

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

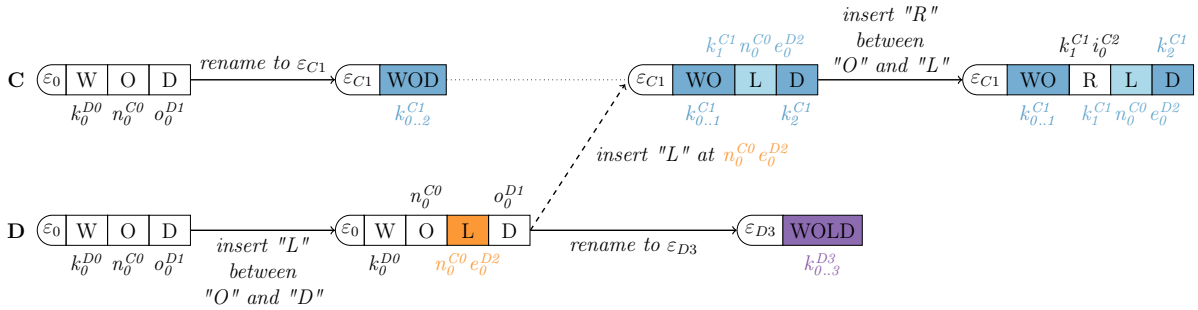
$$i_0^{B0} f_0^{A0} <_{id} i_0^{B0} m_0^{B1} <_{id} i_1^{B0}$$

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

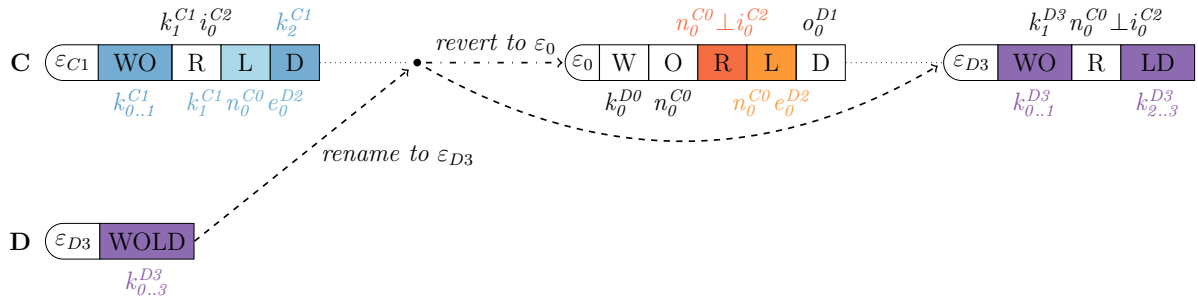
alors :

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

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



(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

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 concurrence, le noeud D insère l'élément "L" entre les éléments "O" et "D". L'opération *insert* correspondante est délivrée au noeud C, qui l'intègre en suivant le comportement défini en sous-section 2.2.2. Le noeud C procède ensuite à l'insertion de l'élément "R" entre les éléments "O" et "L". Cette insertion dépend donc causalement de l'opération *rename* effectuée précédemment par C. En parallèle, le noeud D effectue un renommage de son état. Cette opération *rename* est donc concurrente à l'opération *rename* générée précédemment par C.

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

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

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

D'après les règles présentées précédemment :

— k_1^{C1} est transformé en n_0^{C0} (cas (i))

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

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

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

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

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

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

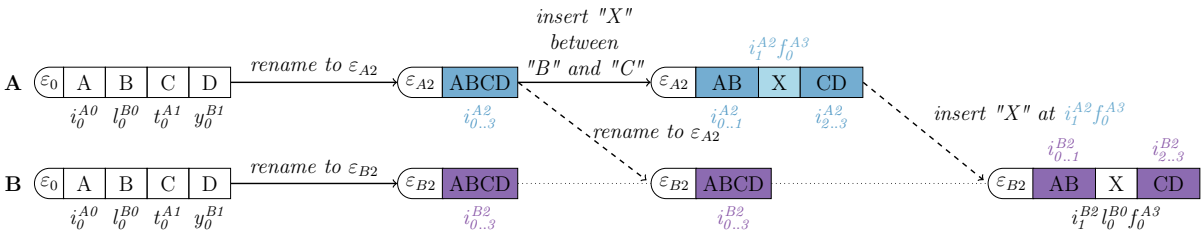
$$newId \in]newPredId\ succId, newSuccId[$$

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

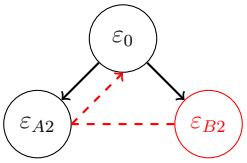
Comme pour l'Algorithme 1, l'Algorithme 2 ne présente seulement que le cas principal de revertRenameId. Il s'agit du cas où l'identifiant à restaurer appartient à l'intervalle des identifiants renommés $newFirstId \leq_{id} id \leq_{id} newLastId$). Les fonctions pour gérer les cas restants sont présentées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

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

Ce problème est toutefois évité dans notre système grâce à la relation *priority* utilisée. Puisque la relation *priority* est définie en utilisant l'ordre lexicographique sur le chemin



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



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

FIGURE 2.11 – Intégration d'une opération *insert* distante

des époques dans l'*arbre des époques*, les noeuds se déplacent seulement vers l'époque la plus à droite de l'*arbre des époques* lorsqu'ils changent d'époque. Les noeuds évitent donc d'aller et revenir entre deux mêmes époques, et donc d'annuler et rejouer les opérations *rename* correspondantes.

2.3.4 Processus d'intégration d'une opération

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

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

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

Cet algorithme se décompose en de multiples étapes. Afin d'illustrer chacune d'entre elles, nous utilisons l'exemple représenté par la Figure 2.11.

Dans la Figure 2.11a, deux noeuds A et B éditent une séquence répliquée via RenamableLogootSplit. Initialement, les deux noeuds possèdent des répliques identiques. Le noeud A commence par effectuer une opération *rename*. Il génère alors l'état équivalent à son état précédent, à la nouvelle époque ε_{A2}. Puis il effectue une opération *insert*, insérant un nouvel élément "X" entre les éléments "B" et "C". L'identifiant $i_1^{A2} f_0^{A3}$ est attribué à ce nouvel élément. Chacune des opérations du noeud A est diffusée sur le réseau.

