

Chapitre 3

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

Sommaire

3.1	Couche interface utilisateur	64
3.2	Couche réplication	66
3.2.1	Modèle de données du document texte	66
3.2.2	Module de livraison des opérations	66
3.2.3	Collaborateurs	74
3.2.4	Curseurs	78
3.3	Couche réseau	78
3.3.1	Établissement d'un réseau P2P entre navigateurs	78
3.3.2	Topologie réseau	80
3.4	Couche sécurité	80
3.5	Conclusion	82

Les systèmes collaboratifs temps réels permettent à plusieurs utilisateurs de réaliser une tâche de manière coopérative. Ils permettent aux utilisateurs de consulter le contenu actuel, de le modifier et d'observer en direct les modifications effectuées par les autres collaborateurs. L'observation en temps réel des modifications des autres favorise une réflexion de groupe et permet une répartition efficace des tâches. L'utilisation des systèmes collaboratifs se traduit alors par une augmentation de la qualité du résultat produit[52, 25].

Plusieurs outils d'édition collaborative centralisés basés sur l'approche OT ont permis de populariser l'édition collaborative temps réel de texte[27, 23]. Ces approches souffrent néanmoins de leur architecture centralisée. Notamment, ces solutions rencontrent des difficultés à passer à l'échelle[32, 17] et posent des problèmes de confidentialité[24, 28].

L'approche CRDT offre une meilleure capacité de passage à l'échelle et est compatible avec une architecture P2P[2]. Ainsi, de nombreux travaux[41, 56, 55] ont été entrepris pour proposer une alternative distribuée répondant aux limites des éditeurs collaboratifs

centralisés. De manière plus globale, ces travaux s'inscrivent dans le nouveau paradigme d'application des *Local-First Softwares*[36, 30]. Ce paradigme vise le développement d'applications collaboratives, P2P, pérennes et rendant la souveraineté de leurs données aux utilisateurs.

De manière semblable, l'équipe Coast conçoit depuis plusieurs années des applications avec ces mêmes objectifs et étudient les problématiques de recherche liées. Elle développe Multi User Text Editor (MUTE)[51]^{2 3}, un éditeur collaboratif P2P temps réel chiffré de bout en bout. MUTE sert de plateforme d'expérimentation et de démonstration pour les travaux de l'équipe. Nous avons donc intégré dans MUTE les travaux de cette thèse.

La Figure 3.1 représente l'architecture système d'une collaboration utilisant MUTE. MUTE étant une application web, chaque noeud représenté ici correspond à un navigateur.

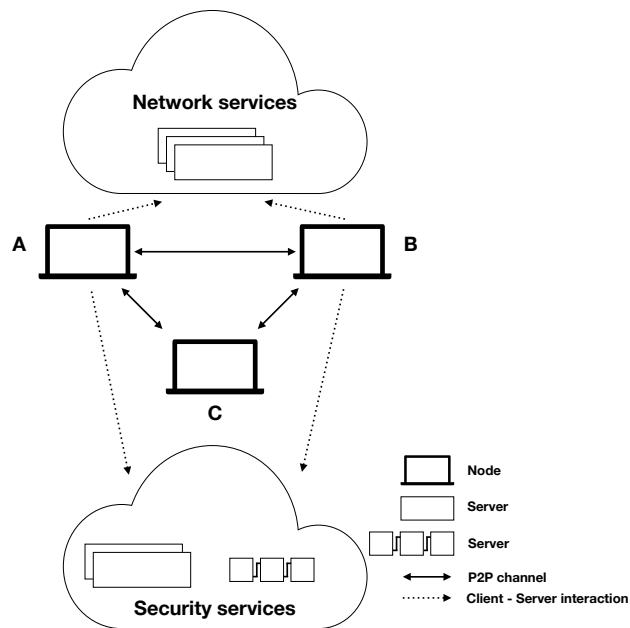


FIGURE 3.1 – Architecture système de l'application MUTE

Nous décrivons l'architecture logicielle d'un nœud dans la Figure 3.2. Dans ce chapitre, nous présentons les différentes couches logicielles de MUTE. Notamment, nous détaillons les couches correspondantes à l'implémentation des travaux présentés dans le chapitre 2. De la même façon, nous précisons dans ce chapitre les différents composants de l'architecture système de MUTE autre que les pairs eux-mêmes.

3.1 Couche interface utilisateur

La Figure 3.3 illustre l'interface utilisateur de l'éditeur de document de MUTE.

L'interface se compose d'un éditeur de texte supportant le langage de balisage Markdown. Ainsi, l'éditeur permet d'inclure plusieurs éléments légers de style. Les balises du

