UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

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

THÈSE

présentée et soutenue publiquement le TODO : Définir une date

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Matthieu Nicolas

Composition du jury

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

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

Mis en page avec la classe thesul.

Remerciements

Les remerciements.

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

Sommaire

Table des figures

Introduction

Chapitre 1

État de l'art

Sommaire

1.1 Transformées opérationnelles

1.2 Séquences répliquées sans conflits

1.2.1 Types de données répliquées sans conflits

Principes

— Nouvelles spécifications des types de données existants
— Structures conçues pour être répliquées au sein d'un système
— Et être modifiées sans coordination par ses différents noeuds
— Doivent donc supporter de nouveaux scénarios uniquement possible dans des exécutions parallèles
— Et définir une sémantique pour ces scénarios inédits

  - Exemple du Registre avec LWW-Register et MV-Register ?
- Pour gérer ces scénarios, intègrent un mécanisme de résolution de conflits directement au sein de leur spécification
- Garantissent la cohérence forte à terme

Familles de types de données répliquées sans conflits

- Une catégorisation des CRDTs a été proposée
- Propose de répartir les CRDTs en différentes familles en fonction de la méthode de synchronisation utilisée
- Chacune de ces méthodes de synchronisation implique des contraintes sur la couche réseau du système et entraîne des répercussions sur la structure de données elle-même
- Types de données répliquées sans conflits à base d'états [31, 30]
  - Les noeuds partagent leur état de manière périodique
  - Une fonction *merge* permet aux noeuds de fusionner leur état courant avec un autre état reçu
  - Aucune hypothèse sur la partie réseau autre que les noeuds arrivent à communiquer à terme
  - Pas un problème si états perdus, les prochains intégreront les informations de ces derniers
  - Pas un problème si états reçus dans le désordre, la fonction *merge* est commutative
  - Pas un problème si états reçus plusieurs fois, *merge* est idempotent
  - Mais nécessite de conserver au sein de la structure de données assez d'informations pour proposer une telle fonction de *merge*
  - Par exemple, besoin de conserver une trace des éléments supprimés pour empêcher leur réapparition suite à une fusion d'états
  - *Matthieu: TODO : Ajouter forces, faiblesses et cas d'utilisation de cette approche*
- Types de données répliquées sans conflits à base d'opérations [31, 30, 7, 6]
  - Les noeuds partagent uniquement des opérations représentant leurs modifications
  - Une modification peut se formaliser en deux étapes
  - *prepare*, qui permet de générer une opération correspondant à une modification
  - *effect*, qui permet d'appliquer l'effet de la modification à un état
  - Les opérations concurrentes doivent être commutatives pour assurer la convergence
  - Mais pas de contraintes sur les opérations causalement liées

  - Pas de contraintes non plus sur l'idempotence des opérations
  - Nécessite donc généralement d'ajouter une couche *livraison* pour faire le lien entre le réseau et le CRDT
  - Permet d'attacher des informations de causalité aux opérations locales avant de les envoyer
  - Permet de ré-ordonner et filtrer les opérations distantes reçues avant de les fournir au CRDT
  - Besoin d'un mécanisme d'anti-entropie [26] pour assurer que l'ensemble des noeuds observent l'ensemble des opérations et ainsi garantir la convergence
  - Permet de lisser la consommation réseau
  - Offre des temps d'intégration et de propagation des modifications rapides
  - Mais accumule des métadonnées puisque les noeuds doivent conserver les opérations passées pour permettre à un nouveau noeud de rejoindre la collaboration et de se synchroniser
  - Possible de tronquer le log des opérations en se basant sur la stabilité causale [8] afin de limiter cette accumulation de métadonnées
- Types de données répliquées sans conflits à base de différences [3, 2]

Adoption dans la littérature et l'industrie

- Conception et développement de librairies mettant à disposition des développeurs d'applications des types de données composés [21, 20, 36, 16, 5]
- Conception de langages de programmation intégrant des CRDTs comme types primitifs, destinés au développement d'applications distribuées [18, 14]
- Conception et implémentation de bases de données distribuées, relationnelles ou non, privilégiant la disponibilité et la minimisation de la latence à l'aide des CRDTs [28, 11, 35, 10, 37]
- Conception d'un nouveau paradigme d'applications, Local-First Software, dont une des fondations est les CRDTs [17, 15]
- Éditeurs collaboratifs temps réel à large échelle et offrant de nouveaux scénarios de collaboration grâce aux CRDTs [19, 24]

1.2.2 Approches pour les séquences répliquées sans conflits

Approche à pierres tombales

- WOOT [25, 34, 1]
- RGA [29]
- RGASplit [9]

Approche à identifiants densément ordonnés

— Treedoc [27]

— Logoot [32, 33]

1.3 LogootSplit

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

1.3.1 Identifiants

Pour ce faire, LogootSplit assigne des identifiants composés d'une liste de tuples aux éléments. Ces tuples sont composés de quatre éléments : (i) une *position*, qui incarne la position souhaitée de l'élément (ii) un *identifiant de noeud*, (iii) un *numéro séquentiel de noeud* et (iv) un *offset*, qui sont combinés pour former des identifiants uniques. En comparant les identifiants en utilisant l'ordre lexicographique, LogootSplit est capable de déterminer la position d'un élément par rapport aux autres. Dans ce manuscrit, nous représentons les identifiants par le biais de la notation suivante : $position_{offset}^{node\_id\ node\_seq}$ où *position* est une lettre minuscule, *node_id* une lettre majuscule et *node_seq* et *offset* des entiers.

