Chapitre 4

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

Sommaire

Les systèmes collaboratifs temps réels permettent à plusieurs utilisateur-rices de réaliser une tâche de manière coopérative. Ils permettent aux utilisateur-rices de consulter le contenu actuel, de le modifier et d'observer en direct les modifications effectuées par

les autres collaborateur-rices. 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 [95, 7].

Plusieurs outils d'édition collaborative centralisés basés sur l'approche Operational Transformation (OT) [62] ont permis de populariser l'édition collaborative temps réel de texte [25, 96]. Ces approches souffrent néanmoins de leur architecture centralisée. Notamment, ces solutions rencontrent des difficultés à passer à l'échelle [90, 97] et posent des problèmes de confidentialité [98, 99].

L'approche CRDT offre une meilleure capacité de passage à l'échelle et est compatible avec une architecture P2P [77]. Ainsi, de nombreux travaux [100, 101, 102] 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 (LFS) [14, 103]. 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) [27] [20] [21], 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.

Ainsi, nous avons contribué à son développement dans le cadre de cette thèse. Notamment, nous avons participé à :

(i) L'implémentation des CRDTs LogootSplit [28] et RenamableLogootSplit [30] pour représenter le document texte.

(ii) L'implémentation de leur modèle de livraison de livraison respectifs.

(iii) L'implémentation d'un protocole d'appartenance au réseau, SWIM [33].

Dans ce chapitre, nous commençons par présenter le projet MUTE : ses objectifs, ses fonctionnalités et son architecture système et logicielle. Puis nous détaillons ses différentes couches logicielles : leur rôle, l'approche choisie pour leur implémentation et finalement leurs limites actuelles. Au cours de cette description, nous mettons l'emphase sur les composants auxquelles nous avons contribué, c.-à-d. les sections 4.3, et 4.4.

4.1 Présentation

4.1.1 Objectifs

Comme indiqué dans l'introduction (cf. section 1.1, page 1), le but de ce projet est de proposer un éditeur de texte collaboratif Local-First Software (LFS), c.-à-d. un éditeur de texte collaboratif qui satisfait les propriétés suivantes :

20. Disponible à l'adresse : `https://mutehost.loria.fr`
21. Code source disponible à l'adresse suivante : `https://github.com/coast-team/mute`

(i) Toujours disponible, c.-à-d. qui permet à tout moment à un-e utilisateur-rice de consulter, créer ou éditer un document, même par exemple en l'absence de connexion internet.

(ii) Collaboratif, c.-à-d. qui permet à un-e utilisateur-rice de partager un document avec d'autres utilisateur-rices pour éditer à plusieurs le document, de manière synchrone et asynchrone. Nous considérons la capacité d'un-e utilisateur-rice à partager le document avec ses propres autres appareils comme un cas particulier de collaboration.

(iii) Performant, c.-à-d. qui garantit que le délai entre la génération d'une modification par un pair et l'intégration de cette dernière par un autre pair connecté soit assimilable à du temps réel et que ce délai ne soit pas impacté par le nombre de pairs dans la collaboration.

(iv) Pérenne, c.-à-d. qui garantit à ses utilisateur-rices qu'iels pourront continuer à utiliser l'application sur une longue période. Notamment, nous considérons la capacité des utilisateur-rices à configurer et déployer aisément leur propre instance du système comme un gage de pérennité du système.

(v) Garantissant la confidentialité des données, c.-à-d. qui permet à un-e utilisateur-rice de contrôler avec quelles personnes une version d'un document est partagée. Aussi, le système doit garantir qu'un adversaire ne doit pas être en mesure d'espionner les utilisateur-rices, e.g. en usurpant l'identité d'un-e utilisateur-rice ou en interceptant les messages diffusés sur le réseau.

(vi) Garantissant la souveraineté des données, c.-à-d. qui permet à un-e utilisateur-rice de maîtriser l'usage de ses données, e.g. pouvoir les consulter, modifier, partager ou encore exporter vers d'autres formats ou applications.

Ainsi, ces différentes propriétés nous conduisent à concevoir un éditeur de texte collaboratif P2P temps réel chiffré de bout en bout et qui est dépourvu d'autorités centrales.

4.1.2 Fonctionnalités

MUTE prend la forme d'une application web qui permet de créer et de gérer des documents textes. Chaque document se voit attribuer un identifiant, supposé unique. L'utilisateur-rice peut alors ouvrir et partager un document à partir de son URL.

L'application permet à l'utilisateur-rice d'être mis-e en relation avec les autres pairs actuellement connectés qui travaillent sur ce même document. Pour cela, l'application utilise le protocole WebRTC afin d'établir des connexions P2P avec ces derniers. Une fois les connexions P2P établies, le service fourni par le système pour mettre en relation les pairs n'est plus nécessaire.

Une fois connecté à un autre pair, l'utilisateur-rice récupère automatiquement les modifications effectuées par ses pairs de façon à obtenir la version courante du document. Iel peut alors modifier le document, c.-à-d. ajouter, supprimer du contenu ou encore modifier son titre. Ses modifications sont partagées en temps réel aux autres pairs connectés. À la réception de modifications, celles-ci sont intégrées à la copie locale du document. Figure 4.1 illustre l'interface utilisateur de l'éditeur de document de MUTE.

Pour garantir la confidentialité des échanges, MUTE utilise un protocole de génération de clés de groupe. Ce protocole permet d'établir une clé de chiffrement connue seulement

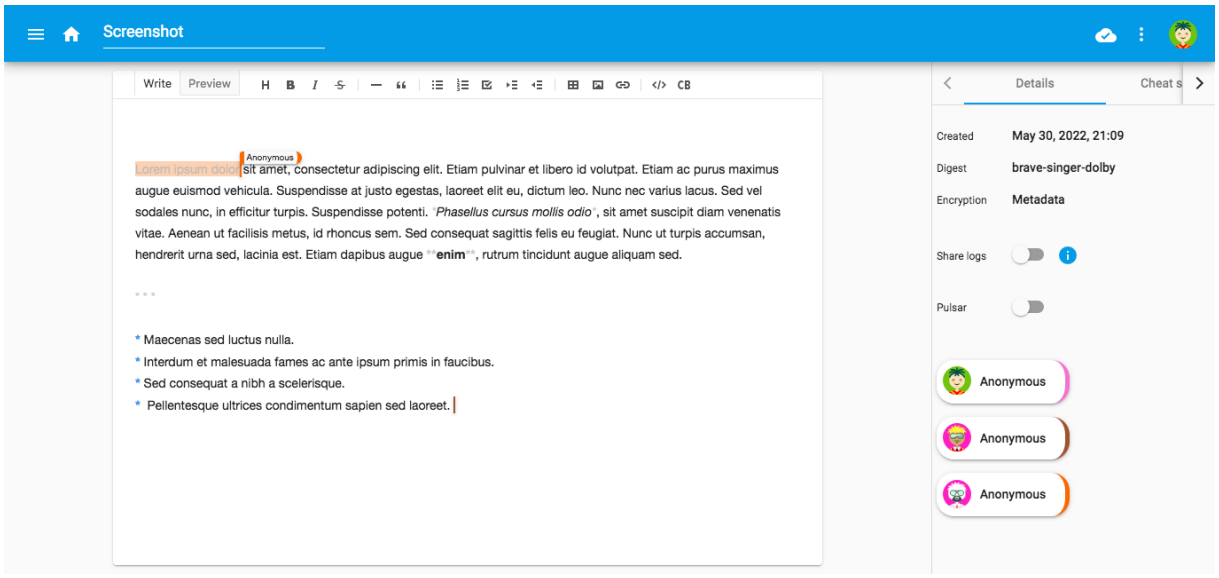


FIGURE 4.1 – Capture d'écran d'une session d'édition collaborative avec MUTE

des pairs actuellement connectés, qui est ensuite utilisée pour chiffrer les messages entre pairs. Ce protocole permet de garantir les propriétés de *backward secrecy* et de *forward secrecy*.

Définition 56 (Backward Secrecy)**.** La *Backward Secrecy* est une propriété de sécurité garantissant qu'un nouveau noeud ne pourra pas déchiffrer avec la nouvelle clé de chiffrement les anciens messages chiffrés avec une clé de chiffrement précédente.

Définition 57 (Forward Secrecy)**.** La *Forward Secrecy* est une propriété de sécurité garantissant qu'un nouveau noeud ne pourra pas déchiffrer avec la nouvelle clé de chiffrement les futurs messages chiffrés avec une prochaine clé de chiffrement.