2. Disponible à l'adresse : <https://mutehost.loria.fr>

3. Code source disponible à l'adresse suivante : <https://github.com/coast-team/mute>

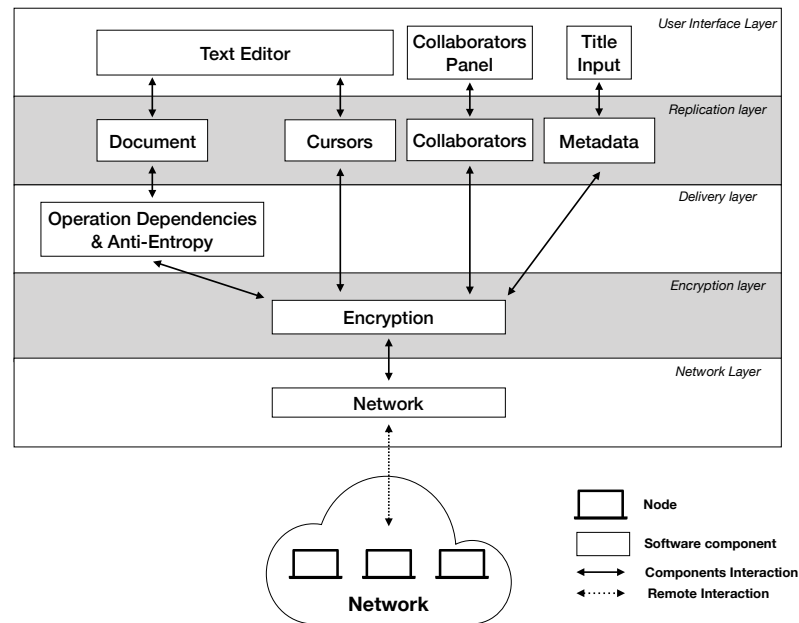


FIGURE 3.2 – Architecture logicielle de l'application MUTE

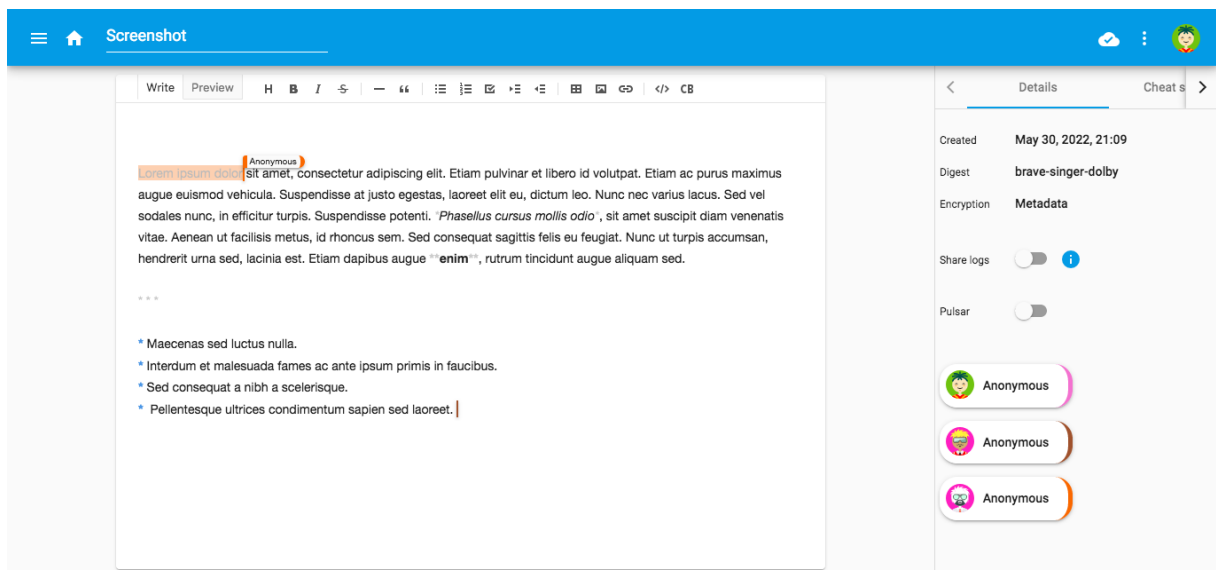


FIGURE 3.3 – Capture d'écran d'une session d'édition collaborative avec MUTE

langage Markdown étant du texte, elles sont répliquées nativement par la structure de données utilisée en interne par MUTE.

L'éditeur est agrémenté de plusieurs mécanismes permettant d'établir une conscience de groupe entre les collaborateurs. L'indicateur en haut à droite de la fenêtre représente le statut de connexion de l'utilisateur. Ceci lui indique s'il est actuellement connecté au réseau P2P, en cours de connexion, ou si un incident réseau a lieu.

MUTE affiche la liste des collaborateurs actuellement connectés sur la droite de l'éditeur. De plus, un curseur ou une sélection distante est associée à chaque collaborateur de

la liste. Elle permet d'indiquer à l'utilisateur courant dans quelles sections du document ses collaborateurs sont en train de travailler. Ainsi, ils peuvent se répartir la rédaction du document de manière implicite ou suivre facilement les modifications d'un collaborateur.

Les documents de l'utilisateur étant sauvegardés dans le navigateur, les documents sont aussi bien disponibles en étant en ligne que hors ligne. Une seconde page, listant les documents sauvegardés, permet à l'utilisateur de parcourir ses différents documents.

3.2 Couche réplication

3.2.1 Modèle de données du document texte

MUTE propose plusieurs alternatives pour représenter le document texte. MUTE permet de soit utiliser une implémentation de LogootSplit⁴, soit de RenamableLogootSplit⁴ ou soit de Dotted LogootSplit⁵. Ce choix est effectué via une valeur de configuration de l'application choisie au moment de son déploiement.

Le modèle de données utilisé interagit avec l'éditeur de texte par l'intermédiaire d'opérations textes. Lorsque l'utilisateur effectue des modifications locales, celles-ci sont détectées et mises sous la forme d'opérations textes. Elles sont transmises au modèle de données, qui les intègre alors à la structure de données répliquées. Le CRDT retourne en résultat l'opération distante à propager aux autres noeuds.

De manière complémentaire, lorsqu'une opération distante est délivrée au modèle de données, elle est intégrée par le CRDT pour actualiser son état. Le CRDT génère les opérations textes correspondantes et les transmet à l'éditeur de texte pour mettre à jour la vue.

En plus du texte, MUTE maintient un ensemble de métadonnées par document. Par exemple, les utilisateurs peuvent donner un titre au document. Pour représenter cette donnée additionnelle, nous associons un Last-Writer-Wins Register CRDT synchronisé par opérations[63] au document. De façon similaire, nous utilisons un First-Writer-Wins Register CRDT synchronisé par opérations pour représenter la date de création du document.

3.2.2 Module de livraison des opérations

Dans le cadre de LogootSplit et de RenamableLogootSplit, le modèle de données utilisé pour représenter le document texte est couplé au composant **Sync**. Le rôle de ce composant est d'assurer le respect du modèle de livraison des opérations au CRDT. Pour cela, le module **Sync** doit implémenter les contraintes présentées dans la Définition 9 et dans la Définition 13.

4. Les deux implémentations proviennent de la librairie `mute-structs` : <https://github.com/coast-team/mute-structs>

5. Implémentation fournie par la librairie suivante : <https://github.com/coast-team/dotted-logootsplit>

Livraison des opérations en exactement un exemplaire

Afin de respecter la contrainte de *exactly-once delivery*, il est nécessaire d'identifier de manière unique chaque opération. Pour cela, le module **Sync** ajoute un *Dot*[5] à chaque opération :

Définition 19 (Dot) *Un Dot est une paire $\langle nodeId, nodeSyncSeq \rangle$ où*

- *$nodeId$ est l'identifiant unique du noeud qui a généré l'opération.*
- *$nodeSyncSeq$ est le numéro de séquence courant du noeud à la génération de l'opération.*

Il est à noter que *nodeSyncSeq* est différent du *nodeSeq* utilisé dans LogootSplit et RenamableLogootSplit (cf. Définition 6). En effet, *nodeSyncSeq* se doit d'augmenter à chaque opération tandis que *nodeSeq* n'augmente qu'à la création d'un nouveau bloc. Les contraintes étant différentes, il est nécessaire de distinguer ces deux données.

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

Plusieurs structures de données sont adaptées pour maintenir l'ensemble des opérations reçues. Dans le cadre de MUTE, nous avons choisi d'utiliser un vecteur de versions. Cette structure nous permet de réduire à un dot par noeud le surcoût en métadonnées du module **Sync**, puisqu'il ne nécessite que de stocker le dot le plus récent par noeud. Cette structure permet aussi de vérifier en temps constant si une opération est déjà connue. La Figure 3.4 illustre son fonctionnement.

Dans cet exemple, qui reprend celui de la Figure 1.3, deux noeuds A et B répliquent une séquence. Initialement, celle-ci contient les éléments "OGNON". Ces éléments ont été insérés un par un par le noeud A, donc par le biais des opérations *a1* à *a5*. Le module **Sync** de chaque noeud maintient donc initialement le vecteur de version $\langle A : 5 \rangle$.

Le noeud A insère l'élément "I" entre les éléments "O" et "G". Cette modification est alors labellisée *a6* par son module **Sync** et est envoyée au noeud B. À la réception de cette opération, le module **Sync** de B compare son dot avec son vecteur de version local. L'opération *a6* étant la prochaine opération attendue de A, celle-ci est acceptée : elle est alors délivrée au CRDT et le vecteur de version est mis à jour.

Le noeud B supprime ensuite l'élément nouvellement inséré. S'agissant de la première modification de B, cette modification *b1* ajoute l'entrée correspondante dans le vecteur de version $\langle A : 6, B : 1 \rangle$. L'opération est envoyée au noeud A. Cette opération étant la prochaine opération attendue de B, elle est acceptée et délivrée.

Finalement, le noeud B reçoit de nouveau l'opération *a6*. Son module **Sync** détermine alors qu'il s'agit d'un doublon : l'opération apparaît déjà dans le vecteur de version $\langle A : 6, B : 1 \rangle$. L'opération est donc ignorée, et la résurgence de l'élément "I" illustrée dans la Figure 1.3 est évitée.