1.3.2 Aggrégation dynamique d'élements en blocs

Au lieu de stocker les identifiants de chaque élément de la séquence, LogootSplit propose d'aggréger de façon dynamique les éléments dans des blocs. Rassembler les éléments en blocs permet à LogootSplit d'assigner logiquement un identifiant à chaque élément, en utilisant un interval d'identifiants, tout en ne stockant réellement que la longueur du bloc et l'identifiant de son premier élément. LogootSplit regroupe les éléments avec des identifiants *contigus* dans un bloc. Nous appelons *contigus* deux identifiants qui sont identiques, sauf pour leur dernier *offset* et avec les deux *offsets* étant consécutifs. Nous représentons l'interval d'identifiants qui composent un bloc à l'aide du formalisme suivant : $position_{begin..end}^{node\_id\ node\_seq}$ où *begin* est l'offset du premier identifiant du bloc et *end* du dernier.

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

Cette fonctionnalité réduit le nombre d'identifiants stockés au sein de la structure de données, puisque les identifiants sont conservés à l'échelle des blocs plutôt qu'à l'échelle de chaque élément. Ceci permet de réduire de manière significative le surcoût en métadonnées de la structure de données.

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

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

1.3.3 Stratégie d'allocation

Matthieu: TODO : Ajouter et dérouler exemple où des noeuds insère dans une séquence répliquée pour illustrer la façon de choisir position, d'append à un bloc, ou de split un bloc

1.3.4 Limites

Comme indiqué précédemment, la taille des identifiants provenant d'un ordre total dense est variable. Quand les noeuds insèrent de nouveaux éléments entre deux autres ayant la même valeur de *position*, LogootSplit n'a pas d'autre choix que d'augmenter la taille de l'identifiant résultant. La Figure 1.2 illustre de tels cas. Dans cet exemple, puisque le noeud A insère un nouvel élément entre deux identifiants contigus i_0^{B0} et i_1^{B0}, LogootSplit ne peut pas générer un identifiant adapté de la même taille. Pour respecter l'ordre souhaité, LogootSplit génère un identifiant en ajoutant un nouveau tuple à l'identifiant du prédecesseur : $i_0^{B0} f_0^{A0}$.

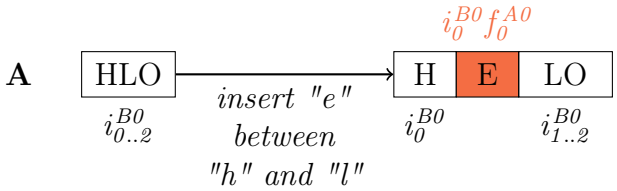


FIGURE 1.2 – Insertion leading to longer identifiers

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

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

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

les blocs à posteriori n'est fourni. L'efficacité de la structure décroît donc puisque chaque bloc entraîne un surcoût.

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

1.4 Mitigation du surcoût des séquences répliquées sans conflits

1.4.1 Core-Nebula

1.4.2 LSEQ

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

1.5 Synthèse

Chapitre 2

Présentation de l'approche

Sommaire

Nous proposons un nouveau Conflict-free Replicated Data Type (CRDT) pour la *Séquence* appartenant à l'approche des identifiants densément ordonnées : RenamableLogootSplit [22, 23]. Cette structure de données permet aux pairs d'insérer et de supprimer des éléments au sein d'une séquence répliquée. Nous introduisons une opération de renommage qui permet de 1. réassigner des identifiants plus courts aux différents éléments de la séquence 2. fusionner les blocs composant la séquence. Ces deux actions permettent à l'opération de renommage de produire un nouvel état minimisant son surcoût en métadonnées.

2.1 Modèle du système

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

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

Puisque RenamableLogootSplit est une extension de LogootSplit, il partage les mêmes contraintes sur la livraison de messages. La couche de livraison de messages sert donc à livrer les messages à l'application exactement une fois. La couche de livraison de messages a aussi pour tâche de garantir la livraison des opérations de suppression après les opérations d'insertion correspondantes. Aucune autre contrainte n'existe sur l'ordre de livraison des opérations. Finalement, la couche de livraison intègre aussi un mécanisme d'anti-entropie [26]. Ce mécanisme permet aux noeuds de se synchroniser par paires, en détectant et ré-échangeant les messages perdus.

2.2 Définition de l'opération de renommage

2.2.1 Objectifs

2.2.2 Propriétés

2.2.3 Contraintes

Chapitre 3

Renommage dans un système centralisé

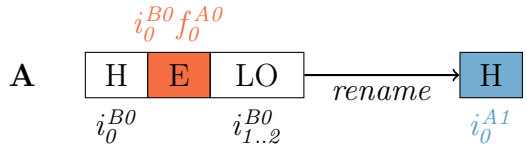
Sommaire

3.1 RenamableLogootSplit

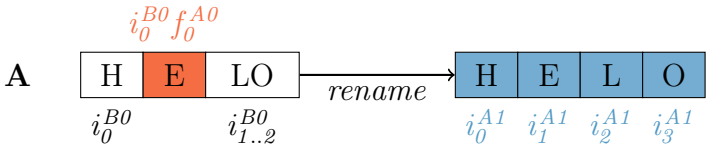
3.1.1 Opération de renommage proposée

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.

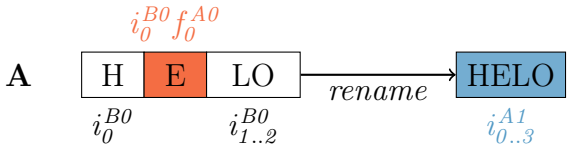
Son comportement est illustré dans la Figure 3.1. Dans cet exemple, le noeud A initie une opération *rename* sur son état local. Tout d'abord, le noeud A réutilise l'identifiant du premier élément de la séquence (i_0^{B0}) mais en le modifiant avec son propre identifiant de noeud (**A**) et numéro de 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



(a) Selecting the new identifier of the first element



(b) Selecting the new identifiers of the remaining ones



(c) Final state obtained

FIGURE 3.1 – Renaming the sequence on node A

dans 3.1a. 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 3.1b. 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 3.1c. 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. En utilisant cet état, les autres noeuds calculent le nouvel identifiant de chaque identifiant renommé. Concernant les identifiants insérés de manière concurrente au renommage, nous expliquons dans sous-section 3.1.2 comment les noeuds peuvent les renommer de manière déterministe.