De son côté, le noeud B génère en concurrence sa propre opération *rename* sur l'état initial. Il obtient alors un état équivalent, à l'époque ε_{B2}. Il reçoit ensuite l'opération *rename* du noeud A, qu'il intègre. Puisque $\varepsilon_{A2} <_\varepsilon \varepsilon_{B2}$, le noeud B ne modifie pas son époque courante (ε_{B2}). Le noeud B obtient toutefois l'*arbre des époques* représenté dans la Figure 2.11b.

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

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

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

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

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

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

Tout d'abord, le noeud B compare l'époque de l'opération avec l'époque courante de la séquence. Si les deux époques correspondaient, le noeud B pourrait intégrer l'opération directement en utilisant l'algorithme de LogootSplit dénommé ici INSERT. Mais dans le cas présent, l'époque de l'opération (ε_{A2}) est différente de l'époque courante (ε_{B2}). Il lui est donc nécessaire de transformer l'opération avant de pouvoir l'appliquer.

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

La fonction GETPATHBETWEENEPOCHS applique l'algorithme suivant : (i) elle calcule le chemin entre l'époque de l'opération et la racine de l'*arbre des époques* ($[\varepsilon_{A2}, \varepsilon_0]$) (ii) elle calcule le chemin entre l'époque courante et la racine de l'*arbre des époques*

([ε_{B2}, ε_0]) (iii) elle détermine la première intersection entre ces deux chemins (ε_0). Cette époque correspond au Plus Petit Ancêtre Commun (PPAC) entre l'époque de l'opération et l'époque courante. (iv) elle tronque les deux chemins au niveau du PPAC ([ε_{A2}] et [ε_{B2}]) (v) elle inverse l'ordre des époques du chemin entre l'époque courante et la racine ([ε_{B2}]) (vi) elle retourne les deux chemins obtenus ($\langle$[ε_{A2}], [ε_{B2}]$\rangle$).

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

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

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

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

Intégration d'une opération *rename* distante

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

Comme précédemment, nous utilisons l'exemple illustré dans la Figure 2.12 pour présenter le fonctionnement de cet algorithme.

La Figure 2.12 reprend le scénario décrit précédemment dans la Figure 2.11. Elle complète ce dernier en faisant apparaître la réception de l'opération *rename* vers l'époque ε_{B2} par le noeud A. C'est sur ce point que nous allons nous focaliser ici.

À la réception de l'opération *rename* vers l'époque ε_{B2}, le noeud A utilise RENREMOTE pour intégrer cette opération. Tout d'abord, le noeud A ajoute l'époque ε_{B2} et les métadonnées associées (ancien état, auteur de l'opération *rename*, numéro de séquence de l'auteur de l'opération *rename*) à son propre arbre des époques (ligne 6).