Il est à noter que dans le cas où un noeud reçoit une opération avec un dot plus élevé que celui attendu (e.g. le noeud A reçoit une opération *b3* à la fin de l'exemple), cette

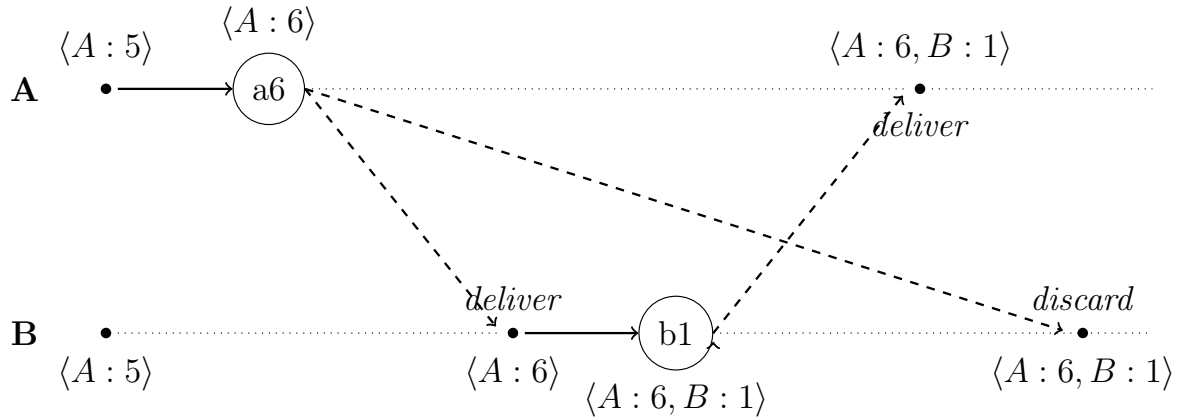
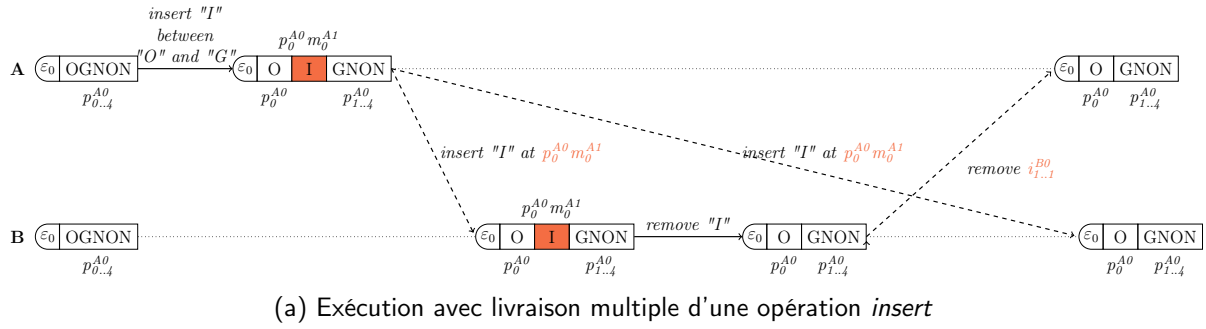


FIGURE 3.4 – Gestion de la livraison *exactly-once* des opérations

opération est mise en attente. En effet, délivrer cette opération nécessiterait de mettre à jour le vecteur de version à $\langle A : 6, B : 3 \rangle$ et masquerait le fait que l'opération *b2* n'a jamais été reçue. L'opération *b3* serait donc mise en attente jusqu'à la livraison de l'opération *b2*.

Ainsi, l'implémentation de livraison *exactly-once* avec un vecteur de version comme structure de données force une livraison First In, First Out (FIFO) des opérations par noeuds. Il s'agit d'une contrainte non-nécessaire et qui peut introduire des délais dans la collaboration, notamment si une opération d'un noeud est perdue par le réseau. Nous jugeons cependant acceptable ce compromis entre le surcoût du mécanisme de livraison *exactly-once* et son impact sur l'expérience utilisateur.

Pour retirer cette contrainte superflue, il est possible de remplacer cette structure de données par un *Interval Version Vector* [40]. Au lieu d'enregistrer seulement le dernier dot observé par noeud, cette structure de données enregistre les intervalles de dots observés. Ceci permet une livraison *out of order* des opérations tout en garantissant une livraison *exactly-once* et en compactant efficacement les données stockées par le module **Sync** à terme.

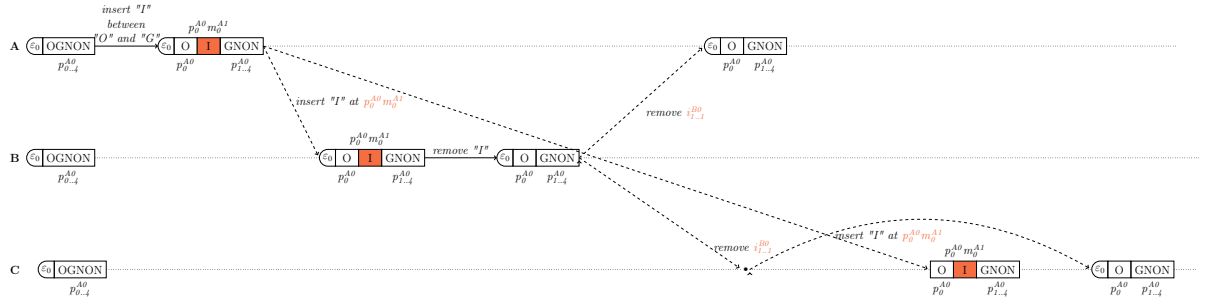
Livraison de l'opération *remove* après l'opération *insert*

La seconde contrainte que le modèle de livraison doit respecter spécifie qu'une opération *remove* doit être délivrée après les opérations *insert* insérant les éléments concernés.

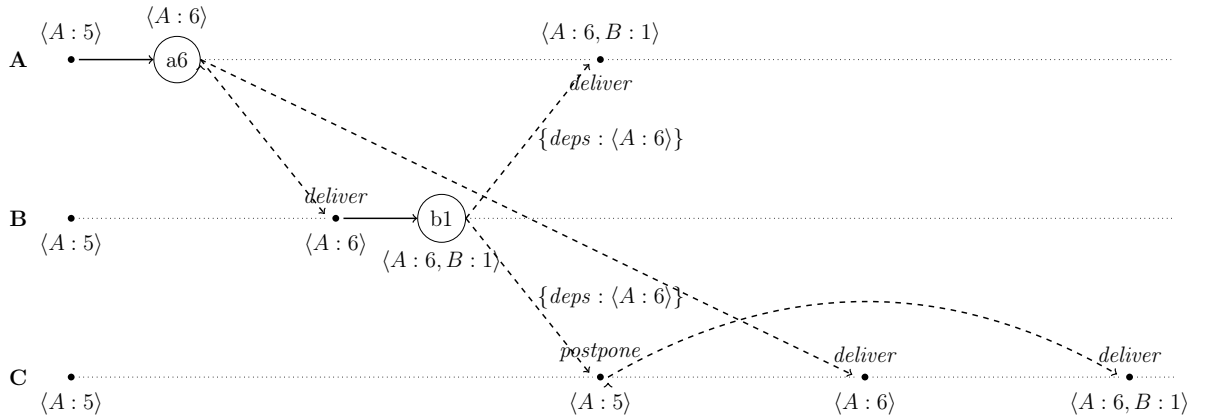
Pour cela, le module **Sync** ajoute un ensemble *Deps* à chaque opération *remove* avant de la diffuser :

Définition 20 (Deps) *Deps* est un ensemble d'opérations. Il représente l'ensemble des opérations dont dépend l'opération *remove* et qui doivent donc être livrées au préalable.

Plusieurs structures de données sont adaptées pour représenter les dépendances de l'opération *remove*. Dans le cadre de MUTE, nous avons choisi d'utiliser un ensemble de dots : pour chaque élément supprimé par l'opération *remove*, nous identifions le noeud l'ayant inséré et nous ajoutons le dot correspondant à l'opération la plus récente de ce noeud à l'ensemble des dépendances. Cette approche nous permet de limiter à un dot par élément supprimé le surcoût en métadonnées des dépendances et de les calculer en un temps linéaire par rapport au nombre d'éléments supprimés. Nous illustrons le calcul et l'utilisation des dépendances de l'opération *remove* à l'aide de la Figure 3.5.