3.1.2 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 3.2 illustre de tels cas.

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 3.1. 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,

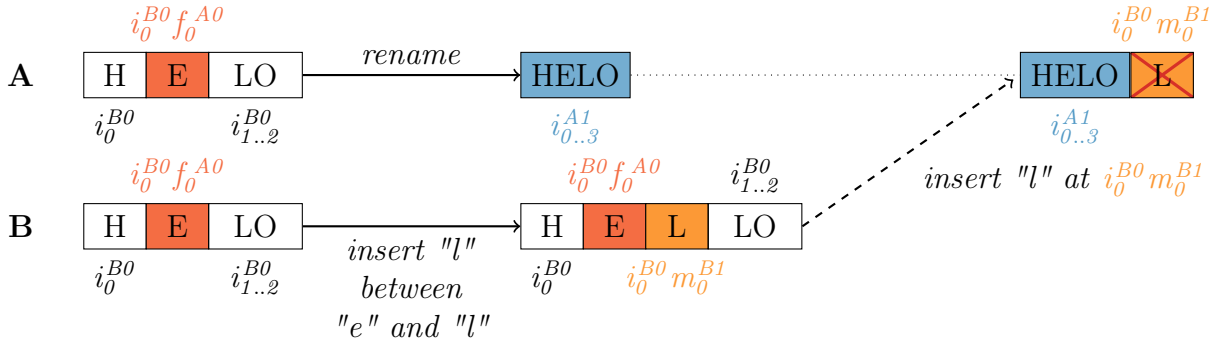


FIGURE 3.2 – Concurrent update leading to inconsistency

le noeud A insère le nouvel élément à la fin de sa séquence (puisque $i_3^{A1} < i_0^{B0} m_0^{B1}$) au lieu d'à sa position prévue. Comme décrit par cet exemple, appliquer naivement les modifications concurrentes provoquerait des anomalies. Il est donc nécessaire de traiter les opérations concurrentes aux opérations *rename* de manière particulière.

Tout d'abord, les noeuds doivent détecter les opérations concurrentes aux opérations *rename*. Pour cela, nous utilisons un système basé sur des *époques*. Initialement, la séquence répliquée débute à l'époque *origine* notée ε_0. Chaque opération *rename* introduit une nouvelle époque et permet aux noeuds d'y avancer depuis l'époque précédente. L'époque générée est caractérisée en utilisant l'identifiant du noeud et son numéro de séquence courant au moment de la génération de l'opération *rename*. Par exemple, l'opération *rename* décrite dans Figure 3.2 permet aux noeuds de faire progresser leur état de ε_0 à ε_{A1}.

Au fur et à mesure qu'ils reçoivent des opérations *rename*, les noeuds construisent et maintiennent localement la *chaîne des époques*, une structure de données ordonnant les époques en fonction de leur relation *parent-enfant*. De plus, les noeuds marquent chaque opération avec leur époque courante au moment de génération de l'opération. À la réception d'une opération, les noeuds compare l'époque de l'opération à la leur. Si les époques diffèrent, les noeuds doivent transformer l'opération avant de pouvoir l'appliquer. 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*. Pour ce faire, il est nécessaire d'ajouter la règle suivante aux contraintes existante sur la livraison des opérations : les opérations doivent désormais être livrées après l'opération *rename* qui a introduit leur époque.