Une copie locale du document est sauvegardée dans le navigateur, avec l'ensemble des modifications. L'utilisateur-rice peut ainsi accéder à ses documents même sans connexion internet, pour les consulter ou modifier. Les modifications effectuées dans ce mode hors-ligne seront partagées aux collaborateur-rices à la prochaine connexion de l'utilisateur-rice.

Finalement, la page d'accueil de l'application permet aussi de lister ses documents. L'utilisateur-rice peut ainsi facilement parcourir ses documents, récupérer leur url pour les partager ou encore supprimer leur copie locale. Figure 4.2 illustre cette page de l'application.

4.1.3 Architecture système

Nous représentons l'architecture système d'une collaboration utilisant MUTE par la Figure 4.3.

Plusieurs types de noeuds composent cette architecture. Nous décrivons ci-dessous le type de chacun de ces noeuds ainsi que leurs rôles.

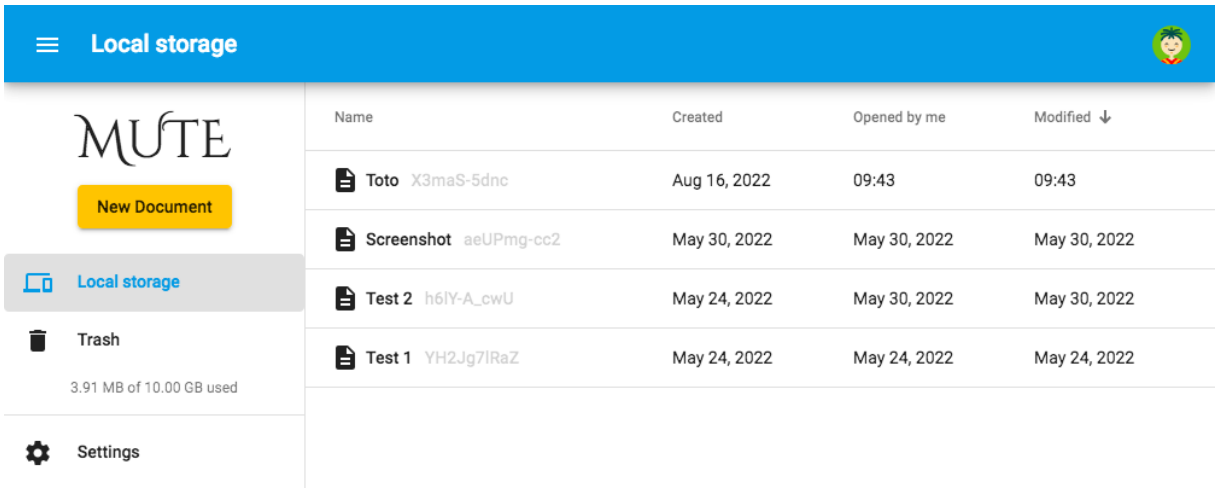


FIGURE 4.2 – Capture d'écran de la liste des documents.

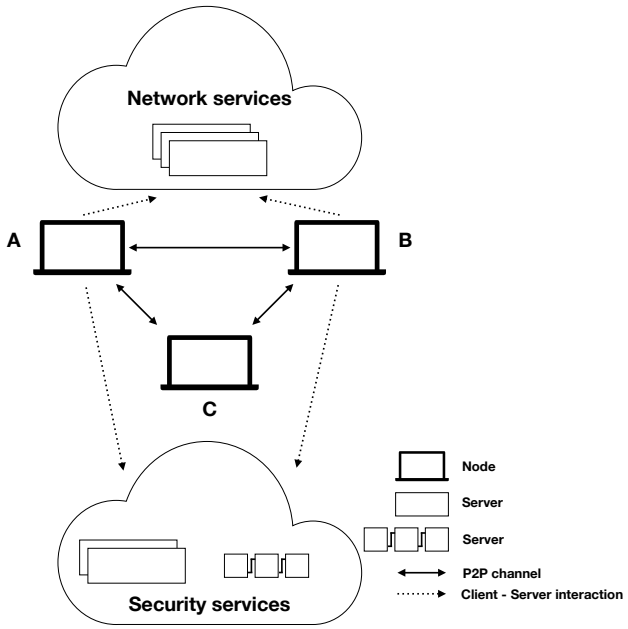


FIGURE 4.3 – Architecture système de l'application MUTE

Pairs

Au centre de la collaboration se trouvent les noeuds qui correspondent aux utilisateurrices de l'application et à leurs appareils. Chaque noeud correspond à une instance de l'application MUTE, c.-à-d. l'éditeur collaboratif de texte. Chacun de ces noeuds peut donc consulter des documents et les modifier.

Ces noeuds forment un réseau P2P, qui leur permet d'échanger directement notamment pour diffuser les modifications effectuées sur le document. Les pairs interagissent aussi avec les autres types de noeuds, que nous décrivons dans les parties suivantes.

Notons qu'un noeud peut toutefois être déconnecté du système, c.-à-d. dans l'incapa-

cité de se connecter aux autres pairs et d'interagir avec les autres types de noeuds. Cela ne l'empêche toutefois pas l'utilisateur-rice d'utiliser MUTE.

Services réseau

Nous décrivons par cette appelation l'ensemble des composants nécessaires à l'établissement et le bon fonctionnement du réseau P2P entre les appareils des utilisateur-rices.

Il s'agit de serveurs ayant pour buts de :

(i) Permettre à un pair d'obtenir les informations sur son propre état nécessaires pour l'établissement de connexions P2P.

(ii) Permettre à un pair de découvrir les autres pairs travaillant sur le même document et d'établir une connexion avec eux.

(iii) Permettre à des pairs de communiquer même si leur configurations réseaux respectives empêchent l'établissement d'une connection P2P directe.

Nous détaillons plus précisément chacun de ces services et les interactions entre les pairs et ces derniers dans la section 4.5.

Services sécurité

Nous décrivons par cette appelation l'ensemble des composants nécessaires à l'authentification des utilisateur-rices et à l'établissement de clés de groupe de chiffrement.

Il s'agit de serveurs ayant pour buts de :

(i) Permettre à un pair de s'authentifier.

(ii) Permettre à un pair de faire connaître sa clé publique de chiffrement.

(iii) Vérifier l'identité d'un pair.

(iv) Permetttre à un pair de vérifier le comportement honnête du ou des serveurs servant les clés publiques de chiffrement.

Nous dédions la section 4.6 à la description de ces différents services et les interactions des pairs avec ces derniers.

4.1.4 Architecture logicielle

Nous décrivons l'architecture logicielle d'un pair, c.-à-d. d'une instance de l'application MUTE dans un navigateur, dans la Figure 4.4.

Cette architecture logicielle se compose de plusieurs composants, que nous regroupons par couche. Chacune de ces couches possède un rôle, que nous présentons brièvement ci-dessous avant de les décrire de manière plus détaillée dans leur section respective.

(i) La couche *interface utilisateur*, qui regroupe l'ensemble des composants permettant de communiquer des informations aux pairs et avec lesquelles ils peuvent interagir, c.-à-d. le document lui-même, son titre mais aussi la liste des collaborateur-rices actuellement connectés. Cette couche se charge de transmettre les actions de l'utilisateur-rice aux couches inférieures, et inversement de présenter à l'utilisateur-rice les modifications effectuées par ses pairs.

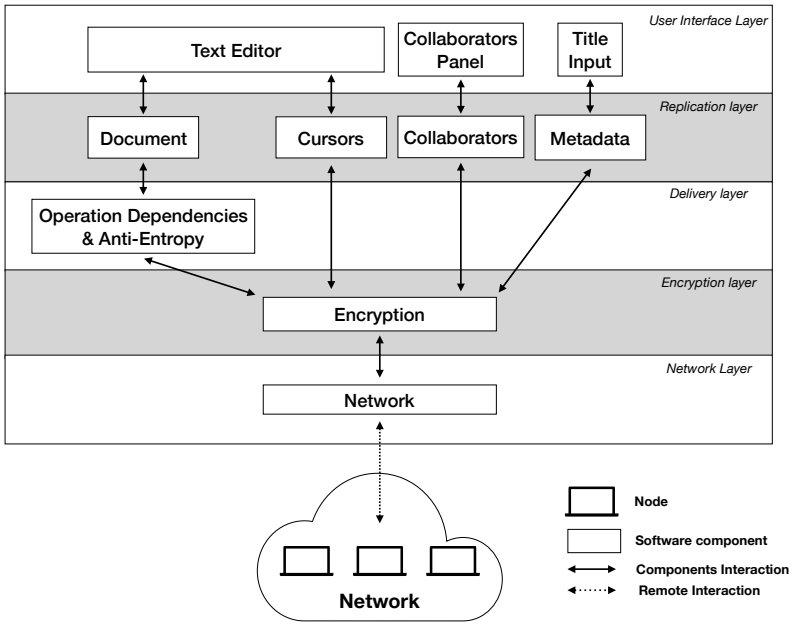


FIGURE 4.4 – Architecture logicielle de l'application MUTE

(ii) La couche *réplication*, qui regroupe l'ensemble des composants permettant de représenter les données répliquées entre pairs, c.-à-d. les CRDTs utilisés pour représenter le document, ses métadonnées (titre, date de création...), l'ensemble des collaborateurs et leur curseur. Cette couche se charge d'intégrer les modifications effectuées par l'utilisateur-rice et de transmettre les opérations correspondantes aux couches inférieures, et inversement d'intégrer les opérations effectuées par ses pairs et d'indiquer à la couche *interface utilisateur* les modifications correspondantes.