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

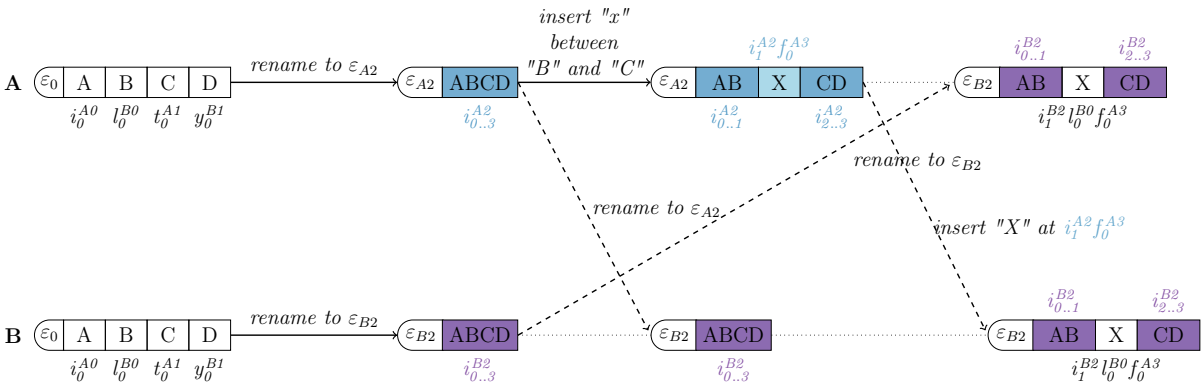
Algorithme 4 Algorithme d'intégration d'une opération *rename* distante

```
    function renRemote(seq, epochTree, currentEpoch, renOp)
        opEpoch ← getEpoch(renOp)
        renamedIds ← getRenamedIds(renOp)
        introducedEpoch ← getIntroducedpoch(renOp)
 5:
        newEpochTree ← addEpoch(epochTree, introducedEpoch, opEpoch, renamedIds)

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

            ⟨epochsToRevert, epochsToApply⟩ ← getPathBetweenEpochs(newEpochTree, currentEpoch,
    introducedEpoch)
15:
            for epoch in epochsToRevert do
                renamedIds ← getRenamedIds(newEpochTree, epoch)
                nId ← getNodeId(newEpochTree, epoch)
                nSeq ← getNodeSeq(newEpochTree, epoch)
20:             revertRenameId_partial ← papply(revertRenameId, renamedIds, nId, nSeq)
                ids ← map(ids, revertRenameId_partial)
            end for

            for epoch in epochsToApply do
25:             renamedIds ← getRenamedIds(newEpochTree, epoch)
                nId ← getNodeId(newEpochTree, epoch)
                nSeq ← getNodeSeq(newEpochTree, epoch)
                renameId_partial ← papply(renameId, renamedIds, nId, nSeq)
                ids ← map(ids, renameId_partial)
30:         end for

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

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

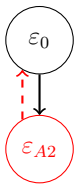
Dans le cas présent, on a $\varepsilon_{A2} <_\varepsilon \varepsilon_{B2}$. ε_{B2} devient donc la nouvelle époque courante. Le noeud A procède au renommage de son état vers cette nouvelle époque.

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

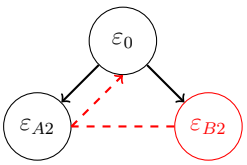
Le noeud obtient alors la liste des identifiants courant, à la nouvelle époque cible. Il ne



(a) Exécution nécessitant l'intégration d'opérations *rename* concurrentes



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



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

FIGURE 2.12 – Intégration d'une opération *rename* distante

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

2.3.5 Règles de récupération de la mémoire des états précédents

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

Pour proposer un tel mécanisme, nous nous reposons sur la notion de *stabilité causale des opérations* [9]. Une opération est causalement stable une fois qu'elle a été délivrée à tous les noeuds. Dans le contexte de l'opération *rename*, cela implique que tous les noeuds ont progressé à l'époque introduite par cette opération ou à une époque plus grande d'après la relation *priority*. À partir de ce constat, nous définissons les *potentielles époques courantes* :

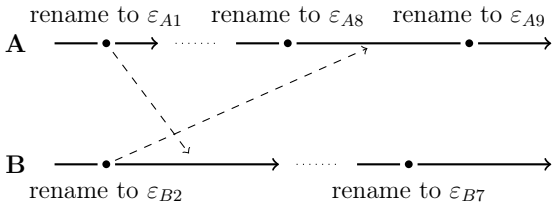
Définition 16 (Potentielles époques courantes) *L'ensemble des époques auxquelles les noeuds peuvent se trouver actuellement et à partir desquelles ils peuvent émettre des*

opérations, du point de vue du noeud courant. Il s'agit d'un sous-ensemble de l'ensemble des époques, composé de l'époque maximale introduite par une opération rename *causalement stable et de toutes les époques plus grande que cette dernière d'après la relation* priority.

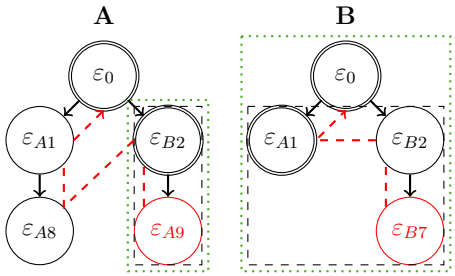
Pour traiter les prochaines opérations, les noeuds doivent maintenir les chemins entre toutes les époques de l'ensemble des *potentielles époques courantes*. Nous appelons *époques requises* l'ensemble des époques correspondant.

Définition 17 (Époques requises) *L'ensemble des époques qu'un noeud doit conserver pour traiter les potentielles prochaines opérations. Il s'agit de l'ensemble des époques qui forment les chemins entre chaque époque appartenant à l'ensemble des* potentielles époques courantes *et leur 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 2.13 illustre un cas d'utilisation du mécanisme de récupération de mémoire proposé.



(a) Exécution d'opérations *rename*



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

FIGURE 2.13 – Suppression des époques obsolètes et récupération de la mémoire des *anciens états* associés

Dans la Figure 2.13a, nous représentons une exécution au cours de laquelle deux noeuds A et B génère respectivement plusieurs opérations *rename*. Dans la Figure 2.13b, 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} comme époque courante (puisque $\varepsilon_{e0}\varepsilon_{A1} < \varepsilon_{e0}\varepsilon_{B2}$). Après, le noeud B génère une autre opération *rename* vers ε_{B7}.

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

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

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

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

Pour déterminer qu'une opération *rename* donnée est causalement stable, les noeuds doivent être conscients des autres et de leur avancement. Un protocole de gestion de groupe tel que [16, 15] 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

2.4.1 Complexité en temps des opérations

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

Hypothèses

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

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

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

La complexité de l'intégration de l'opération *insert* locale est inchangée par rapport à celle obtenue pour LogootSplit. Son intégration consiste toujours à déterminer entre quels identifiants se situe les nouveaux éléments insérés, à générer de nouveaux identifiants correspondants à l'ordre souhaité puis à insérer le bloc dans l'arbre AVL. D'après André et al. [5], nous obtenons donc une complexité de $\mathcal{O}(\log b)$ pour cette opération locale, où b représente le nombre de blocs dans la séquence.

La complexité de l'intégration de l'opération *insert* distante, elle, évolue par rapport à celle définie pour LogootSplit. Comme indiqué dans la section 2.3.4, plusieurs étapes se rajoutent au processus d'intégration de l'opération notamment dans le cas où celle-ci provient d'une autre époque que l'époque courante.

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

L'obtention de l'époque PPAC entre l'époque de l'opération et l'époque courante permet de déterminer les k renommages dont les effets doivent être retirés de l'opération et les

l renommages dont les effets doivent être intégrés à l'opération. Le noeud intégrant l'opération procède ainsi aux k inversions de renommages successives puis aux l application de renommages, et ce pour tous les s identifiants insérés par l'opération.