(a) Exécution avec livraison dans le désordre d'une insertion et de sa suppression



(b) État et comportement de la couche Sync au cours de l'exécution décrite en Figure 3.5a

FIGURE 3.5 – Gestion de la livraison *causale-remove* des opérations

Cet exemple reprend et complète celui de la Figure 3.5. Trois noeuds A, B et C répliquent et éditent collaborativement une séquence. Les trois noeuds partagent le même

état initial : une séquence contenant les éléments "OGNON" et un vecteur de version $\langle A : 5 \rangle$.

Le noeud A insère l'élément "I" entre les éléments "O" et "G". Cet élément se voit attribué l'identifiant $p_o^{A0} m_o^{A1}$. L'opération correspondante $a6$ est diffusée aux autres noeuds.

À la réception de cette dernière, le noeud B supprime l'élément "I" nouvellement inséré et génère l'opération $b1$ correspondante. Comme indiqué précédemment, l'opération $b1$ étant une opération *remove*, le module **Sync** calcule ses dépendances avant de la diffuser. Pour chaque élément supprimé ("I"), **Sync** récupère l'identifiant de l'élément ($p_o^{A0} m_o^{A1}$) et en extrait l'identifiant du noeud qui l'a inséré (A). **Sync** ajoute alors le dot de l'opération la plus récente reçue de ce noeud ($\langle A : 6 \rangle$) à l'ensemble des dépendances de l'opération. L'opération est ensuite diffusée.

À la réception de l'opération $b1$, le noeud A vérifie s'il possède l'ensemble des dépendances de l'opération. Le noeud A ayant déjà observé l'opération $a6$, le module **Sync** délivre l'opération $b1$ au CRDT.

À l'inverse, lorsque le noeud C reçoit l'opération $b1$, il n'a pas encore reçu l'opération $a6$. L'opération $b1$ est alors mise en attente. À la réception de l'opération $a6$, celle-ci est délivrée. Le module **Sync** ré-évalue alors le cas de l'opération $b1$ et détermine qu'elle peut à présent être délivrée.

Il est à noter que notre approche pour générer l'ensemble des dépendances est une approximation. En effet, nous ajoutons les dots des opérations les plus récentes des auteurs des éléments supprimés. Nous n'ajoutons pas les dots des opérations qui ont spécifiquement insérés les éléments supprimés. Pour cela, il serait nécessaire de parcourir le log des opérations à la recherche des opérations *insert* correspondante. Cette méthode serait plus coûteuse, sa complexité dépendant du nombre d'opérations dans le log d'opérations, et incompatible avec un mécanisme tronquant le log des opérations en utilisant la stabilité causale. Notre approche introduit un potentiel délai dans la livraison d'une opération *remove* par rapport à une livraison utilisant ses dépendances exactes, puisqu'elle va reposer sur des opérations plus récentes et potentiellement encore inconnues par le noeud. Mais il s'agit là aussi d'un compromis que nous jugeons acceptable entre le surcoût du mécanisme de livraison et l'expérience utilisateur.

Livraison des opérations après l'opération *rename* introduisant leur époque

La troisième contrainte spécifiée par le modèle de livraison est qu'une opération doit être délivrée après l'opération *rename* qui a introduite son époque de génération.

Pour cela, le module **Sync** doit donc récupérer l'époque courante de la séquence répliquée, récupérer le dot de l'opération *rename* l'ayant introduite et l'ajouter en tant que dépendance de chaque opération. Cependant, dans notre implémentation, le module **Sync** et le module représentant la séquence répliquée sont découplés et ne peuvent interagir directement l'un avec l'autre.

Pour remédier à ce problème, le module **Sync** maintient une structure supplémentaire : un vecteur des dots des opérations *rename* connues. À la réception d'une opération *rename* distante, l'entrée correspondante de son auteur est mise à jour avec le dot de la nouvelle époque introduite. À la génération d'une opération locale, l'opération est examinée pour récupérer son époque de génération. **Sync** conserve alors seulement l'entrée correspondante

dans le vecteur des dots des opérations *rename*. À ce stade, le contenu du vecteur est ajouté en tant que dépendance de l'opération. Ensuite, si l'opération locale s'avère être une opération *rename*, le vecteur est modifié pour ne conserver que le dot de l'époque introduite par l'opération. La Figure 3.6 illustre ce fonctionnement.

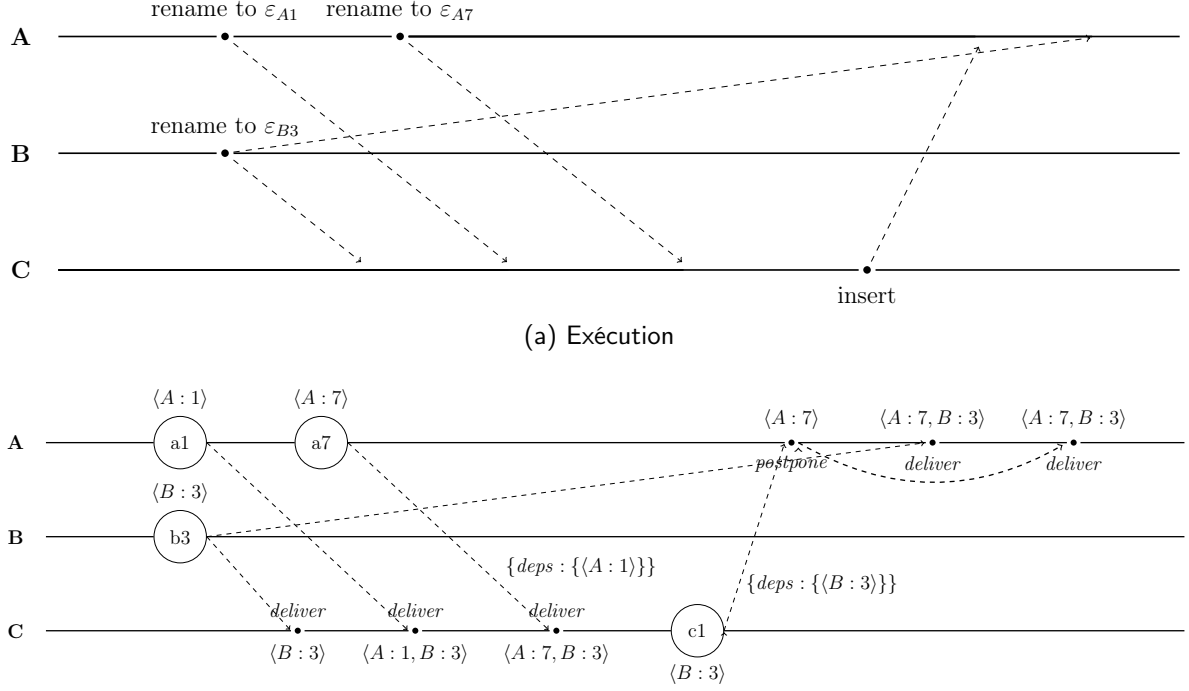


FIGURE 3.6 – Gestion de la livraison *epoch based* des opérations

Dans la Figure 3.6a, nous décrivons une exécution suivante en ne faisant apparaître que les opérations importantes : les opérations *rename* et une opération *insert* finale. Dans cette exécution, trois noeuds A, B et C répliquent et éditent collaborativement une séquence. Initialement, aucune opération *rename* n'a encore eu lieu. Le noeud A effectue une première opération *rename* (*a1*) puis une seconde opération *rename* (*a7*), et les diffuse. En concurrence, le noeud B génère et propage sa propre opération *rename* (*b3*). De son côté, le noeud C reçoit les opérations *b3*, puis *a1* et *a7*. Il émet ensuite une opération *insert* (*c1*). Le noeud A reçoit cette opération avant de finalement recevoir l'opération *b3*.