(iii) La couche *livraison*, qui est constitué d'un unique composant permettant de garantir les modèles de livraison requis par les différents CRDTs implémentés pour représenter le document. Cette couche se charge d'adjoindre aux opérations de l'utilisateur-rice leur(s) dépendance(s) avant de les transmettre aux couches inférieures, et de livrer les opérations de ses pairs une fois leur(s) dépendance(s) livrées au préalable, ou de les mettre en attente le cas échéant.

(iv) La couche *sécurité*, qui est constitué d'un unique composant gérant le chiffrement des messages. Cette couche se charge d'établir la clé de chiffrement de groupe, puis de chiffrer les messages de l'utilisateur-rice avec cette dernière avant de les transmettre à la couche inférieure, et inversement de déchiffrer les messages chiffrés de ses pairs avant de les transmettre aux couches supérieures.

(v) La couche *réseau*, qui est constitué d'un unique composant permettant d'interagir avec le réseau P2P. Cette couche se charge d'établir les connexions P2P, puis permet de diffuser les messages chiffrés de l'utilisateur-rice à un ou plusieurs de ses pairs, et inversement de transmettre les messages chiffrés de ses pairs à la couche supérieure.

4.2 Couche interface utilisateur

Comme illustré par la Figure 4.1, l'interface de la page d'un document se compose principalement d'un éditeur de texte. Ce dernier supporte le langage de balisage Markdown [104]. Ainsi, l'éditeur permet d'inclure plusieurs éléments légers de style. Les balises du langage Markdown étant du texte, elles sont répliquées nativement par le CRDT utilisé en interne par MUTE pour représenter la séquence.

L'interface de la page de l'éditeur de document est agrémentée de plusieurs mécanismes permettant d'établir une conscience de groupe entre les collaborateur-rices. L'indicateur en haut à droite de la page représente le statut de connexion de l'utilisateur-rice. Celui-ci permet d'indiquer à l'utilisateur-rice s'iel est actuellement connecté-e au réseau P2P, en cours de connexion, ou si un incident réseau a lieu.

De plus, MUTE affiche sur la droite de l'éditeur la liste des collaborateur-rices actuellement connecté-es. Un curseur ou une sélection distante est associée pour chaque membre de la liste. Ces informations permettent d'indiquer à l'utilisateur-rice dans quelles sections du document ses collaborateur-rices sont en train de travailler. Ainsi, iels peuvent se répartir la rédaction du document de manière implicite ou suivre facilement les modifications d'un-e collaborateur-rice.

Bien que fonctionnelle, cette interface souffre néanmoins de plusieurs limites. Notamment, nous n'avons pas encore pu étudier la littérature concernant les mécanismes de conscience pour supporter la collaboration, au-delà du système de curseurs distants.

Nous identifions ainsi plusieurs axes de travail pour ces mécanismes. Tout d'abord, l'axe des *mécanismes de conscience des changements*. Le but serait de proposer des mécanismes pour :

(i) Mettre en lumière de manière intelligible les modifications effectuées par les collabateur-rices dans le cadre de collaborations temps réel à large échelle. Un tel mécanisme représente un défi de part le débit important de changements, potentiellement à plusieurs endroits du document de manière quasi-simultanée, à présenter à l'utilisateur-rice.

(ii) Mettre en lumière de manière intelligible les modifications effectuées par les collaborateur-rices dans le cadre de collaborations asynchrones. De nouveau, ce mécanisme représente un défi de part la quantité massive de changements, une fois encore potentiellement à plusieurs endroits du document, à présenter à l'utilisateur-rice.

Une piste de travail potentiellement liée serait l'ajout d'une fonctionnalité d'historique du document, permettant aux utilisateur-rices de parcourir ses différentes versions obtenues au fur et à mesure des modifications. L'intégration d'une telle fonctionnalité dans un éditeur P2P pose cependant plusieurs questions : quel historique présenter aux utilisateur-rices, sachant que chacun-e a potentiellement observé un ordre différent des modifications ? Doit-on convenir d'une seule version de l'historique ? Dans ce cas, comment choisir et construire cet historique ?

Le second axe de travail sur les mécanismes de conscience concerne les *mécanismes*

de conscience de groupe. Actuellement, nous affichons l'ensemble des collaborateur-rices actuellement connecté-es. Cette approche s'avère lourde voire entravante dans le cadre de collaborations à large échelle où le nombre de collaborateur-rices dépasse plusieurs centaines. Il convient donc de déterminer quelles informations présenter à l'utilisateur-rice dans cette situation, e.g. une liste compacte de pairs et leur curseur respectif, ainsi que le nombre de pairs total.

Ainsi, pour chacun de ses axes d'amélioration, il convient d'étudier leur littérature respective et de déterminer les solutions proposées qui sont adaptées à MUTE.

4.3 Couche réplication

4.3.1 Modèle de données du document texte

MUTE propose plusieurs alternatives pour représenter le document texte. MUTE permet de soit utiliser une implémentation de LogootSplit [22] (cf. section 2.4, page 52), soit de RenamableLogootSplit[22] (cf. chapitre 3, page 68) ou soit de Dotted LogootSplit [23] [29]. 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 livrée 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 [22] au document. De façon similaire, nous utilisons un First-Writer-Wins Register CRDT synchronisé par opérations pour représenter la date de création du document.

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

23. Implémentation fournie par la librairie suivante : `https://github.com/coast-team/dotted-logootsplit`

4.3.2 Collaborateur-rices

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 collaborateur-rices actuellement connectés. Les protocoles d'appartenance au réseau sont une catégorie de protocoles spécifiquement dédiée à cet effet.

MUTE présente 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 [33]. Ce protocole d'appartenance au réseau offre les propriétés intéressantes suivantes. Tout d'abord, le nombre de messages diffusés sur le réseau est proportionnel linéairement 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 collaborateur-rices cohérente à terme, même en cas de réception désordonnée des messages du protocole. 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 executé 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 4.5.

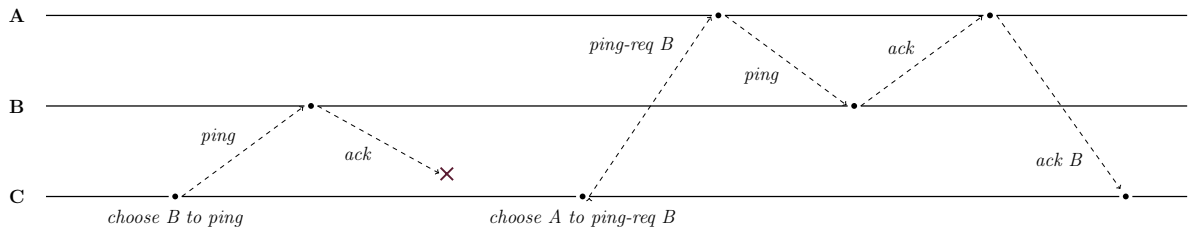


FIGURE 4.5 – Exécution du mécanisme de détection des défaillances par le noeud C pour tester le noeud B

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 devrait prendre 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 de 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.

Afin de représenter la liste des collaborateur-rices, le protocole SWIM utilise la structure de données présentée par la Définition 58 :

Définition 58 (Liste des collaborateur-rices)**.** La *liste des collaborateur-rices* est un ensemble de triplets $\langle nodeId, nodeStatus, nodeIncarn \rangle$ avec :

(i) *nodeId*, l'identifiant du noeud correspondant à ce tuple.

(ii) *nodeStatus*, le 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.

(iii) *nodeIncarn*, le 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 collaborateur-rices, 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 [24] 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 entier, initialisé avec $\lambda \log n$ où n est 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 4.3.2. De manière additionnelle, il intègre dans sa liste des collaborateur-rices les mises à jour attachées au message en utilisant la règle suivante :

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

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 collaborateur-rices sans ajouter de message supplémentaire. De plus, les règles de précédence 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 [33]. Notre première modification porte sur l'ordre de priorité entre les états d'un pair.