Les noeuds utilisent la fonction RENAMEID, décrite dans Figure 3.3, pour transformer les opérations *insert* et *remove* par rapport aux opérations *rename*. Cet algorithme associe les identifiants d'une époque *parente* aux identifiants correspondant dans l'époque *enfant*. L'idée principale de cet algorithm 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 3.4. Cette figure décrit le même scénario que la Figure 3.2, à l'exception que le noeud A utilises 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 pré-

```
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 3.3 – Main functions to rename an identifier

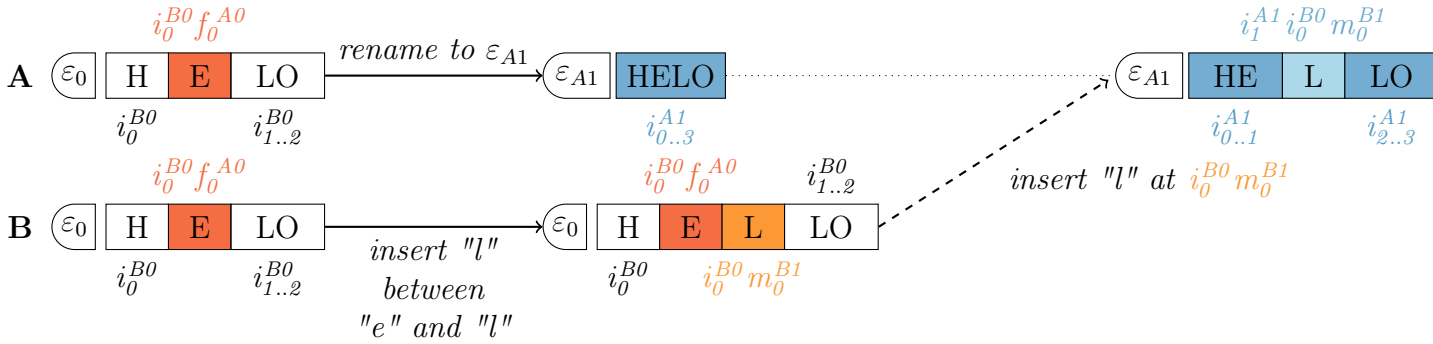


FIGURE 3.4 – Renaming concurrent update using RENAMEID before applying it to maintain intended order

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

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 modi-

fications concurrentes. Ce cas correspond à celui du noeud B dans la Figure 3.4. À la réception de l'opération *rename* du noeud A, le noeud B utilise RENAMEID sur chacun des identifiants de son état pour le renommer et atteindre un état équivalent à celui du noeud A.

Figure 3.3 présente seulement le cas principal de RENAMEID, c.-à-d. le cas où l'identifiant à renommer appartient à l'interval des identifiants formant l'ancien état ($firstId \leq id \leq lastId$). Les fonctions pour gérer les autres cas, c.-à-d. les cas où l'identifiant à renommer n'appartient pas à cet interval ($id < firstId$ ou $lastId < id$), sont présentées dans **??**.

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 temporelle 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.

3.1.3 Processus d'intégration d'une opération

3.1.4 Évolution du modèle de cohérence

Matthieu: TODO : Revoir si une livraison causale de l'opération rename *(et non epoch-based) peut avoir un intérêt*

3.1.5 Récupération de la mémoire des états précédents

3.2 Validation

3.2.1 Preuve de correction

3.2.2 Complexité temporelle

3.3 Discussion

3.3.1 Stockage des états précédents sur disque

3.3.2 Utilisation de l'opération de renommage comme snapshot

- L'opération de renommage embarque la somme de toutes les opérations passées sous la forme du *former state*
- On peut à tout moment re-calculer l'état courant du document à partir de l'opération de renommage primaire, des opérations concurrentes à cette dernière et des opérations générées depuis
- Peut utiliser ce principe pour le mécanisme de synchronisation

- Lorsqu'un nouveau pair rejoint la collaboration, le noeud avec lequel il se synchronise peut lui fournir uniquement ce sous-ensemble des opérations
- De la même manière, on pourrait généraliser l'utilisation de cette méthode de synchronisation
- À la réception d'une demande de synchronisation d'un pair présentant un important retard, le noeud peut choisir d'employer cette méthode
  - Plutôt que de lui envoyer et de lui faire rejouer l'ensemble des opérations
- La question étant de comment procéder pour quantifier ce retard et pour définir le seuil à partir duquel ce retard est considéré comme important
- Cette méthode de synchronisation pose néanmoins le problème suivant
- Le pair synchronisé de cette manière ne possède qu'une partie du log des opérations
- S'il reçoit ensuite une demande de synchronisation d'un autre pair, il est possible qu'il ne puisse y répondre
  - Cas où il manque à l'autre pair juste une opération d'avant le renommage (possible si les dépendances causales ne sont pas requises pour intégrer l'opération de renommage)
- Dans ce cas, ne peut pas fournir la seule opération manquante au pair qui la demande
  - *Matthieu: NOTE : Mais dans ce cas, le pair peut tout à fait générer un état courant à jour à partir des infos qu'il possède puisque l'opération qui lui manque est intégrée dans l'opé de renommage*
- *Matthieu: TODO : Étudier si y a un intérêt à privilégier la synchronisation basée sur l'intégration successive de toutes les opérations quand on a cette méthode de synchronisation par snapshot/checkpoint de possible*

3.3.3 Compression de l'opération de renommage

3.3.4 Limitation de la taille de l'opération de renommage

3.4 Conclusion

Chapitre 4

Renommage dans un système distribué

Sommaire

4.1 RenamableLogootSplit v2

4.1.1 Conflits en cas de renommages concurrents

Nous considérons à présent les scénarios avec des opérations *rename* concurrentes. Figure 4.1 développe le scénario décrit précédemment dans Figure 3.4.

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.

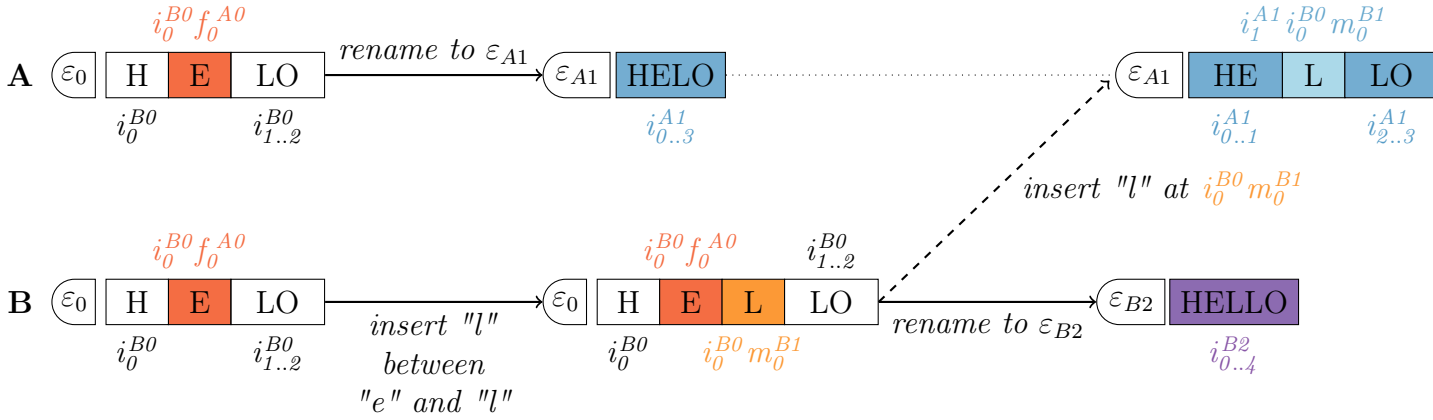


FIGURE 4.1 – Concurrent *rename* operations leading to divergent states

Puisque des époques concurrentes sont générées, les époques forment désormais l'*arbre des époques*. Nous représentons dans la Figure 4.2 l'*arbre des époques* que les neoeuds 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.

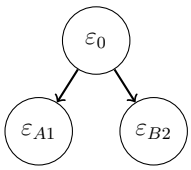