Dans la Figure 3.6b, nous faisons apparaître l'état de **Sync** et les décisions prises par ce dernier au cours de l'exécution. Initialement, le vecteur des dots des opérations *rename* connues est vide. Ainsi, lorsque A génère l'opération *a1*, celle-ci ne se voit ajouter aucune dépendance (nous ne représentons pas les dépendances des opérations qui correspondent à l'ensemble vide). A met ensuite à jour son vecteur des dots des opérations *rename* avec le dot $\langle A : 1 \rangle$. B procède de manière similaire avec l'opération *b3*.

Quand A génère l'opération *a7*, le dot $\langle A : 1 \rangle$ est ajouté en tant que dépendance. Le dot $\langle A : 7 \rangle$ remplace ensuite ce dernier dans le vecteur des dots des opérations *rename*.

À la réception de l'opération *b3*, le module **Sync** de C peut la délivrer au CRDT,

l'ensemble de ses dépendances étant vérifié. Le noeud C ajoute alors à son vecteur des dots des opérations *rename* le dot $\langle B : 3 \rangle$. Il procède de même pour l'opération *a1* : il la délivre et ajoute le dot $\langle A : 1 \rangle$. Le module **Sync** ne connaissant pas l'époque courante de la séquence répliquée, il maintient les deux dots localement.

Lorsque le noeud C reçoit l'opération *a7*, l'ensemble de ses contraintes est vérifié : l'opération *a1* a été délivrée précédemment. L'opération est donc délivrée et le vecteur de dots des opérations *rename* mis à jour avec $\langle A : 7 \rangle$.

Quand le noeud C effectue l'opération locale *c1*, le module **Sync** obtient l'information de l'époque courante de la séquence : ε_{b3} . C met à jour son vecteur de dots des opérations *rename* pour ne conserver que l'entrée du noeud B : $\langle B : 3 \rangle$. Ce dot est ajouté en tant que dépendance de l'opération *c1* avant sa diffusion.

À la réception de l'opération *c1* par le noeud A, cette opération est mise en attente par **Sync**, l'opération *b3* n'ayant pas encore été délivrée. Le noeud reçoit ensuite l'opération *b3*. Son vecteur des dots des opérations *rename* est mis à jour et l'opération délivrée. Les conditions pour l'opération *c1* étant désormais remplies, l'opération est alors délivrée.

Cette implémentation de la contrainte de la livraison *epoch-based* dispose de plusieurs avantages : sa complexité spatiale dépend linéairement du nombre de noeuds et les opérations de mise à jour du vecteur des dots des opérations *rename* s'effectuent en temps constant. De plus, seul un dot est ajouté en tant que dépendance des opérations, la taille du vecteur des dots étant ramené à 1 au préalable. Finalement, cette implémentation ne contraint pas une livraison causale des opérations *rename* et permet donc de les appliquer dès que possible.

Livraison des opérations à terme

La dernière contrainte du modèle de livraison précise que toutes les opérations doivent être délivrées à tous les noeuds à terme. Cependant, le réseau étant non-fiable, des messages peuvent être perdus au cours de l'exécution. Il est donc nécessaire que les noeuds rediffusent les messages perdus pour assurer leur livraison à terme.

Pour cela, nous implémentons un mécanisme d'anti-entropie basé sur [54]. Ce mécanisme permet à un noeud source de se synchroniser avec un autre noeud cible. Il est exécuté par l'ensemble des noeuds de manière indépendante. Nous décrivons ci-dessous son fonctionnement.

De manière périodique, le noeud choisit un autre noeud cible de manière aléatoire. Le noeud source lui envoie alors une représentation de son état courant, c.-à-d. son vecteur de version.

À la réception de ce message, le noeud cible compare le vecteur de version reçu par rapport à son propre vecteur de version. À partir de ces données, il identifie les dots des opérations de sa connaissance qui sont inconnues au noeud source. Grâce à leur dot, le noeud cible retrouve ces opérations depuis son log des opérations. Il envoie alors une réponse composée de ces opérations au noeud source.

À la réception de la réponse, le noeud source intègre normalement les opérations reçues. La Figure 3.7 illustre ce mécanisme.

Dans cette figure, nous représentons une exécution à laquelle participent trois noeuds : A, B et C. Initialement, les trois noeuds sont synchronisés. Leur vecteurs de version sont

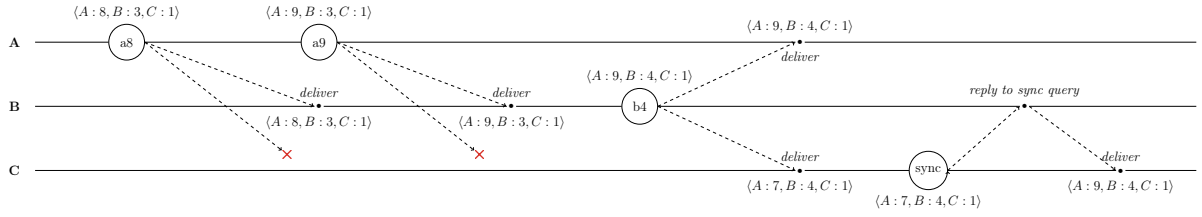


FIGURE 3.7 – Utilisation du mécanisme d’anti-entropie par le noeud C pour se synchroniser avec le noeud B

identiques et ont pour valeur $\langle A : 7, B : 3, C : 1 \rangle$.

Le noeud A effectue les opérations $a8$ puis $a9$ et les diffusent sur le réseau. Le noeud B reçoit ces opérations et les délivre à son CRDT. Il effectue ensuite et propage l’opération $b4$, qui est reçue et délivrée par A. Ils atteignent tous deux la version représenté par le vecteur $\langle A : 9, B : 4, C : 1 \rangle$

De son côté, le noeud C ne reçoit pas les opérations $a8$ et $a9$ à cause d’une défaillance réseau. Néanmoins, cela ne l’empêche pas de délivrer l’opération $b4$ à sa réception et d’obtenir la version $\langle A : 7, B : 4, C : 1 \rangle$.

Le noeud C déclenche ensuite son mécanisme d’anti-entropie. Il choisit aléatoirement le noeud B comme noeud cible. Il lui envoie un message de synchronisation avec pour contenu le vecteur de version $\langle A : 7, B : 8, C : 1 \rangle$.

À la réception de ce message, le noeud B compare ce vecteur avec le sien. Il détermine que le noeud C n’a pas reçu les opérations $a8$ et $a9$. B les récupère depuis son log des opérations et les envoie à C par le biais d’un nouveau message.

À la réception de la réponse de B, le noeud C délivre les opérations $a8$ et $a9$. Il atteint alors le même état que A et B, représenté par le vecteur de version $\langle A : 9, B : 4, C : 1 \rangle$.

Ce mécanisme d’anti-entropie nous permet ainsi de garantir la livraison à terme de toutes les opérations et de compenser les défaillances du réseau. Il nous sert aussi de mécanisme de synchronisation : à la connexion d’un pair, celui-ci utilise ce mécanisme pour récupérer les opérations effectuées depuis sa dernière connexion. Dans le cas où il s’agit de la première connexion du pair, il lui suffit d’envoyer un vecteur de version vide pour récupérer l’intégralité des opérations.

Ce mécanisme propose plusieurs avantages. Son exécution n’implique que le noeud source et le noeud cible, ce qui limite les coûts de coordination. De plus, si une défaillance a lieu lors de l’exécution du mécanisme (perte d’un des messages, panne du noeud cible...), cette défaillance n’est pas critique : le noeud source se synchronisera à la prochaine exécution du mécanisme. Ensuite, ce mécanisme réutilise le vecteur de version déjà nécessaire pour la livraison *exactly-once*, comme présenté en section 3.2.2. Il ne nécessite donc pas de stocker une nouvelle structure de données pour détecter les différences entre noeuds.