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

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

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

Ce choix n'est cependant pas anodin : il implique que la taille de la liste des collaborateur-rices 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 précédence de la manière suivante :

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

24. [33] montre que choisir une valeur constante faible comme λ suffit néanmoins à garantir la dissémination des mises à jour à l'ensemble du réseau.

et

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

Ces modifications permettent de donner la précédence 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 collaborateur-rices devient dès lors linéaire par rapport au nombre de noeuds.

Ces modifications n'ont pas d'impact sur la convergence des listes des collaborateur-rices 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 collaborateur-rices, ce dernier doit donc la demander à un autre pair.

Nous avons donc implémenté pour la liste des collaborateur-rices un mécanisme d'anti-entropie : 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 les défaillances éventuelles du réseau. Ainsi, dans les faits, nous avons mis en place un CRDT synchronisé par différences d'états (cf. section 2.2.2, page 26) pour la liste des collaborateur-rices.

Synthèse

Pour générer et maintenir la liste des collaborateur-rices, nous avons implémenté le protocole distribué d'appartenance au réseau SWIM [33]. Par rapport à la version originale, nous avons procédé à plusieurs modifications, notamment pour gérer plus efficacement les reconnexions 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 ont été proposées dans [34]. 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.

4.3.3 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 collaborateur-rices 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 collaborateur-rices.

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.

4.4 Couche livraison

Comme indiqué précédemment, la couche livraison est formée d'un unique composant, que nous nommons module de livraison. Ce module est associé aux CRDTs synchronisés par opérations représentant le document texte, c.-à-d. LogootSplit ou RenamableLogootSplit.

Le rôle de ce module est de garantir que le modèle de livraison des opérations requis par le CRDT pour assurer la convergence à terme (cf. Définition 11, page 12) soit satisfait, c.-à-d. que l'ensemble des opérations soient livrées dans un ordre correct à l'ensemble des noeuds.

Pour cela, le module de livraison doit implémenter les contraintes imposées par ces CRDTs sur l'ordre de livraison des opérations (cf. Définition 45, page 59 et Définition 50, page 59). Pour rappel, le modèle de livraison de RenamableLogootSplit est le suivant :

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

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

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

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

Nous décrivons ci-dessous comment nous assurons chacune de ces contraintes.

4.4.1 Livraison des opérations en exactement un exemplaire

Afin de respecter la contrainte de livraison en exactement un exemplaire, il est nécessaire d'identifier de manière unique chaque opération. Pour cela, le module de livraison ajoute un *Dot* [75] à chaque opération :

Définition 59 (Dot)**.** Un *Dot* est une paire $\langle nodeId, nodeSyncSeq \rangle$ où

(i) *nodeId*, l'identifiant unique du noeud qui a généré l'opération.

(ii) *nodeSyncSeq*, 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. section 2.4, page 52). 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 de livraison peut livrer l'opération au CRDT et ajouter son dot à la structure. Le cas échéant, cela indique que l'opération a déjà été livrée précédemment et doit être ignorée cette fois-ci.

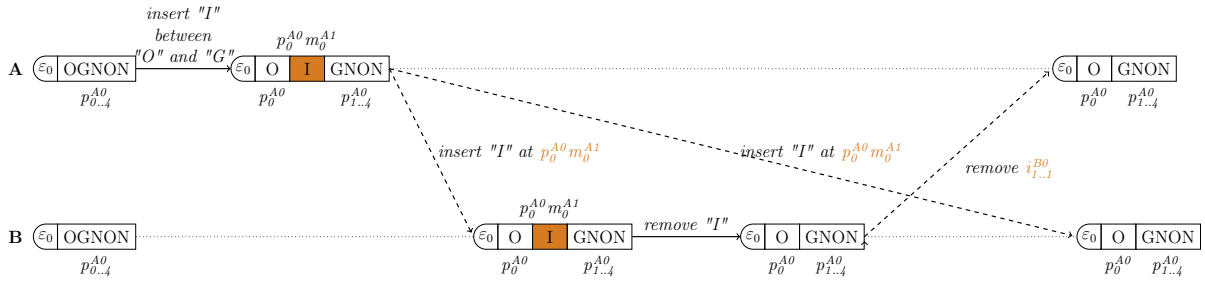
Plusieurs structures de données sont adaptées pour maintenir l'ensemble des opérations reçues. Dans le cadre de MUTE, nous avons choisi d'utiliser un vecteur de version. Cette structure nous permet de réduire à un dot par noeud le surcoût en métadonnées du module de livraison, puisqu'il ne nécessite 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 4.6 illustre son fonctionnement.

Dans cet exemple, qui reprend celui de la Figure 2.24, 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 de livraison de chaque noeud maintient donc initialement le vecteur de version $\langle A : 5 \rangle$.

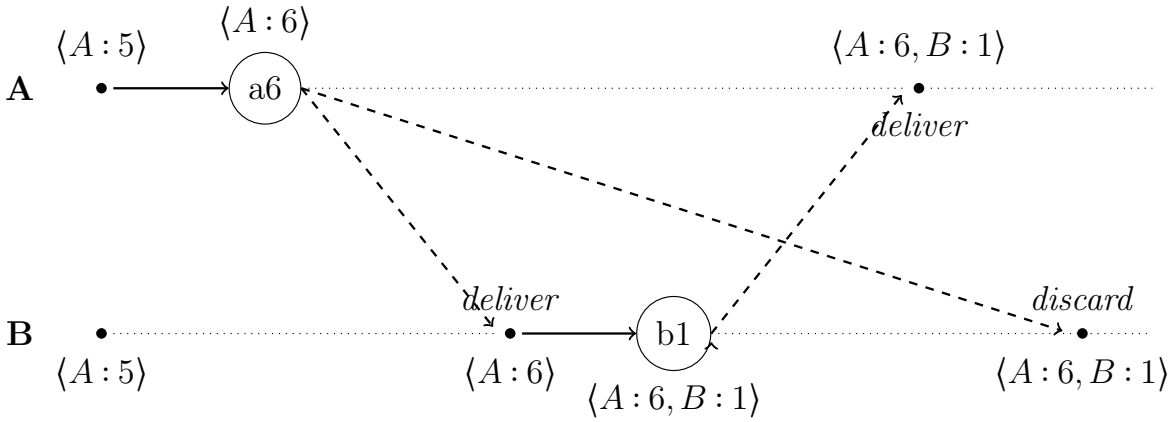
Le noeud A insère l'élément "I" entre les éléments "O" et "G". Cette modification est alors labellisée *a6* par son module de livraison et est envoyée au noeud B. À la réception de cette opération, le module de B compare son dot avec son vecteur de version local. L'opération *a6* étant la prochaine opération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

Finalement, le noeud B reçoit de nouveau l'opération *a6*. Son module de livraison détermine alors qu'il s'agit d'un doublon : l'opération apparaît déjà dans le vecteur de



(a) Exécution avec livraison multiple d'une opération *insert*



(b) État et comportement du module de livraison au cours de l'exécution décrite en Figure 4.6a

FIGURE 4.6 – Gestion de la livraison en exactement un exemplaire des opérations

version $\langle A:6, B:1\rangle$. L'opération est donc ignorée, et la résurgence de l'élément "I" illustrée dans la Figure 2.24 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 opération est mise en attente. En effet, livrer 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* est donc mise en attente jusqu'à la livraison de l'opération *b2*.

Ainsi, l'implémentation de livraison en exactement un exemplaire d'une opération 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 en exactement un exemplaire 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* [105]. Au lieu d'enregistrer seulement le dernier dot intégré par noeud, cette structure de données enregistre les intervalles de dots intégrés. Ceci permet une livraison *dans le désordre* des opérations, c.-à-d. une livraison

des opérations dans un ordre différent de leur ordre d'émission, tout en garantissant une livraison en exactement un exemplaire et en compactant efficacement les données stockées par le module de livraison à terme.

4.4.2 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 livrée après les opérations *insert* insérant les éléments concernés.

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

Définition 60 (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 4.7.

Cet exemple reprend et complète celui de la Figure 2.25. 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 $\langle A:5\rangle$.