FIGURE 4.2 – The *epoch tree* corresponding to the scenario of Figure 4.1

À l'issue du scénario décrit dans la Figure 4.1, les noeuds A et B sont respectivement aux époques ε_{A1} et ε_{B2}. Pour converger, tous les noeuds devraient atteindre la même époque à terme. Cependant, la fonction RENAMEID décrite dans Figure 3.3 permet seulement au noeuds de progresser d'une époque *parente* à une de ses époques *enfants*. Le noeud A (resp. B) est donc dans l'incapacité de progresser vers l'époque du noeud B (resp. A). Il est donc nécessaire de faire évoluer notre mécanisme de renommage pour sortir de cette impasse.

Tout d'abord, les noeuds doivent se mettre d'accord sur une époque commune de l'*arbre des époques* comme époque cible. Afin d'éviter des problèmes de performances dûs à une coordination synchrone, les noeuds doivent sélectionner cette époque de manière non-coordonnée, c.-à-d. en utilisant seulement les données présentes dans l'*arbre des époques*. Nous présentons un tel mécanisme dans sous-section 4.1.2.

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

l'effet des opérations *rename* précédemment appliquées. Nous proposons un algorithme qui remplit cet objectif dans la sous-section 4.1.3.

4.1.2 Relation de priorité entre renommages

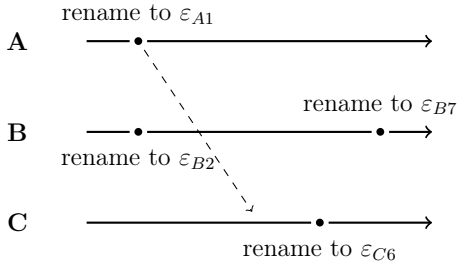
Pour que chaque noeud sélectionne la même époque cible de manière non-coordonnée, nous définissons la relation *priority*.

Définition 1 (Relation *priority*) *La relation* priority *est un ordre total strict sur l'ensemble des époques. Elle permet aux noeuds de comparer n'importe quelle paire d'époques.*

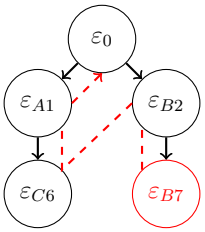
En utilisant la relation *priority*, nous définissons l'époque cible de la manière suivante :

Définition 2 (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 4.3 fournit un exemple.



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



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

FIGURE 4.3 – Selecting target epoch from execution with concurrent *rename* operations

La 4.3a 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 le problème qui nous occupe, seules ces opérations sont représentées dans cette figure. Initialement, le noeud A génère une opération *rename* qui introduit l'époque ε_{A1}. Cette opération est délivrée au noeud C, qui génère ensuite sa propre opération *rename* qui introduit l'époque ε_{C6}. De manière concurrente à ces opérations, le noeud B génère deux opérations *rename*, introduisant ε_{B2} et ε_{B7}.

Une fois que les noeuds se sont synchronisés, ils obtiennent l'*arbre des époques* représenté dans la 4.3b. Dans cette figure, la flèche pointillé rouge représente l'ordre entre les époques d'après la relation *priority* tandis que l'époque cible choisie est représentée sous la forme d'un noeud rouge.

Pour déterminer l'époque cible, les noeuds reposent sur la relation *priority*. D'après l'ordre lexicographique sur le chemin des époques dans l'*arbre des époques*, nous avons $\varepsilon_0 < \varepsilon_0\varepsilon_{A1} < \varepsilon_0\varepsilon_{A1}\varepsilon_{C6} < \varepsilon_0\varepsilon_{B2} < \varepsilon_0\varepsilon_{B2}\varepsilon_{B7}$. 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.

4.1.3 Algorithme d'annulation de l'opération de renommage

Nous présentons maintenant la fonction REVERTRENAMEID. Décrite dans Figure 4.4, cette fonction permet aux noeuds d'annuler une opération *rename* appliquée précédemment. Pour ce faire, REVERTRENAMEID associe les identifiants de l'époque *enfant* aux identifiants correspondant dans l'époque *parente*.

Les objectifs de REVERTRENAMEID sont les suivants : (i) Restaurer à leur ancienne valeur les identifiants générés causalement avant ou de manière concurrente à l'opération *rename* annulée (ii) Assigner de nouveaux identifiants respectant l'ordre souhaité aux éléments qui ont été insérés causalement après l'opération *rename* annulée. Nous illustrons son comportement à l'aide de la Figure 4.5.

Cette figure reprend le scénario de la Figure 3.2. Le noeud A reçoit l'opération *rename* du noeud B, qui est concurrente à l'opération *rename* que le noeud A a appliqué précédemment. Selon la relation *priority* proposée, le noeud A sélectionne l'époque introduite ε_{B2} comme l'époque cible. Il procède donc à ramener son état à un état équivalent à l'époque ε_0, le PPAC de son époque courante ε_{A1} et de l'époque cible ε_{B2}. Pour ce faire, il applique REVERTRENAMEID à chaque identifiant de son état courant.

REVERTRENAMEID détermine quelle stratégie appliquer pour restaurer un identifiant donné en utilisant des motifs. Par exemple, les identifiants de la forme $pos_{offset}^{nId\ nSeq}$ (i_{offset}^{A1} dans l'exemple courant) correspondent aux nouvelles valeurs des identifiants qui composent l'*ancien état*. Pour retrouver les identifiants d'origine, REVERTRENAMEID utilise simplement leur offset puisqu'il correspond à leur index dans l'*ancien état*.