En contrepartie, la principale limite de ce mécanisme d’anti-entropie est qu’il nécessite de maintenir et de parcourir périodiquement le log des opérations pour répondre aux requêtes de synchronisation. La complexité spatiale et en temps du mécanisme dépend donc linéairement du nombre d’opérations. Qui plus est, nous sommes dans l’incapacité de tronquer le log des opérations en se basant sur la stabilité causale des opérations puisque nous utilisons ce mécanisme pour mettre à niveau les nouveaux pairs. À moins de mettre

en place un mécanisme de compression du log comme évoqué en sous-section 2.5.6, ce log des opérations croît de manière monotone. Néanmoins, une alternative possible est de mettre en place un système de chargement différé des opérations pour ne pas surcharger la mémoire.

3.2.3 Collaborateurs

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

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

En raison de ces contraintes, notre choix s’est porté sur le protocole SWIM[18]. Proposé par DAS, GUPTA et MOTIVALA, ce protocole d’appartenance au réseau offre les propriétés intéressantes suivantes. Tout d’abord, le nombre de messages diffusés sur le réseau est proportionnel de façon linéaire au nombre de pairs. Pour être plus précis, le nombre de messages envoyés par un pair par période du protocole est constant. De plus, il fournit à chaque noeud une vue de la liste des collaborateurs cohérente à terme, même en cas de réception désordonnée des messages du protocoles. Finalement, il intègre un mécanisme permettant de réduire le taux de faux positifs, c.-à-d. le taux de pairs déclarés injustement comme défaillants.

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

Mécanisme de détection des défaillances des pairs

Le mécanisme de détection des défaillances des pairs est exécuté de manière périodique, toutes les T unités de temps, par chacun des noeuds du système de manière non-coordonnée. Son fonctionnement est illustré par la Figure 3.8.

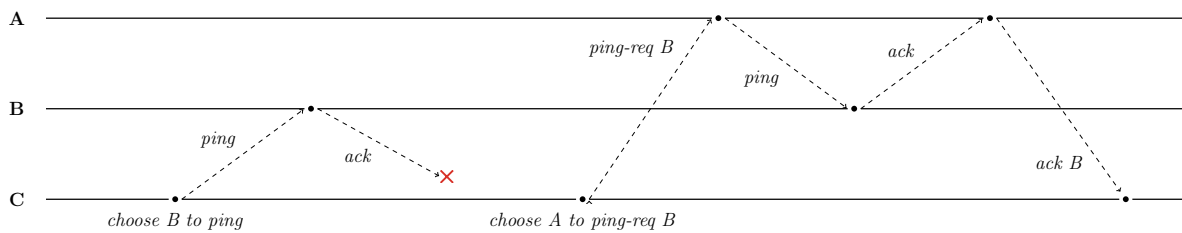


FIGURE 3.8 – Exécution du mécanisme de détection des défaillances par le noeud C pour tester le noeud B

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

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

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

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

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

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

Ainsi, afin de représenter la liste des collaborateurs, le protocole SWIM utilise la structure de données présentée par la Définition 21 :

Définition 21 (Liste des collaborateurs) *La liste des collaborateurs est un ensemble de triplets $\langle nodeId, nodeStatus, nodeIncarn \rangle$ où*

- *$nodeId$ correspond à l'identifiant du noeud correspondant à ce tuple.*
- *$nodeStatus$ correspond au statut courant du noeud correspondant à ce tuple, c.-à-d. *Alive* s'il est considéré comme opérationnel, *Suspect* s'il est suspecté d'une défaillance, *Confirm* s'il est considéré comme défaillant.*
- *$nodeIncarn$ correspond au numéro d'incarnation maximal, c.-à-d. le plus récent, connu pour le noeud correspondant à ce tuple.*

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

Mécanisme de dissémination des mises à jour du groupe

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

Or, diffuser cette mise à jour à l'ensemble du réseau serait coûteux pour un seul noeud. Afin de propager cette information de manière efficace, SWIM propose d'utiliser un protocole de diffusion épidémique : le noeud transmet la mise à jour qu'à un nombre réduit λ^6 de pairs, qui se chargeront de la transmettre à leur tour. Le mécanisme de dissémination des mises à jour de SWIM fonctionne donc de la manière suivante.

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

À la réception d'un message, le noeud le traite comme définit précédemment en section 3.2.3. De manière additionnelle, il intègre dans sa liste des collaborateurs les mises à jour attachées au message en utilisant la règle suivante :

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

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

Modifications apportées

Nous avons ensuite apporté plusieurs modifications à la version du protocole SWIM présentée dans [18]. Notre première modification porte sur l'ordre de priorité entre les états d'un pair.

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

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

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

6. [18] montre que choisir une valeur constante faible comme λ suffit néanmoins à garantir la dissémination des mises à jour à l'ensemble du réseau.

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

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

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

et

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

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

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

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

Nous avons donc implémenté pour la liste des collaborateurs un mécanisme similaire à celui présenté en section 3.2.2 : à sa connexion, puis de manière périodique, un noeud envoie une requête de synchronisation à un noeud cible choisi de manière aléatoire. Ce message sert aussi à transmettre l'état courant du noeud source au noeud cible. En réponse, le noeud cible lui envoie l'état courant de sa liste. À la réception de cette dernière, le noeud source fusionne la liste reçue avec sa propre liste. Cette fusion conserve l'entrée la plus récente pour chaque noeud.

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

Synthèse

Pour générer et maintenir la liste des collaborateurs, nous avons implémenté le protocole distribué d'appartenance au réseau SWIM[18]. Par rapport à la version originale, nous avons procédé à plusieurs modifications, notamment pour gérer plus efficacement les reconnections successives d'un même noeud.

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

Des améliorations au protocole SWIM furent proposées dans [16]. Ces modifications visent notamment à réduire le délai de détection d'un noeud défaillant, ainsi que réduire le taux de faux positifs. Ainsi, une perspective est d'implémenter ces améliorations dans MUTE.

3.2.4 Curseurs

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

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

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

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

3.3 Couche réseau

Pour permettre aux différents noeuds de communiquer, MUTE repose sur la librairie Netflux⁷. Développée au sein de l'équipe Coast, cette librairie permet de construire un réseau P2P entre des navigateurs, mais aussi des bots.

3.3.1 Établissement d'un réseau P2P entre navigateurs

Pour créer un réseau P2P entre navigateurs, Netflux utilise la technologie Web Real-Time Communication (WebRTC). WebRTC est une API⁸ de navigateur spécifiée en 2011, et en cours d'implémentation dans les différents navigateurs depuis 2013. Elle permet de créer une connexion directe entre deux navigateurs pour échanger des médias audio et/ou vidéo, ou simplement des données.

7. <https://github.com/coast-team/netflux>

8. Application Programming Interface (API) : Interface de Programmation

Cette API utilise pour cela un ensemble de protocoles. Ces protocoles réintroduisent des serveurs dans l'architecture système de MUTE. Dans la Figure 3.9, nous représentons un réseau P2P créé avec WebRTC et les différents serveurs impliqués.