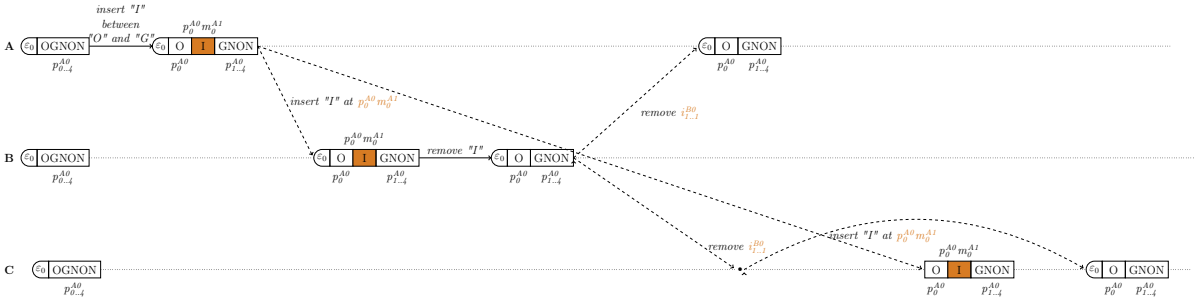
Le noeud A insère l'élément "I" entre les éléments "O" et "G". Cet élément se voit attribué l'identifiant $p_0^{A0}m_0^{A1}$. L'opération correspondante *a6* est diffusée aux autres noeuds.

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 de livraison calcule ses dépendances avant de la diffuser. Pour chaque élément supprimé ("I"), le module de livraison récupère l'identifiant de l'élément ($p_0^{A0}m_0^{A1}$) et en extrait l'identifiant du noeud qui l'a inséré (A). Le module 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.

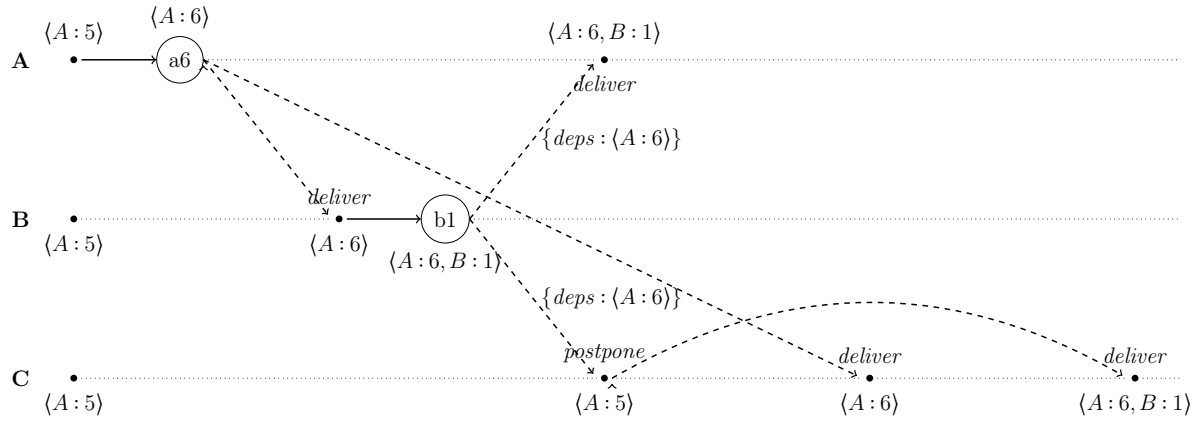
À la réception de l'opération *b1*, le noeud A vérifie s'il possède l'ensemble des dépendances de l'opération. Le noeud A ayant déjà intégré l'opération *a6*, le module de livraison livre l'opération *b1* au CRDT.

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 livrée. Le module de livraison ré-évalue alors le cas de l'opération *b1* et détermine qu'elle peut à présent être livrée.

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



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



(b) État et comportement du module de livraison au cours de l'exécution décrite en Figure 4.7a

FIGURE 4.7 – Gestion de la livraison des opérations *remove* après les opérations *insert* correspondantes

des éléments supprimés. Nous n'ajoutons pas les dots des opérations qui ont spécifiquement inséré les éléments supprimés. Pour cela, il serait nécessaire de parcourir le journal des opérations à la recherche des opérations *insert* correspondantes. Cette méthode serait plus coûteuse, sa complexité dépendant du nombre d'opérations dans le journal des opérations, et incompatible avec un mécanisme tronquant le journal 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.

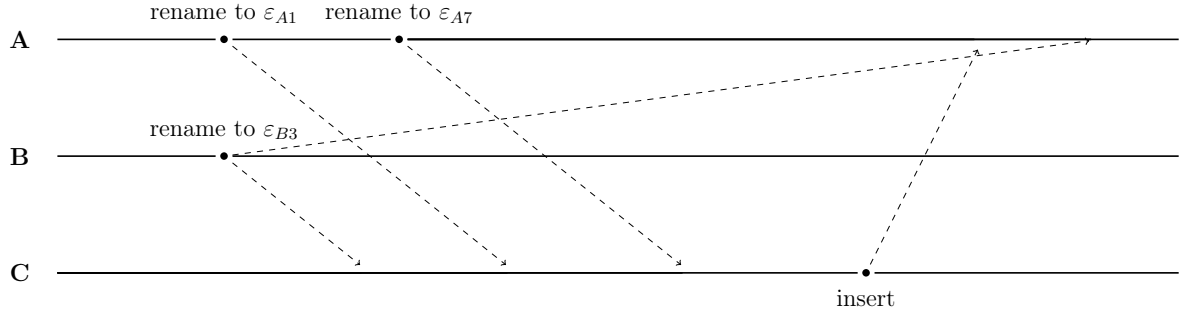
4.4.3 Livraison des opérations après l'opération *rename* introduisant leur époque

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

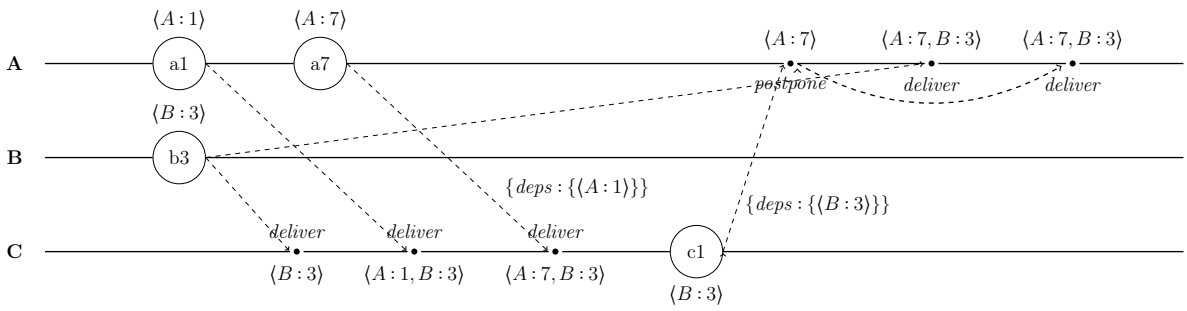
Pour cela, le module de livraison doit donc récupérer l'époque courante de la séquence répliquée, récupérer le dot de l'opération *rename* l'ayant introduite et l'ajouter en tant que dépendance de chaque opération. Cependant, dans notre implémentation, le module

de livraison 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 de livraison 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. Le module 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 4.8 illustre ce fonctionnement.



(a) Exécution



(b) État et comportement du module de livraison au cours de l'exécution décrite en Figure 4.8a

FIGURE 4.8 – Gestion de la livraison des opérations après l'opération *rename* qui introduit leur époque

Dans la Figure 4.8a, 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 4.8b, nous faisons apparaître l'état du module de livraison 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 de livraison de C peut la livrer 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 livre et ajoute le dot $\langle A : 1 \rangle$. Le module de livraison 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 livrée précédemment. L'opération est donc livrée 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 de livraison 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 le module de livraison, l'opération *b3* n'ayant pas encore été livrée. Le noeud reçoit ensuite l'opération *b3*. Son vecteur des dots des opérations *rename* est mis à jour et l'opération livrée. Les conditions pour l'opération *c1* étant désormais remplies, l'opération est alors livrée.

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.

4.4.4 Livraison des opérations à terme

La contrainte restante du modèle de livraison précise que toutes les opérations doivent être livrées à l'ensemble des 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 [55]. Ce mécanisme permet à un noeud source de se synchroniser avec un autre noeud cible. Il est executé 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 journal 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 4.9 illustre ce mécanisme.