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

Pour inclure les effets des renommages à appliquer, le noeud utilise ensuite RENAMEID. De manière similaire à REVERTRENAMEID, RENAMEID 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 : $\mathcal{O}(h + s(k + l \cdot \log n + \log b))$.

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

Complexité en temps de l'opération *rename*

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

L'opération *rename* locale se décompose en 2 étapes : (i) La génération de l'*ancien état* à intégrer au message de l'opération (cf. Définition 11) (ii) Le remplacement de la séquence courante par une séquence équivalente, renommée. La première étape consiste à parcourir et à linéariser la séquence actuelle pour en extraire les intervalles d'identifiants. Elle s'effectue donc en $\mathcal{O}(b)$. La seconde consiste à instancier une nouvelle séquence vide, et à y insérer un bloc qui associe le contenu actuel de la séquence à l'intervalle d'identifiants $pos_{0..n-1}^{nodeId\ nodeSeq}$, avec *pos* la position du premier tuple du premier id de l'état, *nodeId* et *nodeSeq* l'identifiant et le numéro de séquence actuel du noeud et n la taille du contenu. Cette seconde étape s'effectue en $\mathcal{O}(1)$. L'opération *rename* locale a donc une complexité de $\mathcal{O}(b)$.

L'intégration de l'opération *rename* se décompose en les étapes suivantes : (i) L'insertion de la nouvelle époque et de l'*ancien état* associé dans l'*arbre des époques* (ii) La

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

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

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

Nous pouvons néanmoins améliorer ce premier résultat. Notamment, nous pouvons tirer parti des faits suivants : (i) Le fonctionnement de RENAMEID repose sur l'utilisation de l'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 $\mathcal{O}(h + n(k + l) + b)$ pour le traitement de l'opération *rename* distante.

Récapitulatif

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

Complexité en temps du mécanisme de récupération de mémoire des époques

Pour compléter notre analyse théorique des performances de RenamableLogootSplit, nous proposons une analyse en complexité en temps du mécanisme présenté en sous-section 2.3.5 qui permet de supprimer les époques devenues obsolètes et de récupérer la mémoire occupée par leur *ancien état* respectif.

L'algorithme du mécanisme de récupération de la mémoire se compose des étapes suivantes. Tout d'abord, il établit le vecteur de version des opérations causalement stables. Pour cela, chaque noeud doit maintenir une matrice des vecteurs de version de tous les noeuds. L'algorithme génère le vecteur de version des opérations causalement stable en récupérant pour chaque noeud la valeur minimale qui y est associée dans la matrice des

TABLE 2.1 – Complexité en temps des différentes opérations

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

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

vecteurs de version. Cette étape correspond à fusionner n vecteurs de version contenant n entrées, elle s'exécute donc en $\mathcal{O}(n^2)$ instructions.

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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 SEC.

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

Consommation mémoire

Nous avons ensuite procédé à l'évaluation de l'évolution de la consommation mémoire du document au cours des simulations, en fonction du CRDT utilisé et du nombre de *renaming bots*. Nous présentons les résultats obtenus dans la Figure 2.14.

Pour chaque graphique dans la Figure 2.14, nous représentons 4 données différentes. La ligne tiretée bleue correspond à la taille du contenu du document, c.-à-d. du texte,

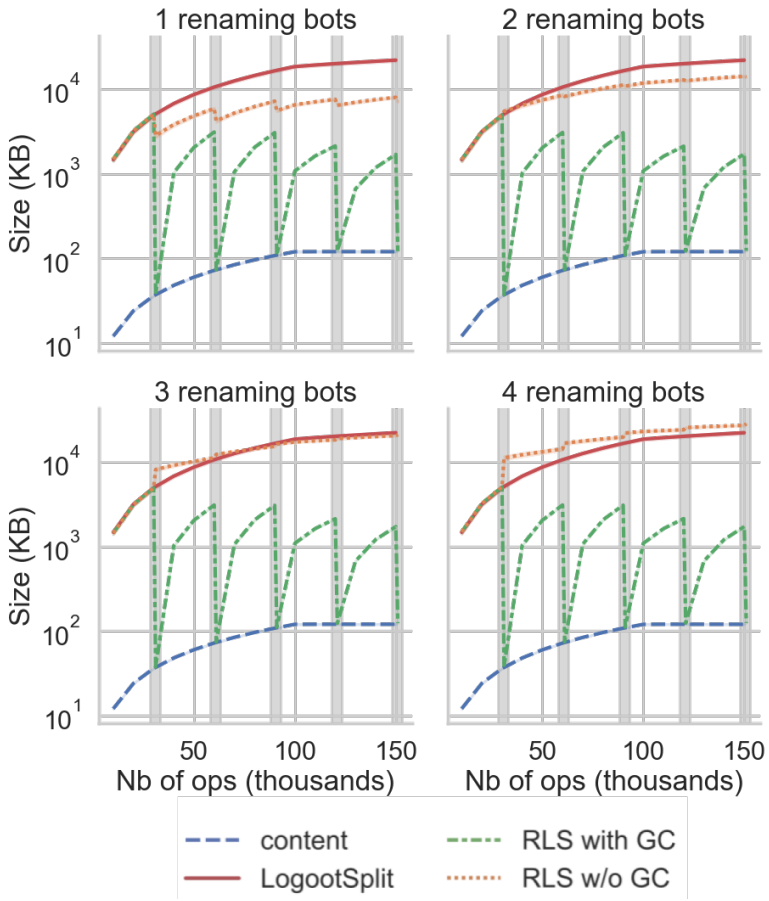


FIGURE 2.14 – Évolution de la taille du document en fonction du CRDT utilisé et du nombre de *renaming bots* dans la collaboration

tandis que la ligne continue rouge représente la taille complète du document LogootSplit.

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

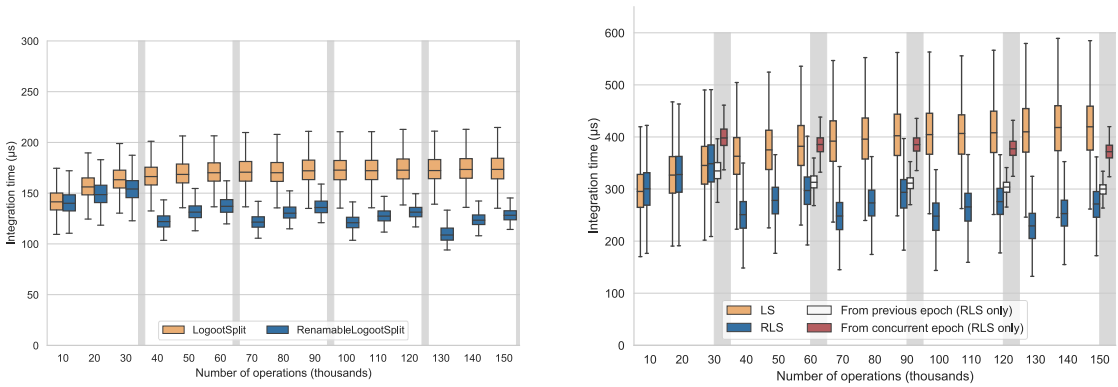
La ligne pointillée orange représente la taille du document RenamableLogootSplit dans son pire cas. Dans ce scénario, les noeuds considèrent que les opérations *rename* ne deviennent jamais causalement stables. Les noeuds doivent alors conserver de façon permanente les métadonnées introduites par le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

de la structure de données utilisée en interne pour représenter la séquence (c.-à-d. l'arbre AVL).

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

Nous avons ensuite comparé l'évolution du temps d'intégration des opérations standards, c.-à-d. les opérations *insert* et *remove*, sur des documents LogootSplit et RenamableLogootSplit. Puisque les deux types d'opérations partagent la même complexité en temps, nous avons seulement utilisé des opérations *insert* dans nos benchmarks. Nous faisons par contre la différence entre les mises à jours *locales* et *distantes*. Conceptuellement, les modifications locales peuvent être décomposées comme présenté dans [7] en les deux étapes suivantes : (i) la génération de l'opération correspondante (ii) l'application de l'opération correspondante sur l'état local. Cependant, pour des raisons de performances, nous avons fusionné ces deux étapes dans notre implémentation. Nous distinguons donc les résultats des modifications *locales* et des modifications *distantes* dans nos benchmarks. La Figure 2.15 présente les résultats obtenus.



(a) Modifications locales

(b) Modifications distantes

FIGURE 2.15 – Temps d'intégration des opérations standards

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 2.15b.

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

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

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

Temps d'intégration de l'opération de renommage

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

Nous 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 plus grande* (resp. *petite*) *époque* désignent les opérations *rename* qui introduisent une époque *soeur* de l'époque courante du noeud. D'après la relation *priority*, l'époque introduite est plus grande (resp. petite) que l'époque courante du noeud. Les résultats obtenus sont présentés dans le Tableau 2.3.

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 di-*

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. plus grande époque	30	644	644	16.6	620	683
	60	1318	1316	26.5	1263	1400
	90	1998	1994	46.6	1906	2112