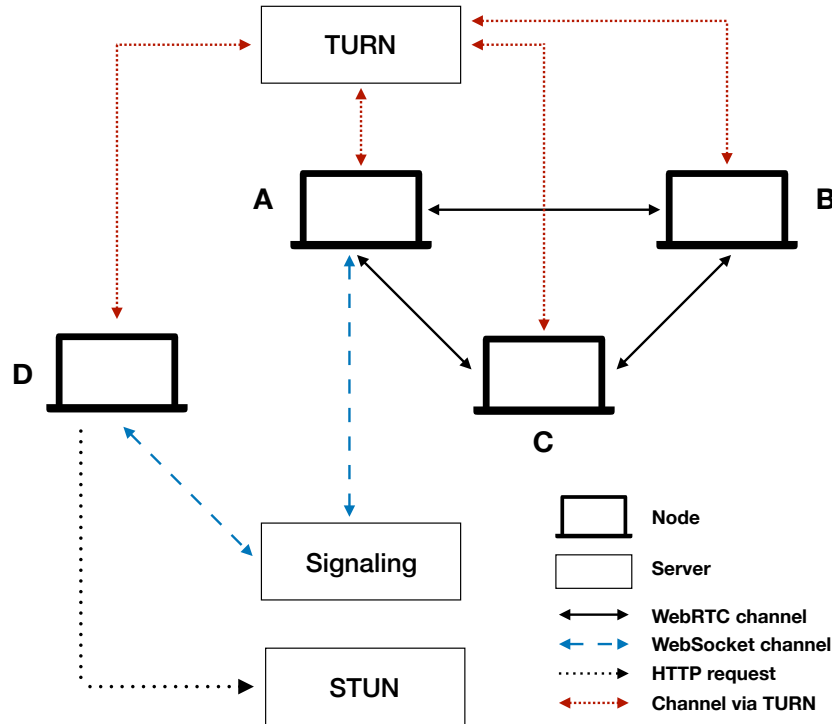


FIGURE 3.9 – Architecture système pour la couche réseau de MUTE

Nous décrivons ci-dessous leur rôle respectif dans la collaboration.

Serveur de signalisation

Pour rejoindre un réseau P2P déjà établi, un nouveau nœud a besoin de découvrir les nœuds déjà connectés et de pouvoir communiquer avec eux. Le serveur de signalisation offre ces fonctionnalités.

Au moins un nœud du réseau P2P doit maintenir une connexion avec le serveur de signalisation. À sa connexion, un nouveau nœud contacte le serveur de signalisation. Il est mis en relation avec un nœud du réseau P2P par son intermédiaire et échange les différents messages de WebRTC nécessaires à l'établissement d'une connexion P2P entre eux.

Une fois cette première connexion P2P établie, le nouveau nœud contacte et communique avec les autres nœuds par l'intermédiaire du premier nœud. Il peut alors terminer sa connexion avec le serveur de signalisation.

Serveur STUN

Pour se connecter, les nœuds doivent s'échanger plusieurs informations logicielles et matérielles, notamment leur adresse IP publique respective. Cependant, un nœud n'a pas

accès à cette donnée lorsque son routeur utilise le protocole NAT. Le noeud doit alors la récupérer.

Pour permettre aux noeuds de découvrir leur adresse IP publique, WebRTC repose sur le protocole STUN. Ce protocole consiste simplement à contacter un serveur tiers dédié à cet effet. Ce serveur retourne en réponse au noeud qui le contacte son adresse IP publique.

Serveur TURN

Il est possible que des noeuds provenant de réseaux différents ne puissent établir une connection P2P directe entre eux, par exemple à cause de restrictions imposées par leur pare-feux respectifs. Pour contourner ce cas de figure, WebRTC utilise le protocole TURN.

Ce protocole consiste à utiliser un serveur tiers comme relais entre les noeuds. Ainsi, les noeuds peuvent communiquer par son intermédiaire tout au long de la collaboration. Les échanges sont chiffrés, afin que le serveur TURN ne représente pas une faille de sécurité.

Rôle des serveurs

Ainsi, WebRTC implique l'utilisation de plusieurs serveurs.

Les serveurs de signalisation et STUN sont nécessaires pour permettre à de nouveaux noeuds de rejoindre la collaboration. Autrement dit, leur rôle est ponctuel : une fois le réseau P2P établi, les noeuds n'ont plus besoin d'eux. Ces serveurs peuvent alors être coupés sans impacter la collaboration.

À l'inverse, les serveurs TURN jouent un rôle plus prédominant dans la collaboration. Ils sont nécessaires dès lors que des noeuds proviennent de réseaux différents et sont alors requis tout au long de la collaboration. Une panne de ces derniers entraverait la collaboration puisqu'elle résulterait en une partition des noeuds. Il est donc primordial de s'assurer de la disponibilité et fiabilité de ces serveurs.

3.3.2 Topologie réseau

Netflux établit un réseau P2P par document. Chaque réseau P2P est un réseau entièrement maillé : chaque noeud se connecte à l'ensemble des autres noeuds.

Cette topologie simple est adaptée à des groupes de petite taille, mais ne passe pas à l'échelle. D'autres topologies limitant le nombre de connexions par noeuds, telle que celle décrite par [43], pourraient être implémentées pour adresser cette limite.

3.4 Couche sécurité

La couche sécurité a pour but de garantir l'authenticité et la confidentialité des messages échangés par les noeuds. Pour cela, elle implémente un mécanisme de chiffrement de bout en bout.

Pour chiffrer les messages, MUTE utilise un mécanisme de chiffrement à base de clé de groupe. Le protocole choisi est le protocole Burmester-Desmedt[13]. Il nécessite que

chaque noeud possède une paire de clés de chiffrement et enregistre sa clé publique auprès d'un PKI⁹.

Afin d'éviter qu'un PKI malicieux n'effectue une attaque de l'homme au milieu sur la collaboration, les noeuds doivent vérifier le bon comportement des PKI de manière non-coordonnée. À cet effet, MUTE implémente le mécanisme d'audit de PKI Trusternity[45, 46]. Son fonctionnement nécessite l'utilisation d'un registre public sécurisé *append-only*, c.-à-d. une blockchain.

L'architecture système nécessaire pour la couche sécurité est présentée dans la Figure 3.10.

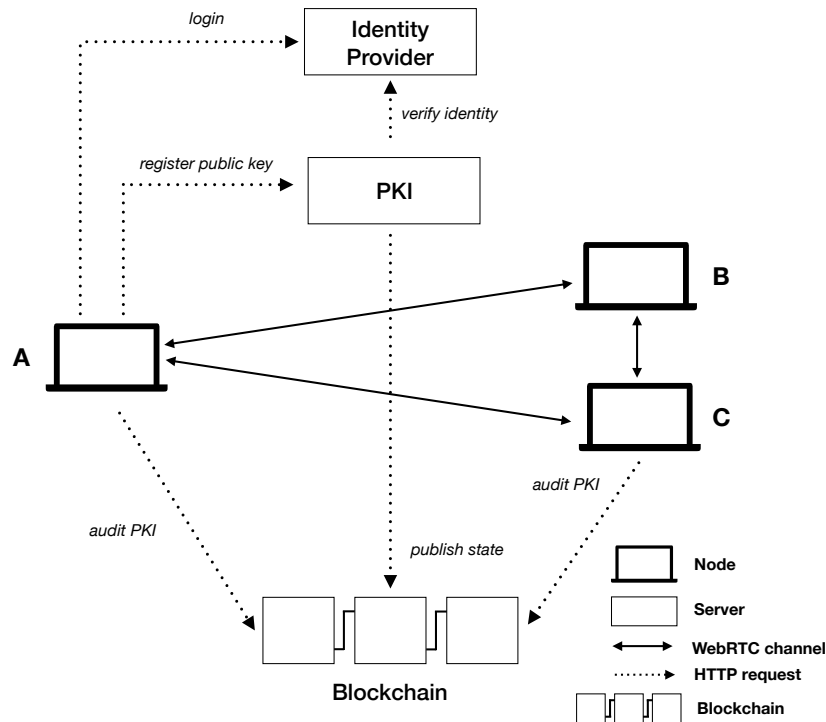


FIGURE 3.10 – Architecture système pour la couche sécurité de MUTE

Cette couche sécurité s'ajoute au mécanisme de chiffrement des messages inhérent à WebRTC. Cela nous offre de nouvelles possibilités : plutôt que de créer un réseau P2P par document, nous pouvons désormais mettre en place un réseau P2P global. Les messages étant chiffrés de bout en bout, les noeuds peuvent communiquer en toute sécurité et confidentialité par l'intermédiaire de noeuds tiers, c.-à-d. des noeuds extérieurs à la collaboration.