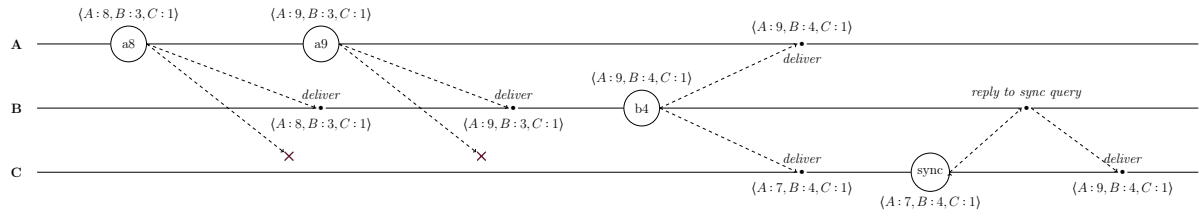


FIGURE 4.9 – Utilisation du mécanisme d'anti-entropie par le noeud C pour se synchroniser avec le noeud B

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 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 livre à son CRDT. Il effectue ensuite et propage l'opération *b4*, qui est reçue et livrée 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 livrer 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 journal 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 livre 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 en exactement un exemplaire, comme présenté en sous-section 4.4.1. 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 journal 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 journal 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 journal comme évoqué en sous-section 3.5.6, ce journal des opérations croit 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.

4.5 Couche réseau

Pour permettre aux différents noeuds de communiquer, MUTE repose sur la librairie Netflux [25]. Développée au sein de l'équipe Coast, cette librairie permet de construire un réseau P2P entre des navigateurs, mais aussi des agents logiciels.

4.5.1 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 API [26] 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.

Cette API utilise pour cela un ensemble de protocoles. Ces protocoles réintroduisent des serveurs dans l'architecture système de MUTE. Dans la Figure 4.10, nous représentons une collaboration réalisée avec MUTE, composé de noeuds formant un réseau P2P, de différents serveurs nécessaires à la mise en place du réseau P2P. Finalement, nous représentons les interactions entre les noeuds et ces serveurs.

Nous décrivons ci-dessous le rôle respectif de chaque type de serveur dans la collaboration.

25. `https://github.com/coast-team/netflux`
26. Application Programming Interface (API) : Interface de Programmation

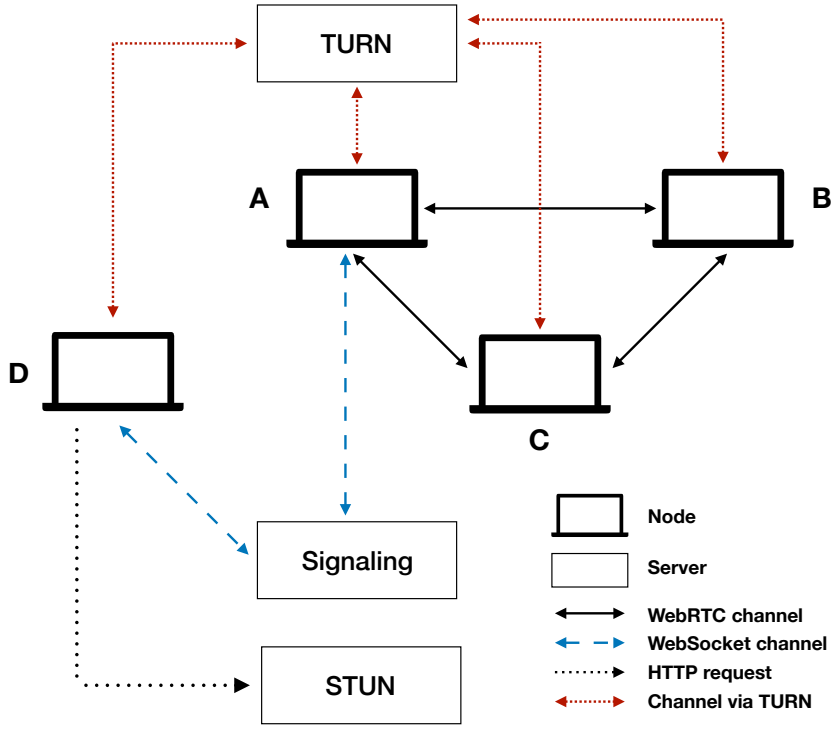


FIGURE 4.10 – Architecture système pour la couche réseau de MUTE

Serveur de signalisation

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

Au moins un noeud du réseau P2P doit maintenir une connexion avec le serveur de signalisation. À sa connexion, un nouveau noeud contacte le serveur de signalisation. Il est mis en relation avec un noeud 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 noeud contacte et communique avec les autres noeuds par l'intermédiaire du premier noeud. Il peut alors terminer sa connexion avec le serveur de signalisation.

Serveur STUN

Pour se connecter, les noeuds doivent s'échanger plusieurs informations logicielles et matérielles, notamment leur adresse IP publique respective. Cependant, un noeud 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 connexion 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.

Rôles 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 établit, 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.

4.5.2 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 [35], pourraient être implémentées pour adresser cette limite.

4.6 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 [36]. Il nécessite que chaque noeud possède une paire de clés de chiffrement et enregistre sa clé publique auprès d'un PKI [27].

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

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 [31, 32]. Son fonctionnement nécessite l'utilisation d'un registre publique sécurisé *append-only*, c.-à-d. une blockchain.

L'architecture système nécessaire pour la couche sécurité est présentée dans la Figure 4.11.

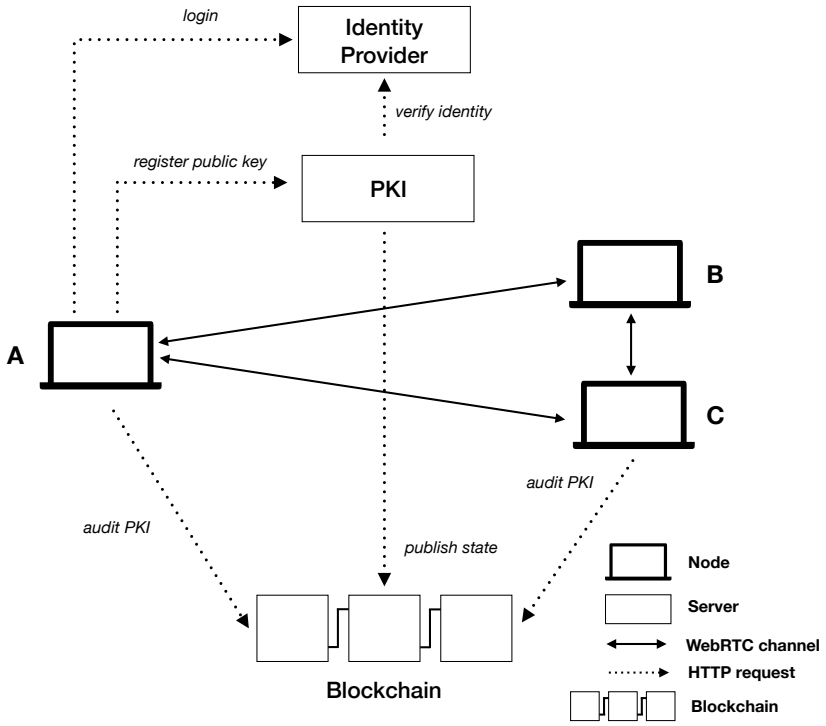


FIGURE 4.11 – Architecture système pour la couche sécurité de MUTE

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 à chaque évolution des noeuds connectés : à chaque connexion ou déconnexion d'un noeud, une nouvelle clé est recalculée avec les collaborateur-rices présents. Ce facteur de changement de la clé de chiffrement, nécessaire pour garantir la *backward secrecy* et *forward secrecy* (cf. Définition 56, page 116 et Définition 57, page 116), induit plusieurs problèmes.

Tout d'abord, ce facteur nous empêche de réutiliser cette même clé de chiffrement pour mettre en place un mécanisme de stockage des opérations chiffrées chez un ou des

tiers, e.g. sur des noeuds du réseau P2P extérieurs à la collaboration ou sur des agents de messages.

Le stockage des opérations chiffrées chez des tiers est une fonctionnalité qui améliorerait l'utilisabilité de l'application sans sacrifier la confidentialité des données. En effet, elle permettrait à un noeud déconnecté de manière temporaire de récupérer à sa reconnexion les modifications effectuées entretemps par ses collaborateur-rices, même si celleux-ci se sont depuis déconnecté-es.

Cependant, la clé de chiffrement est modifiée à la déconnexion du noeud. Ainsi, les opérations suivantes sont chiffrées avec une nouvelle clé que le noeud déconnecté ne possède pas. À sa reconnexion, ce dernier ne sera pas en mesure de déchiffrer et d'intégrer les opérations effectuées en son absence.

L'évolution de la clé de chiffrement de groupe à chaque connexion ou déconnexion d'un noeud est donc incompatible avec l'utilisation de cette même clé pour le stockage sécurisé des opérations chez des tiers. Une autre clé de chiffrement, dédiée, devrait donc être mise en place.