	120	2240	2233	54	2144	2368
	150	2242	2234	63.5	2139	2351
Cc. int. plus petite époque	30	1.36	1.3	0.038	1.22	3.53
	60	2.82	2.69	0.476	2.43	4.85
	90	4.45	4.23	1.1	3.69	5.81
	120	5.33	5.1	1.34	4.42	8.78
	150	5.53	5.26	1.05	4.84	8.7

rectes 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). Il est donc nécessaire de prendre en compte ce résultat et de (i) concevoir des stratégies de génération des opérations *rename* pour éviter d'impacter négativement l'expérience utilisateur (ii) proposer des versions améliorées des algorithmes RENAMEID et REVERTRENAMEID pour réduire ces temps d'intégration :

— Au lieu d'utiliser RENAMEID, qui renomme l'état identifiant par identifiant, nous pourrions définir et utiliser RENAMEBLOCK. Cette fonction permettrait de renommer l'état bloc par bloc, offrant ainsi une 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 ma-

nière 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 log d'opérations

Afin de comparer les performances de RenamableLogootSplit et de LogootSplit de manière globale, nous avons mesuré le temps nécessaire pour un nouveau noeud pour rejouer l'entièreté du log d'opérations d'une session de collaboration, en fonction du nombre de *renaming bots* de la session. Nous présentons les résultats obtenus dans la Figure 2.16.

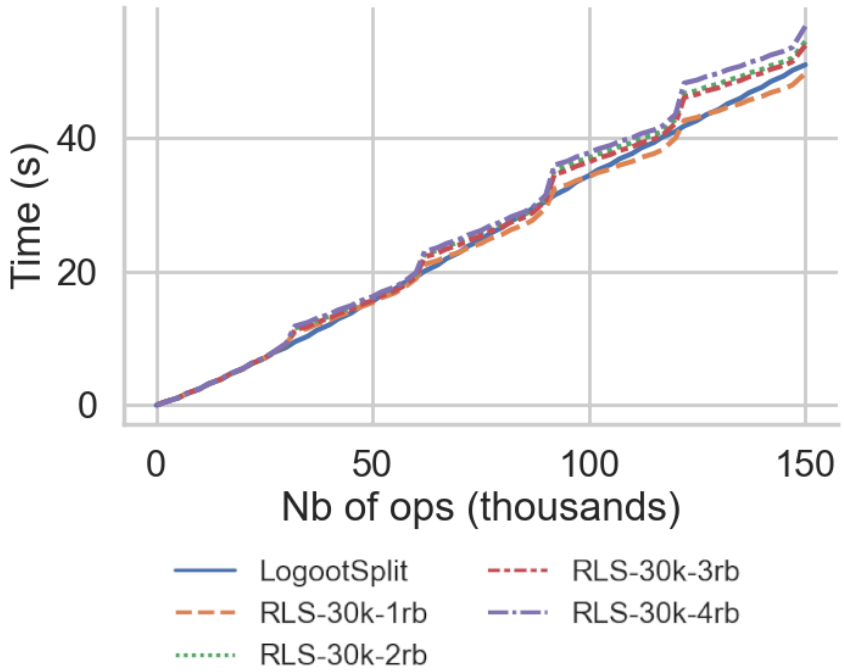


FIGURE 2.16 – Progression du nombre d'opérations du log rejouées en fonction du temps

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

Dans un cas réel d'utilisation, ce scénario (c.-à-d. rejouer l'entièreté du log) ne correspond pas au scénario principal et peut être mitigé, par exemple en utilisant un mécanisme de compression du log d'opérations. Dans la sous-section 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 *rename* sur les performances

Pour évaluer l'impact de la fréquence de l'opération *rename* sur les performances, nous avons réalisé un benchmark supplémentaire. Ce benchmark consiste à rejouer les logs d'opérations des simulations en utilisant divers CRDTs et configurations : LogootSplit, RenamableLogootSplit effectuant des opérations *rename* toutes les 30k opérations, RenamableLogootSplit effectuant des opérations *rename* toutes les 7.5k opérations. Au fur et à mesure que le benchmark rejoue le log des opérations, il mesure le temps d'intégration des opérations ainsi que leur taille. Les résultats de ce benchmark sont présentés dans le Tableau 2.4.

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

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

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

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

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

D'un autre côté, le coût des opérations *rename* est bien plus important (1000x) que celui des autres types d'opérations. Ceci s'explique par le fait que l'opération *rename* intègre l'*ancien état*, c.-à-d. la liste de tous les blocs composant l'état de la séquence au moment de la génération de l'opération. Cependant, nous observons le même phénomène pour les opérations *rename* que pour les autres opérations : la fréquence des opérations *rename* permet d'établir une borne pour la taille des opérations *rename*. Nous pouvons

donc choisir d'émettre fréquemment des opérations *rename* pour limiter leur taille respective. Ceci implique néanmoins un surcoût en computations pour chaque opération *rename* dans l'implémentation actuelle. Nous présentons une autre approche possible pour limiter la taille des opérations *rename* dans la sous-section 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 *rename*

Comme indiqué dans la sous-section 2.1.2, 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 particuliarités 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 2.14, les métadonnées augmentent de manière progressive jusqu'à représenter 99% de la structure de données. En utilisant les opérations *rename*, les noeuds peuvent supprimer les métadonnées et ainsi réduire la taille de la structure à un niveau acceptable. Pour déterminer quand générer des opérations *rename*, les noeuds peuvent donc monitorer le nombre d'opérations effectuées depuis la dernière opération *rename*, le nombre de blocs qui composent la séquence ou encore la taille des identifiants.

Un second aspect à prendre en compte est le temps d'intégration des opérations *rename*. Comme indiqué dans le Tableau 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. [24, 23] 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 2.14 montre que les opérations *rename* concurrentes accroissent la taille de la structure de données tandis que la Figure 2.16 illustre qu'elles augmentent le temps nécessaire pour rejouer le log d'opérations. La stratégie proposée doit donc viser à minimiser le nombre d'opérations *rename* concurrentes générées. Cependant, elle doit éviter d'utiliser des coordinations synchrones entre les noeuds pour cela (e.g. algorithmes de consensus), pour des raisons de performances. Pour réduire la probabilité de générer des opérations *rename* concurrentes, plusieurs méthodes peuvent être proposées. Par exemple, les noeuds peuvent monitorer à quels autres noeuds ils sont connectés actuellement et dé-

léguer au noeud ayant le plus grand *identifiant de noeud* la responsabilité de générer les opérations *rename*.

Pour récapituler, nous pouvons proposer plusieurs stragégies 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, [24] 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 *rename*

Pour limiter la consommation en bande passante des opérations *rename*, nous proposons la technique de compression suivante. Au lieu de diffuser les identifiants complets formant l'*ancien état*, les noeuds peuvent diffuser seulement les éléments nécessaires pour identifier de manière unique les intervalles d'identifiants. En effet, un identifiant peut être caractérisé de manière unique par le triplet composé de l'*identifiant de noeud*, du *numéro de séquence* et de l'*offset* de son dernier tuple. Par conséquent, un intervalle d'identifiants

peut être identifié de manière unique à partir du triplet signature de son identifiant de début et de sa longueur, c.-à-d. du quadruplet $\langle nodeId, nodeSeq, offsetBegin, offsetEnd\rangle$. Cette méthode nous permet de réduire les données à diffuser dans le cadre de l'opération *rename* à un montant fixe par intervalle.

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

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

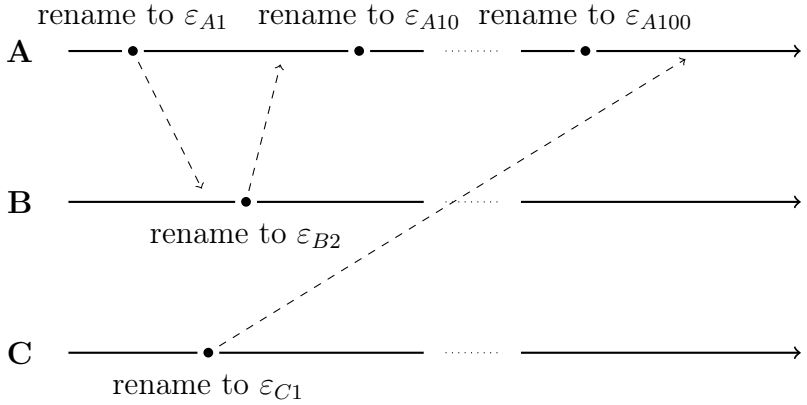
2.5.4 Définition de relations de priorité pour minimiser les traitements