Une limite de l'approche actuelle est que la clé de groupe change avec l'évolution des noeuds connectés : à chaque connexion ou déconnexion d'un noeud, une nouvelle clé est recalculée avec les collaborateurs présents. Cette évolution fréquente de la clé de chiffrement, nécessaire pour garantir la *backward secrecy* et *forward secrecy*, nous empêche par exemple de stocker les opérations de manière chiffrée chez des noeuds tiers. Cette fonctionnalité serait cependant bien pratique pour permettre à un noeud de récupérer la

9. Public Key Infrastructure (PKI) : Infrastructure de gestion de clés

dernière version de ses documents, même en l’absence de ses collaborateurs. Une autre clé de chiffrement, dédiée au stockage, devrait être mise en place, ainsi qu’un mécanisme de découverte des noeuds tiers stockant les données de la collaboration.

3.5 Conclusion

Dans ce chapitre, nous avons présenté Multi User Text Editor (MUTE), notre éditeur collaboratif temps réel P2P chiffré de bout en bout.

MUTE permet d’éditer de manière collaborative des documents texte. Pour représenter les documents, MUTE implémente les structures de données répliquées décrites dans la section 1.4 et le chapitre 2. Ces CRDTs offrent de nouvelles méthodes de collaborer, notamment en permettant de collaborer de manière synchrone ou asynchrone de manière transparente.

Pour permettre aux noeuds de communiquer, MUTE utilise WebRTC. Cette technologie permet de construire un réseau P2P entre navigateurs. Plusieurs serveurs sont néanmoins requis, notamment pour la découverte des pairs et pour la communication entre des noeuds dont les pare-feux respectifs empêche l’établissement d’une connexion directe.

Finalement, MUTE implémente un mécanisme de chiffrement de bout en bout garantissant l’authenticité et la confidentialité des échanges entre les noeuds. Ce mécanisme reposant sur d’autres serveurs, les PKIs, MUTE intègre un mécanisme d’audit permettant de détecter leurs éventuels comportements malicieux.

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] Paulo Sérgio ALMEIDA et al. « Scalable and Accurate Causality Tracking for Eventually Consistent Stores ». In : *Distributed Applications and Interoperable Systems*. Sous la dir. de Kostas MAGOUTIS et Peter PIETZUCH. Berlin, Heidelberg : Springer Berlin Heidelberg, 2014, p. 67–81. ISBN : 978-3-662-43352-2.
- [6] 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.
- [7] AUTOMERGE. *Automerge : data structures for building collaborative applications in Javascript*. URL : <https://github.com/automerge/automerge>.
- [8] Carlos BAQUERO, Paulo Sergio ALMEIDA et Ali SHOKER. *Pure Operation-Based Replicated Data Types*. 2017. arXiv : 1710.04469 [cs.DC].
- [9] 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>.

- [10] 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.
- [11] 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>.
- [12] 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>.
- [13] 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.
- [14] CONCORDANT. *Concordant*. URL : <http://www.concordant.io/>.
- [15] The SyncFree CONSORTIUM. *AntidoteDB : A planet scale, highly available, transactional database*. URL : <http://antidoteDB.eu/>.
- [16] 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.
- [17] Quang-Vinh DANG et Claudia-Lavinia IGNAT. « Performance of real-time collaborative editors at large scale : User perspective ». In : *Internet of People Workshop, 2016 IFIP Networking Conference*. Proceedings of 2016 IFIP Networking Conference, Networking 2016 and Workshops. Vienna, Austria, mai 2016, p. 548–553. DOI : 10.1109/IFIPNetworking.2016.7497258. URL : <https://hal.inria.fr/hal-01351229>.
- [18] 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.
- [19] 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.
- [20] Giuseppe DECANDIA et al. « Dynamo : Amazon's highly available key-value store ». In : *ACM SIGOPS operating systems review* 41.6 (2007), p. 205–220.

-
- [21] 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>.
- [22] Victorien ELVINGER, Gérard 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.
- [23] ETHERPAD. *Etherpad*. URL : <https://etherpad.org/>.
- [24] Barton GELLMAN et Laura POITRAS. *U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program*. URL : https://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html.
- [25] Jim GILES. « Special Report Internet encyclopaedias go head to head ». In : *nature* 438.15 (2005), p. 900–901.
- [26] Ricardo Jorge Tomé GONÇALVES et al. « DottedDB : Anti-Entropy without Merkle Trees, Deletes without Tombstones ». In : *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. 2017, p. 194–203. DOI : 10.1109/SRDS.2017.28.
- [27] GOOGLE. *Google Docs*. URL : <https://docs.google.com/>.
- [28] Glen GREENWALD et Ewen MACASKILL. *NSA Prism program taps in to user data of Apple, Google and others*. URL : <https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data>.
- [29] 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>.
- [30] 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.
- [31] Claudia-Lavinia IGNAT. « Maintaining consistency in collaboration over hierarchical documents ». Thèse de doct. ETH Zurich, 2006.
- [32] 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>.

- [33] 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>.
- [34] 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>.
- [35] 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>.
- [36] 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>.
- [37] Nico KRUBER, Maik LANGE et Florian SCHINTKE. « Approximate Hash-Based Set Reconciliation for Distributed Replica Repair ». In : *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. 2015, p. 166–175. DOI : 10.1109/SRDS.2015.30.
- [38] 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>.
- [39] 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.
- [40] 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.
- [41] 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.

-
- [42] 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>.
 - [43] Brice NÉDELEC et al. « An adaptive peer-sampling protocol for building networks of browsers ». In : *World Wide Web* 21.3 (2018), p. 629–661.
 - [44] 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.
 - [45] 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>.
 - [46] 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>.
 - [47] 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.
 - [48] 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>.
 - [49] 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>.
 - [50] Matthieu NICOLAS, G rald 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>.
 - [51] 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>.

- [52] Sylvie NOËL et Jean-Marc ROBERT. « Empirical study on collaborative writing : What do co-authors do, use, and like ? » In : *Computer Supported Cooperative Work (CSCW)* 13.1 (2004), p. 63–89.
- [53] 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>.
- [54] 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>.
- [55] Jim PICK. *Graf, Nikolaus*. URL : <https://www.serenity.re/en/notes>.
- [56] Jim PICK. *PeerPad*. URL : <https://peerpad.net/>.
- [57] 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.
- [58] Pierre-Antoine RAULT, Claudia-Lavinia IGNAT et Olivier PERRIN. « Distributed Access Control for Collaborative Applications Using CRDTs ». In : *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '22. Rennes, France : Association for Computing Machinery, 2022, p. 33–38. ISBN : 9781450392563. DOI : 10.1145/3517209.3524826. URL : <https://doi.org/10.1145/3517209.3524826>.
- [59] RIAK. *Riak KV*. URL : <http://riak.com/>.
- [60] 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>.
- [61] 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>.
- [62] 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>.
- [63] 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.
- [64] 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.

-
- [65] 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>.
 - [66] 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>.
 - [67] 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>.
 - [68] 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>.
 - [69] 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.
 - [70] 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.
 - [71] 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>.
 - [72] YJS. *Yjs : A CRDT framework with a powerful abstraction of shared data*. URL : <https://github.com/yjs/yjs>.
 - [73] 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>.
 - [74] 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>.
 - [75] 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>.