Une seconde limite liée à ce facteur d'évolution est la complexité en temps du protocole de génération de la clé de groupe. En effet, nos évaluations ont montré que ce protocole met jusqu'à 6 secondes pour s'exécuter.

Dans le cadre d'un système P2P à large échelle sujet au churn, des périodes d'activités où des noeuds se connectent et déconnectent toutes les 6 secondes sont à envisager. Lors de tels pics d'activités, les noeuds seraient incapables de collaborer, faute de clé de chiffrement de groupe. Il convient donc soit d'étudier l'utilisation d'autres protocoles de génération de clés de chiffrement de groupe plus efficaces, soit de considérer relaxer les garanties de *backward secrecy* et de *forward secrecy* dans le cadre des collaborations à large échelle.

4.7 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 2.4 et le chapitre 3. 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] Jonathan A Obar et Steven S Wildman. « Social Media Definition and the Governance Challenge - An Introduction to the Special Issue ». In : *Obar, JA and Wildman, S.(2015). Social media definition and the governance challenge : An introduction to the special issue. Telecommunications policy* 39.9 (2015), p. 745–750.

[2] Statista. *Biggest social media platforms 2022*. Last Accessed : 2022-10-06. URL : https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/.

[3] Wikimedia. *Wikimedia Statistics - English Wikipedia*. Last Accessed : 2022-10-06. URL : https://stats.wikimedia.org/#/en.wikipedia.org.

[4] David Meek. « YouTube and Social Movements : A Phenomenological Analysis of Participation, Events and Cyberplace ». In : *Antipode* 44.4 (2012), p. 1429–1448. DOI : https://doi.org/10.1111/j.1467-8330.2011.00942.x. eprint : https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8330.2011.00942.x. URL : https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8330.2011.00942.x.

[5] Yannis Theocharis. « The wealth of (occupation) networks ? Communication patterns and information distribution in a Twitter protest network ». In : *Journal of Information Technology & Politics* 10.1 (2013), p. 35–56.

[6] José Van Dijck et Thomas Poell. « Understanding social media logic ». In : *Media and communication* 1.1 (2013), p. 2–14.

[7] Jim Giles. « Special Report Internet encyclopaedias go head to head ». In : *nature* 438.15 (2005), p. 900–901.

[8] Lada A Adamic, Jun Zhang, Eytan Bakshy et Mark S Ackerman. « Knowledge sharing and yahoo answers : everyone knows something ». In : *Proceedings of the 17th international conference on World Wide Web*. 2008, p. 665–674.

[9] Paul Baran. « On distributed communications networks ». In : *IEEE transactions on Communications Systems* 12.1 (1964), p. 1–9.

[10] Safiya Umoja Noble. *Algorithms of Oppression : How Search Engines Reinforce Racism*. NYU Press, 2018. ISBN : 9781479849949.

[11] Amnesty International. *#Toxictwitter : Violence and abuse against women online*. Last Accessed : 2022-10-07. URL : https://www.amnesty.org/en/documents/act30/8070/2018/en/.

[12] Wall Street Journal. *Facebook Tried to Make Its Platform a Healthier Place. It Got Angrier Instead.* Last Accessed : 2022-10-07. URL : https://t.co/P6JohMdhQE.

[13] Wall Street Journal. *Facebook Knows Instagram Is Toxic for Teen Girls, Company Documents Show.* Last Accessed : 2022-10-07. URL : https://t.co/JAvzKFc61q.

[14] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg et Mark McGranaghan. « 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.

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

[16] 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.

[17] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer et Carl H Hauser. « Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System ». In : *SIGOPS Oper. Syst. Rev.* 29.5 (déc. 1995), p. 172–182. ISSN : 0163-5980. DOI : 10.1145/224057.224070. URL : https://doi.org/10.1145/224057.224070.

[18] Daniel Stutzbach et Reza Rejaie. « Understanding Churn in Peer-to-Peer Networks ». In : *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement.* IMC '06. Rio de Janeriro, Brazil : Association for Computing Machinery, 2006, p. 189–202. ISBN : 1595935614. DOI : 10.1145/1177080.1177105. URL : https://doi.org/10.1145/1177080.1177105.

[19] Leslie Lamport. « The part-time parliament ». In : *Concurrency : the Works of Leslie Lamport.* 2019, p. 277–317.

[20] Diego Ongaro et John Ousterhout. « In search of an understandable consensus algorithm ». In : *2014 USENIX Annual Technical Conference (Usenix ATC 14).* 2014, p. 305–319.

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

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

[23] 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.

[24] 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.

[25] Google. *Google Docs*. Last Accessed : 2022-10-07. URL : https://docs.google.com/.

[26] Cody Odgen. *Google Graveyard*. Last Accessed : 2022-10-11. URL : https://killedbygoogle.com/.

[27] Matthieu Nicolas, Victorien Elvinger, Gérald Oster, Claudia-Lavinia Ignat et François Charoy. « 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.

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

[29] 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.

[30] Matthieu Nicolas, Gerald Oster et Olivier Perrin. « Efficient Renaming in Sequence CRDTs ». In : *IEEE Transactions on Parallel and Distributed Systems* 33.12 (déc. 2022), p. 3870–3885. DOI : 10.1109/TPDS.2022.3172570. URL : https://hal.inria.fr/hal-03772633.

[31] 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.

[32] Hoang-Long Nguyen, Jean-Philippe Eisenbarth, Claudia-Lavinia Ignat et Olivier Perrin. « 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.

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

[34] 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.

[35] Brice Nédelec, Julian Tanke, Davide Frey, Pascal Molli et Achour Mostéfaoui. « An adaptive peer-sampling protocol for building networks of browsers ». In : *World Wide Web* 21.3 (2018), p. 629–661.

[36] 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.

[37] 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.

[38] 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.

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

[40] Leslie Lamport. « Time, Clocks, and the Ordering of Events in a Distributed System ». In : *Commun. ACM* 21.7 (juil. 1978), p. 558–565. issn : 0001-0782. doi : 10.1145/359545.359563. url : https://doi.org/10.1145/359545.359563.

[41] Nuno M. Preguiça, Carlos Baquero et Marc Shapiro. « Conflict-free Replicated Data Types (CRDTs) ». In : *CoRR* abs/1805.06358 (2018). arXiv : 1805.06358. url : http://arxiv.org/abs/1805.06358.

[42] Nuno M. Preguiça. « Conflict-free Replicated Data Types : An Overview ». In : *CoRR* abs/1806.10254 (2018). arXiv : 1806.10254. url : http://arxiv.org/abs/1806.10254.

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

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

[45] Weihai Yu et Sigbjørn Rostad. « A Low-Cost Set CRDT Based on Causal Lengths ». In : *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. New York, NY, USA : Association for Computing Machinery, 2020. ISBN : 9781450375245. URL : https://doi.org/10.1145/3380787.3393678.

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

[47] 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.

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

[49] 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.

[50] 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.

[51] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin et al. « Consistency, availability, and convergence ». In : *University of Texas at Austin Tech Report* 11 (2011), p. 158.

[52] Friedemann Mattern et al. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[53] Colin Fidge. « Logical Time in Distributed Computing Systems ». In : *Computer* 24.8 (août 1991), p. 28–33. ISSN : 0018-9162. DOI : 10.1109/2.84874. URL : https://doi.org/10.1109/2.84874.

[54] Ravi Prakash, Michel Raynal et Mukesh Singhal. « An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments ». In : *Journal of Parallel and Distributed Computing* 41.2 (1997), p. 190–204. ISSN : 0743-7315. DOI : https://doi.org/10.1006/jpdc.1996.1300. URL : https://www.sciencedirect.com/science/article/pii/S0743731596913003.

[55] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser et C. Kline. « Detection of Mutual Inconsistency in Distributed Systems ». In : *IEEE Trans. Softw. Eng.* 9.3 (mai 1983), p. 240–247. ISSN : 0098-5589. DOI : 10.1109/TSE.1983.236733. URL : https://doi.org/10.1109/TSE.1983.236733.

[56] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall et Werner Vogels. « Dynamo : Amazon's highly available key-value store ». In : *ACM SIGOPS operating systems review* 41.6 (2007), p. 205–220.

[57] 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.

[58] Ricardo Jorge Tomé Gonçalves, Paulo Sérgio Almeida, Carlos Baquero et Vitor Fonte. « DottedDB : Anti-Entropy without Merkle Trees, Deletes without Tombstones ». In : *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. 2017, p. 194–203. DOI : 10.1109/SRDS.2017.28.

[59] Jim Bauwens et Elisa Gonzalez Boix. « Improving the Reactivity of Pure Operation-Based CRDTs ». In : *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '21. Online, United Kingdom : Association for Computing Machinery, 2021. ISBN : 9781450383387. DOI : 10.1145/3447865.3457968. URL : https://doi.org/10.1145/3447865.3457968.