Bien que la relation *priority* proposée dans la sous-section 2.3.2 est simple et garantit que tous les noeuds désignent la même époque comme époque cible, elle introduit un surcoût computationnel significatif dans certains cas. La Figure 2.17 présente un tel cas.

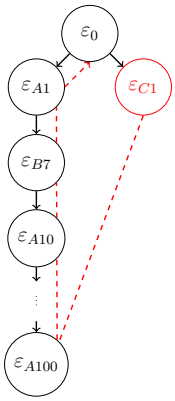
Dans cet exemple, les noeuds A et B éditent en collaboration un document. Au fur et à mesure de leur collaboration, ils effectuent plusieurs opérations *rename*. Cependant, après un nombre conséquent de modifications de leur part, un autre noeud C se reconnecte. Celui-ci leur transmet sa propre opération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

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

Afin d'offrir une plus grande flexibilité dans la conception de la relation *priority*, il est nécessaire de retirer la contrainte interdisant aux noeuds de rejouer une opération *rename*. Pour cela, un couple de fonctions réciproques doit être proposée pour RENAMEID



(a) Exécution d'opérations *rename* concurrentes



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

FIGURE 2.17 – Livraison d'une opération *rename* d'un noeud

et REVERTRENAMEID. Une solution alternative est de proposer une implémentation du mécanisme de renommage qui repose sur les identifiants originaux plutôt que sur ceux transformés, par exemple en utilisant le log des opérations.

2.5.5 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 délaient le renommage de leur état vers l'époque cible jusqu'à ce qu'ils aient obtenu un niveau de confiance donné en l'époque cible. Ce délai réduit la probabilité que les noeuds effectuent des traitements inutiles. Plusieurs stratégies peuvent être proposées pour calculer le niveau de confiance en l'époque cible. Ces stratégies peuvent reposer sur

une variété de métriques pour produire le niveau de confiance, tel que le temps écoulé depuis que le noeud a reçu une opération *rename* concurrente et le nombre de noeuds en ligne qui n'ont pas encore reçu l'opération *rename*.

Durant cette période d'incertitude introduite par le report, les noeuds peuvent recevoir des opérations provenant d'époques différentes, notamment de l'époque cible. Néanmoins, les noeuds peuvent toujours intégrer les opérations *insert* et *remove* en utilisant RENAMEID et REVERTRENAMEID au prix d'un surcoût computationnel pour chaque identifiant. Cependant, ce coût est négligeable (plusieurs centaines de microsecondes par identifiant d'après la Figure 2.15b) 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 log d'opérations

Lorsqu'un nouveau pair rejoint la collaboration, il doit tout d'abord récupérer l'état courant du document avant de pouvoir participer. Le nouveau pair utilise un mécanisme d'anti-entropie [42] pour récupérer l'ensemble des opérations via un autre pair. Puis il reconstruit l'état courant en appliquant successivement chacune des opérations. Ce processus peut néanmoins s'avérer coûteux pour les documents comprenant des milliers d'opérations.

Pour pallier ce problème, des mécanismes de compression du log ont été proposés dans la littérature. Les approches présentées dans [49, 22] consistent à remplacer un sous-ensemble des opérations du log par une opération équivalente, par exemple en aggrégeant les opérations *insert* adjacentes. Une autre approche, présentée dans [8], définie une relation *obsolete* sur les opérations. La relation *obsolete* permet de spécifier qu'une nouvelle opération rend obsolètes des opérations précédentes et permet de les retirer du log. Pour donner un exemple, une opération d'ajout d'un élément donné dans un OR-Set CRDT rend obsolètes toutes les opérations précédentes d'ajout et de suppression de cet élément.

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

Le mécanisme consiste à réduire le nombre d'opérations transmises à un nouveau pair rejoignant la collaboration grâce à l'opération *rename* de l'époque courante. L'opération *rename* ayant introduite l'époque courante fournit un état initial au nouveau pair. À partir

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

Notons que les données contenues dans l'opération *rename* telle que nous l'avons définie précédemment (cf. Définition 11) sont insuffisantes pour cette utilisation. En effet, les données incluses (*ancien état* au moment du renommage, identifiant du noeud auteur de l'opération *rename* et son numéro de séquence au moment de la génération) nous permettent seulement de recréer la structure de la séquence après le renommage. Mais le contenu de la séquence est omis, celui-ci n'étant jusqu'ici d'aucune utilité pour l'opération *rename*. Afin de pouvoir utiliser l'opération *rename* comme état initial, il est nécessaire d'y inclure cette information.

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

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

Définition 18 (rename enrichie) *Une opération* rename enrichie *est un quintuplet* ⟨*nodeId, nodeSeq, formerState, versionVector, content*⟩ *où*

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

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

Une solution possible dans ce cas de figure est de rediriger le second pair vers un troisième pour qu'il se synchronise avec lui. Cependant, cette solution pose des problèmes de latence/temps de réponse si le troisième pair s'avère indisponible à ce moment. Une autre approche possible est de généraliser le processus de synchronisation que nous avons présenté ici (opération *rename* comme état initial puis application des autres opérations)

à l'ensemble des pairs, et non plus seulement aux nouveaux pairs. Nous présentons les avantages et inconvénients de cette approche dans la sous-section suivante.

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

Nous avons décrit précédemment dans la section 2.3.4, et plus précisément dans l'Algorithme 4, le processus d'intégration de l'opération *rename* évaluée dans ce manuscrit. Pour rappel, le processus consiste à (i) identifier le chemin entre l'époque courante et l'époque cible (ii) appliquer les fonctions de transformations revertRenameId et renameId à l'ensemble des identifiants composant l'état courant (iii) re-créer une séquence à partir des nouveaux identifiants calculés et du contenu courant.

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

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

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

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