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

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

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

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

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

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

Figure 4.4 – Main functions to revert an identifier renaming

queue comme identifiant résultant tout en préservant l'ordre souhaité, puisque sa valeur est comprise entre celles des identifiants du prédécesseur et du successeur.

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

Matthieu: TODO : Modifier exemple pour illustrer le cas de figure où on a besoin de

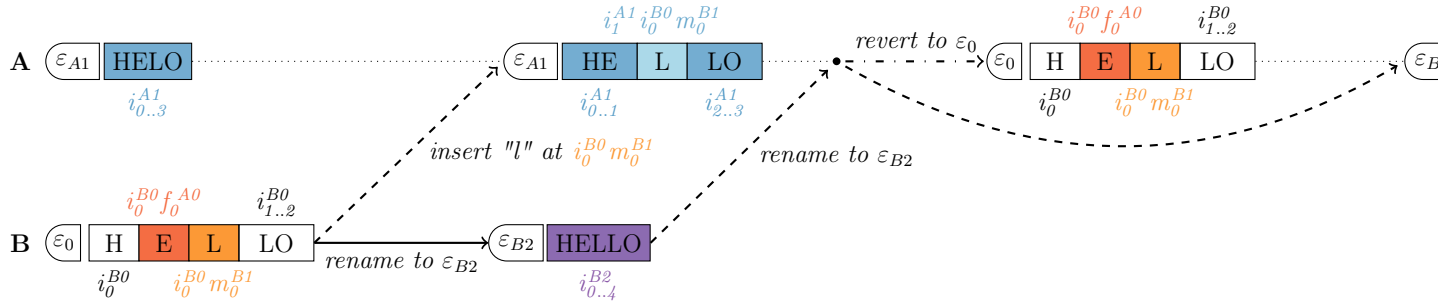


FIGURE 4.5 – Reverting a previously applied *rename* operation

MIN/MAX_ TUPLE

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

Comme pour Figure 3.3, Figure 4.4 ne présente seulement que le cas principal de REVERTRENAMEID. Il s'agit du cas où l'identifiant à restaurer appartient à l'interval des identifiants renommés $newFirstId \leq id \leq newLastId$). Les fonctions pour gérer les cas restants sont présentées dans **??**.

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

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

4.1.4 Mise à jour du processus d'intégration d'une opération

4.1.5 Mise à jour des 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.

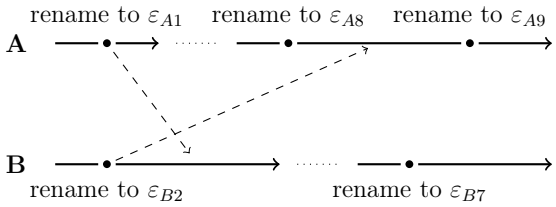
Pour proposer un tel mécanisme, nous nous reposons sur la notion de *stabilité causale des opérations* [8]. Une opération est causalement stable une fois qu'elle a été délivrée à tous les noeuds. Dans le contexte de l'opération *rename*, cela implique que tous les noeuds ont progressé à l'époque introduite 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 3 (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*.*

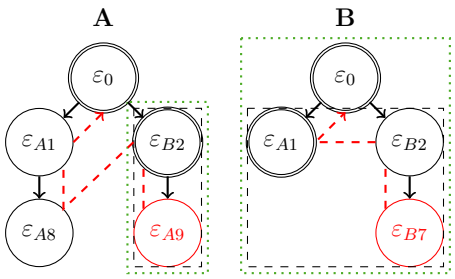
Pour traiter les prochaines opérations, les noeuds doivent maintenir les chemins entre toutes les époques de l'ensemble des *potentielles époques courantes*. Nous appelons *époques requises* l'ensemble des époques correspondant.

Définition 4 (Époques requises) *L'ensemble des époques qu'un noeud doit conserver pour traiter les potentielles prochaines opérations. Il s'agit de l'ensemble des époques qui forment les chemins entre chaque époque appartenant à l'ensemble des* potentielles époques courantes *et leur PPAC.*

Il s'ensuit que toute époque qui n'appartient pas à l'ensemble des *époques requises* peut être retirée par les noeuds. La Figure 4.6 illustre un cas d'utilisation du mécanisme de récupération de mémoire proposé.



(a) Execution



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

FIGURE 4.6 – Garbage collecting epochs and corresponding *former states*

Dans la 4.6a, nous représentons une exécution au cours de laquelle deux noeuds A et B génère respectivement plusieurs opérations *rename*. Dans la 4.6b, nous représentons les

arbre des époques respectifs de chaque noeud. Les époques introduites par des opérations *rename* causalement stables sont representées en utilisant des doubles cercles. L'ensemble des *potentielles époques courantes* est montré sous la forme d'un rectangle noir pointillé, tandis que l'ensemble des *époques requises* est représenté par un rectangle vert pointillé.

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

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

De manière concurrente, le noeud B génère l'opération *rename* vers ε_{B2}. Il reçoit ensuite l'opération *rename* vers ε_{A1} du noeud A. Cependant, le noeud B conserve ε_{B2} comme époque courante (puisque $\varepsilon_{e0}\varepsilon_{A1} < \varepsilon_{e0}\varepsilon_{B2}$). Après, le noeud B génère une autre opération *rename* vers ε_{B7}.

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

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

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

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

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

groupe tel que [13, 12] est donc requis.

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

4.2 Validation

4.2.1 Complexité temporelle

4.2.2 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 reproduisent le scénario suivant.

Scénario d'expérimentation

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

Implémentation des simulations

Nous avons effectué nos simulations avec les paramètres expérimentaux suivants : nous avons déployé 10 bots à l'aide de conteneurs Docker sur une même machine. Chaque conteneur correspond à un processus Node.js mono-threadé et permet de simuler un auteur. Les bots partagent et éditent de façon collaborative le document en utilisant soit LogootSplit soit RenamableLogootSplit en fonction de la session. Dans chaque cas, chaque bot génère localement une opération *insert* ou *remove* toutes les 200 ± 50ms et la diffuse immédiatement aux autres noeuds via un réseau P2P maillé. Au cours de la première phase, la probabilité d'émetter une opération *insert* (resp. *remove*) est de 80% (resp. 20%). Une fois que le document atteint 60k caractères (environ 15 pages), les bots passent à la seconde phase et mettent chaque probabilité à 50%. Après chaque opération locale, le bot peut