[60] 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.

[61] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero et João Leitão. « Efficient Synchronization of State-Based CRDTs ». In : *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, p. 148–159. DOI : 10.1109/ICDE.2019.00022.

[62] Clarence A. Ellis et Simon J. Gibbs. « Concurrency Control in Groupware Systems ». In : *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. Portland, Oregon, USA : Association for Computing Machinery, 1989, p. 399–407. ISBN : 0897913175. DOI : 10.1145/67544.66963. URL : https://doi.org/10.1145/67544.66963.

[63] Chengzheng Sun et Clarence Ellis. « Operational transformation in real-time group editors : issues, algorithms, and achievements ». In : *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. 1998, p. 59–68.

[64] Matthias Ressel, Doris Nitsche-Ruhland et Rul Gunzenhäuser. « An integrating, transformation-oriented approach to concurrency control and undo in group editors ». In : *Proceedings of the 1996 ACM conference on Computer supported cooperative work*. 1996, p. 288–297.

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

[66] David Sun et Chengzheng Sun. « Context-Based Operational Transformation in Distributed Collaborative Editing Systems ». In : *Parallel and Distributed Systems, IEEE Transactions on* 20 (nov. 2009), p. 1454–1470. doi : 10.1109/TPDS.2008.240.

[67] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang et David Chen. « Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems ». In : *ACM Transactions on Computer-Human Interaction (TOCHI)* 5.1 (1998), p. 63–108.

[68] Gérald Oster, Pascal Molli, Pascal Urso et Abdessamad Imine. « Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems ». In : *2006 International Conference on Collaborative Computing : Networking, Applications and Worksharing*. 2006, p. 1–10. doi : 10.1109/COLCOM.2006.361867.

[69] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang et David Chen. « Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems ». In : *ACM Trans. Comput.-Hum. Interact.* 5.1 (mar. 1998), p. 63–108. issn : 1073-0516. doi : 10.1145/274444.274447. url : https://doi.org/10.1145/274444.274447.

[70] 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.

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

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

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

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

[75] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça et Victor Fonte. « 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.

[76] Charbel Rahhal, Stéphane Weiss, Hala Skaf-Molli, Pascal Urso et Pascal Molli. *Undo in Peer-to-peer Semantic Wikis*. Research Report RR-6870. INRIA, 2009, p. 18. URL : https://hal.inria.fr/inria-00366317.

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

[78] 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.

[79] Ben Shneiderman. « Response Time and Display Rate in Human Performance with Computers ». In : *ACM Comput. Surv.* 16.3 (sept. 1984), p. 265–285. ISSN : 0360-0300. DOI : 10.1145/2514.2517. URL : https://doi.org/10.1145/2514.2517.

[80] Caroline Jay, Mashhuda Glencross et Roger Hubbold. « Modeling the Effects of Delayed Haptic and Visual Feedback in a Collaborative Virtual Environment ». In : *ACM Trans. Comput.-Hum. Interact.* 14.2 (août 2007), 8–es. ISSN : 1073-0516. DOI : 10.1145/1275511.1275514. URL : https://doi.org/10.1145/1275511.1275514.

[81] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang et Marek Zawirski. « Specification and Complexity of Collaborative Text Editing ». In : *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. PODC '16. Chicago, Illinois, USA : Association for Computing Machinery, 2016, p. 259–268. ISBN : 9781450339643. DOI : 10.1145/2933057.2933090. URL : https://doi.org/10.1145/2933057.2933090.

[82] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang et Marek Zawirski. « Specification and space complexity of collaborative text editing ». In : *Theoretical Computer Science* 855 (2021), p. 141–160. issn : 0304-3975. doi : https://doi.org/10.1016/j.tcs.2020.11.046. url : http://www.sciencedirect.com/science/article/pii/S0304397520306952.

[83] Automerge. *Automerge : data structures for building collaborative applications in Javascript*. Last Accessed : 2022-10-07. url : https://github.com/automerge/automerge.

[84] 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.

[85] 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.

[86] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan et Alastair R. Beresford. « Interleaving Anomalies in Collaborative Text Editors ». In : *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '19. Dresden, Germany : Association for Computing Machinery, 2019. isbn : 9781450362764. doi : 10.1145/3301419.3323972. url : https://doi.org/10.1145/3301419.3323972.

[87] Matthew Weidner. *There Are No Doubly Non-Interleaving List CRDTs*. Last Accessed : 2022-10-07. url : https://mattweidner.com/assets/pdf/List_CRDT_Non_Interleaving.pdf.

[88] 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.

[89] Claudia-Lavinia Ignat, Gérald Oster, Meagan Newman, Valerie Shalin et François Charoy. « Studying the Effect of Delay on Group Performance in Collaborative Editing ». In : *Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014, Springer 2014 Lecture Notes in Computer Science*. Proceedings of 11th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2014. Seattle, WA, United States, sept. 2014, p. 191–198. doi : 10.1007/978-3-319-10831-5_29. url : https://hal.archives-ouvertes.fr/hal-01088815.

[90] Claudia-Lavinia Ignat, Gérald Oster, Olivia Fox, François Charoy et Valerie Shalin. « How Do User Groups Cope with Delay in Real-Time Collaborative Note Taking ». In : *European Conference on Computer Supported Cooperative Work*

2015. Sous la dir. de Nina Boulus-Rodje, Gunnar Ellingsen, Tone Bratteteig, Margunn Aanestad et Pernille Bjorn. Proceedings of the 14th European Conference on Computer Supported Cooperative Work. Oslo, Norway : Springer International Publishing, sept. 2015, p. 223–242. doi : 10.1007/978-3-319-20499-4_12. url : https://hal.inria.fr/hal-01238831.

[91] Brice Nédelec, Pascal Molli, Achour Mostéfaoui et Emmanuel Desmontils. « LSEQ : an adaptive structure for sequences in distributed collaborative editing ». In : *Proceedings of the 2013 ACM Symposium on Document Engineering*. DocEng 2013. Sept. 2013, p. 37–46. doi : 10.1145/2494266.2494278.

[92] 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.

[93] 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.

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

[95] 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.

[96] Etherpad. *Etherpad*. Last Accessed : 2022-10-07. url : https://etherpad.org/.

[97] 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.

[98] Barton Gellman et Laura Poitras. *U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program*. Last Accessed : 2022-10-07. 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.

[99] Glen Greenwald et Ewen MacAskill. *NSA Prism program taps in to user data of Apple, Google and others*. Last Accessed : 2022-10-07. url : https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data.

[100] 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.

[101] Jim Pick. *PeerPad*. Last Accessed : 2022-10-07. URL : https://peerpad.net/.

[102] Jim Pick. *Graf, Nikolaus*. Last Accessed : 2022-10-07. URL : https://www.serenity.re/en/notes.

[103] 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.

[104] John Gruber. *Daring Fireball : Markdown*. Last Accessed : 2022-10-17. URL : https://daringfireball.net/projects/markdown/.

[105] 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.

[106] Victorien Elvinger, Gérald Oster et Francois Charoy. « Prunable Authenticated Log and Authenticable Snapshot in Distributed Collaborative Systems ». In : *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. 2018, p. 156–165. DOI : 10.1109/CIC.2018.00031.

[107] OpenRelay. *OpenRelay*. Last Accessed : 2022-10-07. URL : https://openrelay.xyz/.

[108] Protocol Labs. *IPFS*. Last Accessed : 2022-10-07. URL : https://ipfs.io/.

[109] Quang Vinh Dang et Claudia-Lavinia Ignat. « Quality Assessment of Wikipedia Articles : A Deep Learning Approach by Quang Vinh Dang and Claudia-Lavinia Ignat with Martin Vesely as Coordinator ». In : *SIGWEB Newsl.* Autumn (nov. 2016). ISSN : 1931-1745. DOI : 10.1145/2996442.2996447. URL : https://doi.org/10.1145/2996442.2996447.

[110] Leslie Lamport, Robert Shostak et Marshall Pease. « The Byzantine Generals Problem ». In : *Concurrency : The Works of Leslie Lamport*. New York, NY, USA : Association for Computing Machinery, 2019, p. 203–226. ISBN : 9781450372701. URL : https://doi.org/10.1145/3335772.3335936.

[111] Jim Bauwens et Elisa Gonzalez Boix. « Flec : A Versatile Programming Framework for Eventually Consistent Systems ». 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 : 10.1145/3380787.3393685. URL : https://doi.org/10.1145/3380787.3393685.