Cette approche présente plusieurs avantages par rapport à l'implémentation décrite dans la section 2.3.4. Tout d'abord, elle modifie le facteur du nombre de transformations à effectuer. La version décrite dans la section 2.3.4 transforme de l'époque courante vers l'époque cible chaque identifiant (ou chaque bloc si on dispose de renameBlock) de l'état courant. La version présentée ici effectue une transformation pour chaque opération du log concurrente à l'opération *rename* à intégrer. Le nombre de transformation peut donc être réduit de plusieurs ordres de grandeur avec cette approche, notamment si les opérations sont propagées aux pairs du réseau rapidement.

Un autre avantage de cette approche est qu'elle permet de récupérer et de réutiliser les identifiants originaux des opérations. Lorsqu'une suite de transformations est appliquée sur les identifiants d'une opération, elle est appliquée sur les identifiants originaux et non plus sur leur équivalents présents dans l'état courant. Ceci permet de réinitialiser les transformations appliquées à un identifiant et d'éviter le cas de figure mentionné dans la sous-section 2.3.3 : le cas où revertRenameId est utilisé pour retirer l'effet d'une opération *rename* sur un identifiant, avant d'utiliser renameId pour ré-intégrer l'effet de la même opération *rename*. Cette implémentation supprime donc la contrainte de

définir un couple de fonctions réciproques RENAMEID et REVERTRENAMEID, ce qui nous offre une plus grande flexibilité dans le choix de la relation $<_\varepsilon$ et du couple de fonctions RENAMEID et REVERTRENAMEID.

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

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

Finalement, cette approche implique aussi de parcourir le log des opérations à la recherche d'opérations concurrentes. Comme dit précédemment, la taille du log croît de façon monotone au fur et à mesure que les noeuds émettent des opérations. Cette étape du nouvel algorithme d'intégration de l'opération *rename* devient donc de plus en plus coûteuse. Des méthodes permettent néanmoins de réduire son coût computationnel. Notamment, chaque noeud traquent les informations de progression des autres noeuds afin de supprimer les métadonnées du mécanisme de renommage (cf. sous-section 2.3.5). Ces informations permettent de déterminer la stabilité causale des opérations et donc d'identifier les opérations qui ne peuvent plus être concurrentes à une nouvelle opération *rename*. Les noeuds peuvent ainsi maintenir, en plus du log complet des opérations, un log composé uniquement des opérations non stables causalement. Lors du traitement d'une nouvelle opération *rename*, les noeuds peuvent alors parcourir ce log réduit à la recherche des opérations concurrentes.

2.6 Comparaison avec les approches existantes

2.6.1 Core-Nebula

L'approche *core-nebula*[28, 60] a été proposée pour réduire la taille des identifiants dans Treedoc [43]. Dans ces travaux, les auteurs définissent l'opération *rebalance* qui permet aux noeuds de réassigner des identifiants plus courts aux éléments du document. Cependant, cette opération *rebalance* n'est ni commutative avec les opérations *insert* et *remove*, ni avec elle-même. Pour assurer la convergence à terme [51], l'approche *core-nebula* empêche

la génération d'opérations *rebalance* concurrentes. Pour ce faire, l'approche requiert un consensus entre les noeuds pour générer les opérations *rebalance*. Des opérations *insert* et *remove* sont elles toujours générées sans coordination entre les noeuds et peuvent donc être concurrentes aux opérations *rebalance*. Pour gérer les opérations concurrentes aux opérations *rebalance*, les auteurs proposent de transformer les opérations concernées par rapport aux effets des opérations *rebalance*, à l'aide un mécanisme de *catch-up*, avant de les appliquer.

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

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

Dans les systèmes disposant d'un *core*, nous pouvons donc combiner RenamableLogootSplit avec un protocole de consensus pour éviter la génération d'opérations *rename* concurrentes. Cette approche offre plusieurs avantages. Elle permet de se passer de tout ce qui à attrait au support d'opérations *rename* concurrentes, c.-à-d. la définition d'une relation *priority* et l'implémentation de revertRenameId. Elle permet aussi de simplifier l'implémentation du mécanisme de récupération de mémoire des époques et *anciens états* pour reposer seulement sur la stabilité causale des opérations. Concernant ses performances, cette approche se comporte de manière similaire à RenamableLogootSplit avec un seul *renaming bot* (cf. sous-section 2.4.3), mais avec un surcoût correspondant au coût du protocole de consensus sélectionné.

2.6.2 LSEQ

L'approche LSEQ [33, 32] est une approche visant à réduire la croissance des identifiants dans les Séquences CRDTs à identifiants densément ordonnés. Au lieu de réduire périodiquement la taille des métadonnées liées aux identifiants à l'aide d'un mécanisme de renommage coûteux, les auteurs définissent de nouvelles stratégies d'allocation des identifiants pour réduire leur vitesse de croissance. Dans ces travaux, les auteurs notent que la stratégie d'allocation des identifiants proposée dans Logoot [52] n'est adaptée qu'à un seul comportement d'édition : de gauche à droite, de haut en bas. Si les insertions sont effectuées en suivant d'autres comportements, les identifiants générés saturent rapidement l'espace des identifiants pour une taille donnée. Les insertions suivantes déclenchent alors une augmentation de la taille des identifiants. En conséquent, la taille des identifiants dans

Logoot augmente de façon linéaire au nombre d'insertions, au lieu de suivre la progression logarithmique attendue.

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

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

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

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 [45, 60], à l'exception de l'approche LSEQ [32]. En pratique, il serait intéressant d'étudier la mise en place d'un mécanisme d'éviction des noeuds inactifs pour répondre à ce problème.

La seconde limitation de RenamableLogootSplit concerne la génération d'opérations *rename* concurrentes. Chaque opération *rename* est coûteuse, aussi bien en terme de métadonnées à stocker et diffuser qu'en terme de traitements à effectuer. Il est donc important de chercher à minimiser le nombre d'opérations *rename* concurrentes émises par les noeuds. Une approche possible est d'adopter une architecture du type *core-nebula*[60]. Mais pour les systèmes incompatibles avec ce type d'architecture système, il serait intéressant de proposer d'autres approches ne nécessitant aucune coordination entre les noeuds. Mais par définition, ces approches ne pourraient offrir de garanties fortes sur le nombre d'opérations concurrentes possibles.