déplacer son curseur à une autre position aléatoire dans le document avec une probabilité de 5%. Chaque bot génère 15k opérations *insert* ou *remove* et s'arrête une fois qu'il a observé 150k opérations. Des instantanés de l'état du bot sont pris de façon périodique, toutes les 10k opérations observées.

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

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

4.2.3 Résultats

En utilisant les instantanés générés, nous avons effectué plusieurs benchmarks. Ces benchmarks évaluent les performances de RenamableLogootSplit et les compare à celles de LogootSplit. Les résultats sont présentés et analysés ci-dessous.

Convergence

Nous avons tout d'abord vérifié la convergence de l'état des noeuds à l'issue des simulations. Pour chaque simulation, nous avons comparé l'état final de chaque noeud à l'aide de leur instantanés respectifs. Nous avons pu confirmer que les noeuds convergaient sans aucune autre forme de communication que les opérations, satisfaisant donc le modèle de la Cohérence forte à terme (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 4.7.

Pour chaque graphique dans la Figure 4.7, nous représentons 4 données différentes. La ligne pointillée bleue correspond à la taille du contenu du document, c.-à-d. du texte, tandis que la ligne continue rouge représente la taille complète du document LogootSplit.

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

La ligne pointillée orange représente la taille du document RenamableLogootSplit dans son pire cas. Dans ce scénario, les noeuds considèrent que les opérations *rename* ne

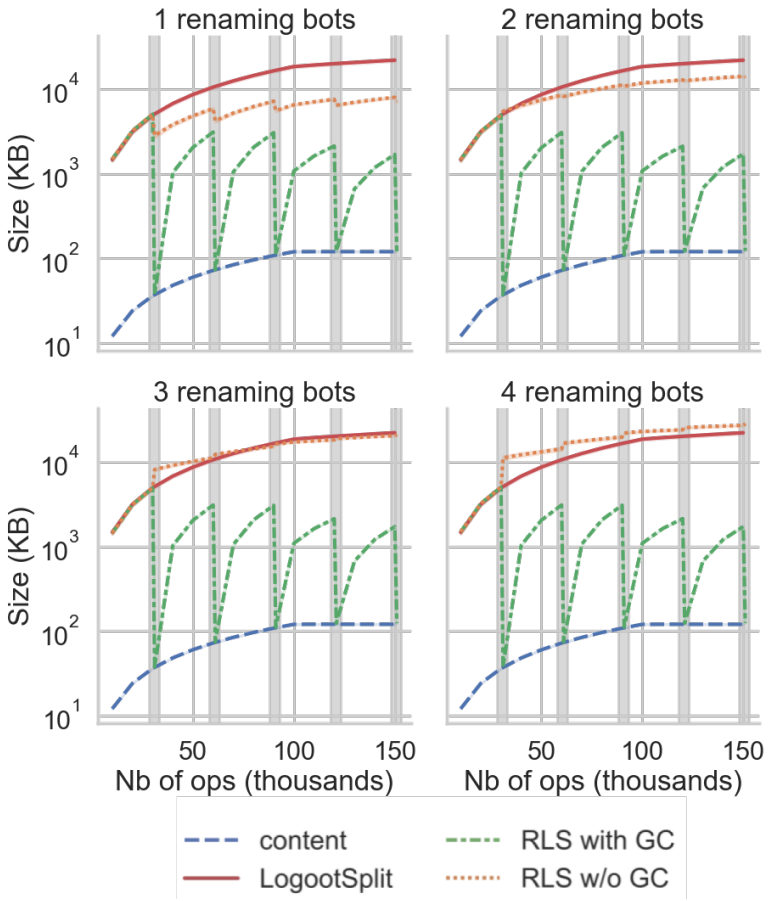


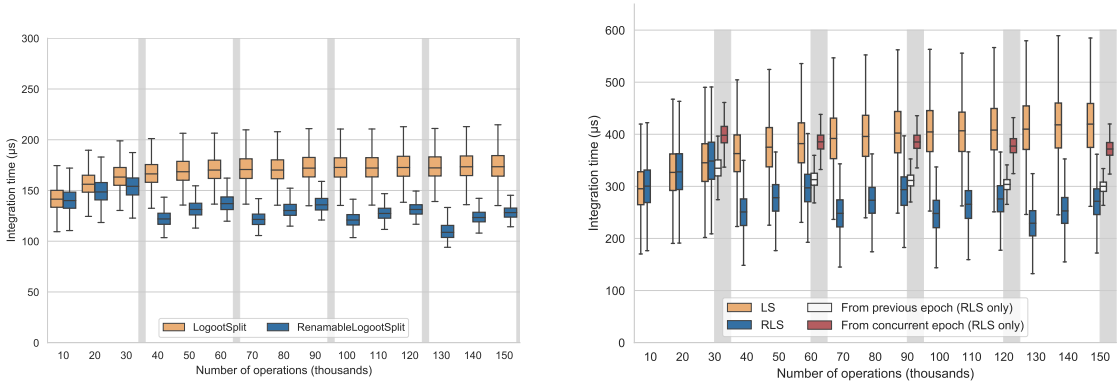
FIGURE 4.7 – Evolution of the size of the document

deviennent jamais causalement stables et qu'elles ne peuvent donc jamais être supprimées. Les noeuds doivent alors conserver de façon permanente les métadonnées introduites par le mécanisme de renommage. Les performances de RenamableLogootSplit diminuent donc à mesure que le nombre de *renaming bots* et d'opérations *rename* générées augmente. Néanmoins, nous observons que RenamableLogootSplit peut surpasser les performances de LogootSplit tant que le nombre de *renaming bots* reste faible (1 ou 2). Ce résultat s'explique par le fait que le mécanisme de renommage permet aux noeuds de supprimer les métadonnées de la structure de données utilisée en interne pour représenter la séquence.

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

Temps d'intégration des opérations standards

Nous avons ensuite comparé l'évolution du temps d'intégration des opérations standards, c.-à-d. les opérations *insert* et *remove*, sur des documents LogootSplit et RenamableLogootSplit. Puisque les deux types d'opérations partagent la même complexité temporelle, 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 [6] 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 4.8 présente les résultats obtenus.



(a) Local updates

(b) Remote updates

FIGURE 4.8 – Integration time of standard operations

Dans ces figures, les boxplots orange correspondent aux temps d'intégration sur des documents LogootSplit, les boxplots bleu sur des documents RenamableLogootSplit. Bien que les deux mesures soient initialement équivalentes, les temps d'intégration sur des documents RenamableLogootSplit sont ensuite réduits par rapport à ceux de LogootSplit une fois que des opérations *rename* ont été intégrées. Cette amélioration s'explique par le fait que l'opération *rename* optimise la représentation interne de la séquence.

Dans le cadre des opérations distantes, nous avons mesuré des temps d'intégration spécifiques à RenamableLogootSplit : le temps d'intégration d'opérations distantes provenant d'époques *parentes* et d'époques *soeurs*, respectivement affiché sous la forme de boxplots blanche et rouge dans la 4.8b.

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

Concernant les opérations provenant d'époques *soeurs*, nous observons un surcoût

additionnel puisque les noeuds doivent tout d'abord annulé les effets de l'opération *rename* concurrente en utilisant REVERTRENAMEID. À cause de cette étape supplémentaire, les performances de RenamableLogootSplit pour ces opérations sont comparables à celles de LogootSplit.

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

Temps d'intégration de l'opération de renommage