Annexe A

Algorithmes RENAMEID

Algorithme 5 Remaining functions to rename an identifier

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

        return concat(predNewFirstId, id)
    end if
end function

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

Annexe B

Algorithmes revertRenameId

Algorithme 6 Remaining functions to revert an identifier renaming

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

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

Bibliographie

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

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

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

[4] Paulo Sérgio Almeida, Ali Shoker et Carlos Baquero. « Efficient State-Based CRDTs by Delta-Mutation ». In : *Networked Systems*. Sous la dir. d'Ahmed Bouajjani et Hugues Fauconnier. Cham : Springer International Publishing, 2015, p. 62–76. isbn : 978-3-319-26850-7.

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

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

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

[8] Carlos Baquero, Paulo Sérgio Almeida et Ali Shoker. « Making Operation-Based CRDTs Operation-Based ». In : *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. PaPEC '14. Amsterdam, The Netherlands : Association for Computing Machinery, 2014. isbn : 9781450327169. doi : 10.1145/2596631.2596632. url : https://doi.org/10.1145/2596631.2596632.

[9] Carlos Baquero, Paulo Sérgio Almeida et Ali Shoker. « Making Operation-Based CRDTs Operation-Based ». In : *Distributed Applications and Interoperable Systems*. Sous la dir. de Kostas Magoutis et Peter Pietzuch. Berlin, Heidelberg : Springer Berlin Heidelberg, 2014, p. 126–140.

[10] 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.

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

[12] Mike Burmester et Yvo Desmedt. « A secure and efficient conference key distribution system ». In : *Advances in Cryptology — EUROCRYPT'94*. Sous la dir. d'Alfredo De Santis. Berlin, Heidelberg : Springer Berlin Heidelberg, 1995, p. 275–286. isbn : 978-3-540-44717-7.

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

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

[15] 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.

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

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

[18] 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.

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

[20] Victor Grishchenko et Mikhail Patrakeev. « Chronofold : A Data Structure for Versioned Text ». In : *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '20. Heraklion, Greece : Association for Computing Machinery, 2020. isbn : 9781450375245. doi : 10.1145/3380787.3393680. url : https://doi.org/10.1145/3380787.3393680.

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

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

[23] Claudia-Lavinia IGNAT et al. « 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 et al. 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.

[24] Claudia-Lavinia IGNAT et al. « 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.

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

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

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

[28] 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.

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

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

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

[32] Brice Nédelec, Pascal Molli et Achour Mostéfaoui. « A scalable sequence encoding for collaborative editing ». In : *Concurrency and Computation : Practice and Experience* (), e4108. doi : 10.1002/cpe.4108. eprint : https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4108. url : https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4108.

[33] Brice Nédelec et al. « 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.

[34] Hoang-Long Nguyen, Claudia-Lavinia Ignat et Olivier Perrin. « Trusternity : Auditing Transparent Log Server with Blockchain ». In : *Companion of the The Web Conference 2018*. Lyon, France, avr. 2018. doi : 10.1145/3184558.3186938. url : https://hal.inria.fr/hal-01883589.

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

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

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

[38] Matthieu Nicolas. « Efficient renaming in CRDTs ». In : *Middleware 2018 - 19th ACM/IFIP International Middleware Conference (Doctoral Symposium)*. Rennes, France, déc. 2018. url : https://hal.inria.fr/hal-01932552.

[39] 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.

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

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

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

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

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

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

[46] 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.

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

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

[49] 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.

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

[51] Douglas B Terry et al. « 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.

[52] Stéphane Weiss, Pascal Urso et Pascal Molli. « Logoot : A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks ». In : *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*. Montreal, QC, Canada : IEEE Computer Society, juin 2009, p. 404–412. DOI : 10.1109/ICDCS.2009.75. URL : http://doi.ieeecomputersociety.org/10.1109/ICDCS.2009.75.

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

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

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

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

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

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

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

[60] Marek Zawirski, Marc Shapiro et Nuno Preguiça. « Asynchronous rebalancing of a replicated tree ». In : *Conférence Française en Systèmes d'Exploitation (CFSE)*. Saint-Malo, France, mai 2011, p. 12. URL : https://hal.inria.fr/hal-01248197.