Finalement, nous avons mesuré l'évolution du temps d'intégration de l'opération *rename* en fonction du nombre d'opérations depuis l'opération *rename* précédente. Comme précédemment, nous distinguons les performances des modifications *locales* et *distantes*. Le cas des opérations *rename distantes* se sous-divise en trois catégories. Les opérations *distantes directes* désignent les opérations *rename distantes* qui introduisent une nouvelle époque *enfant* de l'époque courante du noeud. Les opérations *concurrentes introduisant une plus grande* (resp. *petite*) *époque* désigne les opérations *rename* qui introduisent une époque *soeur* de l'époque courante du noeud. D'après la relation *priority*, l'époque introduite est plus grande (resp. petite) que l'époque courante du noeud. Les résultats obtenus sont présentés dans le Tableau 4.1.

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

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

TABLE 4.1 – Integration time of rename operations

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

4.3 Discussion

4.3.1 Implémentation alternative à base d'operation-log

4.3.2 Définition de relations de priorité plus optimales

4.3.3 Report de la transition vers la nouvelle epoch principale

4.4 Conclusion

Chapitre 5

Stratégies de déclenchement du renommage

Sommaire

5.1 Motivation

5.2 Stratégies proposées

5.2.1 Propriétés

5.2.2 Stratégie 1 : ? ? ?

5.2.3 Stratégie 2 : ? ? ?

NOTE : Peut considérer une stratégie où on prend en compte la hauteur de l'epoch tree pour déclencher un renommage : peut attendre qu'on ait plus qu'une epoch pour autoriser un nouveau renommage d'avoir lieu. Permettrait d'empêcher le cas où un noeud revient après 6 mois d'absence et doit intégrer 100 renommages avant de pouvoir collaborer. Mais dans le cas où un noeud ne rejoint plus la collaboration, bloque le mécanisme de renommage pour les autres

5.3 Évaluation

5.4 Conclusion

Chapitre 6

Conclusions et perspectives

Sommaire

6.1 Résumé des contributions

6.2 Perspectives

6.2.1 Définition de relations de priorité plus optimales

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

6.2.3 Définition de types de données répliquées sans conflits plus complexes

Annexe A

Algorithmes

Index

Voici un index

Bibliographie

[1] 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.

[2] 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.

[3] 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.

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

Résumé

Afin d'assurer leur haute disponibilité, les systèmes distribués à large échelle se doivent de répliquer leurs données tout en minimisant les coordinations nécessaires entre noeuds. Pour concevoir de tels systèmes, la littérature et l'industrie adoptent de plus en plus l'utilisation de types de données répliquées sans conflits (CRDTs). Les CRDTs sont des types de données qui offrent des comportements similaires aux types existants, tel l'Ensemble ou la Séquence. Ils se distinguent cependant des types traditionnels par leur spécification, qui supporte nativement les modifications concurrentes. À cette fin, les CRDTs incorporent un mécanisme de résolution de conflits au sein de leur spécification.

Afin de résoudre les conflits de manière déterministe, les CRDTs associent généralement des identifiants aux éléments stockés au sein de la structure de données. Les identifiants doivent respecter un ensemble de contraintes en fonction du CRDT, telles que l'unicité ou l'appartenance à un ordre dense. Ces contraintes empêchent de borner la taille des identifiants. La taille des identifiants utilisés croît alors continuellement avec le nombre de modifications effectuées, aggravant le surcoût lié à l'utilisation des CRDTs par rapport aux structures de données traditionnelles. Le but de cette thèse est de proposer des solutions pour pallier ce problème.

Nous présentons dans cette thèse deux contributions visant à répondre à ce problème : (i) Un nouveau CRDT pour Séquence, RenamableLogootSplit, qui intègre un mécanisme de renommage à sa spécification. Ce mécanisme de renommage permet aux noeuds du système de réattribuer des identifiants de taille minimale aux éléments de la séquence. Cependant, cette première version requiert une coordination entre les noeuds pour effectuer un renommage. L'évaluation expérimentale montre que le mécanisme de renommage permet de réinitialiser à chaque renommage le surcoût lié à l'utilisation du CRDT. (ii) Une seconde version de RenamableLogootSplit conçue pour une utilisation dans un système distribué. Cette nouvelle version permet aux noeuds de déclencher un renommage sans coordination préalable. L'évaluation expérimentale montre que cette nouvelle version présente un surcoût temporaire en cas de renommages concurrents, mais que ce surcoût est à terme.

Mots-clés: CRDTs, édition collaborative en temps réel, cohérence à terme, optimisation mémoire, performance

Abstract

Keywords: CRDTs, real-time collaborative editing, eventual consistency, memory-wise optimisation, performance